PSoC3/5 Reference Book

Edward H. Currie and David Van Ess

March 29, 2010

Copyright ©2010, Cypress Semiconductor Corporation.
All rights reserved. This work may not be translated or copied either in whole, or in part, without the prior written permission of Cypress Semiconductor Corporation, 198 Champion Court, San Jose, CA 95134. USA Tel: (408) 943 2600, Fax: (408) 943 4730, except for brief excerpts in connection with reviews, or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar, or dissimilar, methodology now known,or hereafter developed, is strictly forbidden.

The authors and Cypress Semiconductor Corporation have made every effort in the preparation of this textbook to ensure the accuracy of the information. However, the information contained herein is provided and intended for pedagogical purposes only, and without any warranty, either express or implied. Neither the authors, nor Cypress Semiconductor Corporation will be held liable for any damages caused, or alleged to be caused, either directly, or indirectly, by this textbook and/or its contents.

Camera-ready copy was prepared using the authors' LaTeX files.

Printed and bound by xxxxxxxxxxxxxxxx in xxxxxxxxxxx, xxxxxxxxxx USA.

Printed in the United States of America.
9 8 7 6 5 4 3 2 1
ISBN XXX-X-XXXXXXX-X-X Cypress Semiconductor Corporation, San Jose, California.

The information contained herein is subject to change without notice. Cypress Semiconductor Corporation (Cypress) assumes no responsibility for the use of any circuitry herein other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Cypress, Programmable System-on-Chip, PSoC, PSoC Creator, PSoC Designer, and PSoC Express are trademarks of Cypress Semiconductor Corporation, Inc., 2010. All other trademarks, or registered trademarks, are the sole and exclusive property of their respective owners.

The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges. Any Source Code (software and/or firmware) contained herein is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code or any of the material contained herein except as specified above is prohibited without the express written permission of Cypress. Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges. Use may be limited by and subject to the applicable Cypress software license agreement. PSoC Designer, and PSoC Creator are trademarks and PSoC is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations. Flash Code Protection Cypress products meet the specifications contained in their particular Cypress PSoC Data Sheets. Cypress believes that its family of PSoC products is one of the most secure families of its kind on the market today, regardless of how they are used. There may be methods, unknown to Cypress, that can breach the code protection features. Any of these methods, to our knowledge, would be dishonest and possibly illegal. Neither Cypress nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable." Cypress is willing to work with the customer who is concerned about the integrity of their code. Code protection is constantly evolving. We at Cypress are committed to continuously improving the code protection features of our products.Cypress, the Cypress logo, PSoC, PowerPSoC, CapSense and West Bridge are registered trademarks and PSoC Creator and TrueTouch are trademarks of Cypress Semiconductor Corp. All other trademarks are property of their owners.

Preface

This textbook is intended to provide a unique, in depth look at programmable system on a chip technology from the perspective of the world's most advanced system-on-a-chip technology, viz., Cypress Semiconductor's PSoC. The book introduces a wide variety of topics and information intended to facilitate your use of true visual embedded design techniques and mixed-signal technology. The authors have attempted to include sufficient background material and illustrative examples to allow the first time PSoC user, as well as, advanced users to quickly "come up to speed". A detailed bibliography, and other sourcing of useful supplementary materials, is also provided.

Readers are encouraged to visit Cypress' website at www.cypress.com and explore the wealth of material available there in the form of user forums, application notes, design examples,product data sheets, the latest versions of development tools (which are provided to users at no cost), data sheets on all Cypress products, detailed information regarding Cypress' University Alliance programs, tutorials, etc. Cypress has amassed a vast wealth of material related to the subject matter of this text, most of which is accessible to readers online,and the authors have shown a complete lack of restraint in harvesting important concepts, illustrations, examples and source code from this source.

The presentation style employed in this textbook is based on the authors' desire to provide relevant and definitive insight into the material under discussion while circumnavigating a swamp of details in which the reader might otherwise become mired. Thus mathematical derivations are provided in what, in some cases, may be viewed as excruciating detail to best accomplish the task at hand. But various forms of metaphorical "syntactic sugar", or a reasonable facsimile thereof, as well as, other forms of embroidery have been freely and liberally applied as required, to leave the reader with both a strong intuitive grasp of the material and a solid foundation. However, true mathematical rigor in the sense that purists and theoretical mathematicians prefer has been largely avoided.

To the extent feasible, the authors have avoided colloquialisms such as "RAM memory" (which is literally "random access memory memory", employed accepted abbreviations, and mnemonics, hopefully without sacrificing clarity and becoming bogged down in details which may well be required for completeness but are often of little real world applicability, or value.

Errors found in this work are the sole and exclusive property of the authors, but much of the material found here is either directly, or indirectly, the result of the efforts of many of the employees of Cypress, and of course Cypress' customers. The authors welcome your suggestions and criticisms and ask that you forward your comments to xxx@cypress.com.

Contents

Preface . . . iv

1 Introduction to Embedded System **1**

1.1 Evolution of Embedded Systems . . . 1
1.2 Evolution of Microprocessors . . . 3
1.3 Embedded System Applications . . . 15
1.4 Embedded System Controlling . . . 17
1.4.1 Types of Embedded System . . . 17
1.4.2 Open Loop, Closed Loop and Feedback . . . 18
1.5 Embedded System Performance Criteria . . . 22
1.5.1 Interrupts . . . 25
1.5.2 Latency . . . 26
1.6 Embedded Systems Subsystems . . . 29
1.7 Sensors and Sensing . . . 32
1.7.1 Types of Sensors . . . 34
1.7.1.1 Optical Sensors . . . 34
1.7.1.2 Capacitive Sensing . . . 34
1.7.1.3 Magnetic Sensors . . . 35
1.7.1.4 RF . . . 36
1.7.1.5 Ultraviolet . . . 36
1.7.1.6 Infrared . . . 37
1.7.1.7 Ionizing Sensors . . . 37
1.7.1.8 Other Types of sensors . . . 37
1.7.1.9 Thermistors . . . 39
1.7.1.10 Thermocouples . . . 42
1.7.2 Use of Bridges for Temperature Measurement . . . 43
1.7.3 Sensors and Microcontroller Interfaces . . . 45
1.8 Embedded System Processing . . . 46
1.9 Microcontroller Sub-systems . . . 49
1.10 Software Development Environments . . . 54
1.11 Embedded Systems Communications . . . 58
1.11.1 The RS232 Protocol . . . 59
1.11.2 USB . . . 59
1.11.3 Inter-Integrated Circuit Bus (I2C) . . . 60
1.11.4 Serial Peripheral Interface (SPI) . . . 60
1.11.5 Controller Area Network (CAN) . . . 60
1.11.6 Local Interconnect Network (LIN) . . . 62
1.12 Programmable Logic . . . 63
1.13 Mixed-Signal Processing . . . 65
1.14 PSoC - Programmable System on Chip . . . 65

A Mnemonics **71**

B Definitions **73**

Bibliography **84**

Index **86**

List of Figures

1.1 A typical embedded system architecture. 2
1.2 The VERDAN Computer. 3
1.3 External devices required microprocessor-based (micro)controller. 4
1.4 Comparison of Von Neumann and Harvard architectures. 5
1.5 Block diagram of the Intel 8048. 6
1.6 Architecture of a basic microcontroller, e.g.,the Intel 8048. 7
1.7 Accumulator operations 7
1.8 PSW bit positions. 8
1.9 An ultraviolet (UV) erasable microcontroller. 8
1.10 Intel 8048 internal architecture. 9
1.11 Schematic view of an open loop system. 19
1.12 Embedded System motor controller. 20
1.13 Schematic view of a closed loop system with direct feedback. 20
1.14 Schematic view of a closed loop system with “sensed” output feedback. 20
1.15 A generalized SISO feedback system. 21
1.16 An embedded system that is subject to external perturbations 22
1.17 Block diagram of typical microcontroller/DMA configuration. 24
1.18 An example of a tri-state device. 25
1.19 Intel 8051 Architecture. 28
1.20 A simple example of analog signal processing. 31
1.21 Example of aliasing 32
1.22 Examples of Tension, Compression, Flexure (Bending) and Torsion. 39
1.23 Example of shear force. 39
1.24 Strain Gauge applied to a duraluminum tensile test specimen. 40
1.25 Closeup of a Strain Gauge. 41
1.26 Seebeck potentials. 42
1.27 The Wheatstone Bridge. 44
1.28 Constant current measurement. 45
1.29 Resistive divider. 45
1.30 Classification of the types of memory used in/with microcontrollers. 51
1.31 An example of an SRAM Cell. 52
1.32 An example of a dynamic cell. 52
1.33 Driving modes for each pin are programmatically selectable. 54
1.34 Development Tool and Hardware Evolution. 56
1.35 A graphical representation of the simplest form of SPI communication. 58
1.36 SPI - Single master multiple slaves. 58
1.37 Hardware example of the SPI network. 59
1.38 The RS232 protocol (1 start bit, 8 data bit, 1 stop bit). 61
1.39 CAN frame format. 62

1.40 LIN frame format. 63
1.41 Digital Logic family tree. 63
1.42 Unprogrammed PAL. 64
1.43 An example of a programmed PAL. 65
1.44 An example of a multiplexer based on a PLD 65
1.45 PSoC1/PSoC2/PSoC3 architectures. 69

List of Tables

1.1 Some of the types of subsystems available in microcontrollers. 30
1.2 Algorithms used in embedded systems. 47

Chapter 1

Introduction to Embedded System

"Embedded Systems are application-domain specific, information processing systems that are tightly coupled to their environment." Dr. T. Stefano (2008)

1.1 Evolution of Embedded Systems

Embedded systems can be less formally defined as dedicated, computer-based systems designed to monitor certain parameters, associated with some process or system, and to use that information to control the process, or system, or some combination thereof. An embedded system, typically engages in "data processing"[1] for the purposes of data logging and/or control of an external process, system, or systems, by providing various outputs to external devices, as required, based on the available input data. Alternatively, embedded systems may be described as dedicated, microcomputer-based systems[2], operating in real time, running code optimized for execution speed and size that are usually designed to perform specific tasks related to control of a process or system, subject to certain predefined constraints and operating conditions.

The majority of embedded systems are gathering data from one or more sensors and other data sources, subjecting the data to some form of conditioning and providing it directly, or via a buffer/multiplexing stage in cases involving large amounts of data, to a Central Processing Unit (CPU), The CPU "processes" the incoming data and typically outputs commands to an output conditioning phase which in turn drives motors, linear and other actuators, display devices, communications channels, etc., as shown in Figure 1.1.

It is difficult to say when the first all solid state embedded system appeared, particularly when compared to modern embedded systems. However, a very early example occurred as a result of the introduction of a series of inertial navigation systems, developed by Autonetics, a division of North American Rockwell, in the early nineteen fifties. This work included, at least philosophically, a

[1] "Data processing" in the present context refers to the embedded system invoking the appropriate algorithms to determine what action, if any, is to be taken based on sensor, or other input data.

[2] Early embedded systems were generally regarded as unalterable (fixed) in terms of their basic functionality. However, in recent years it has become possible to "reconfigure" these systems in the field as a result of human intervention, or in some cases, embedded systems can actually reconfigure themselves, programmatically, to conform and/or respond to changing operating conditions and environments. This latter property is sometimes to referred to as "reconfigurability". This important property allows an embedded system to more fully exploit all of its hardware resources.

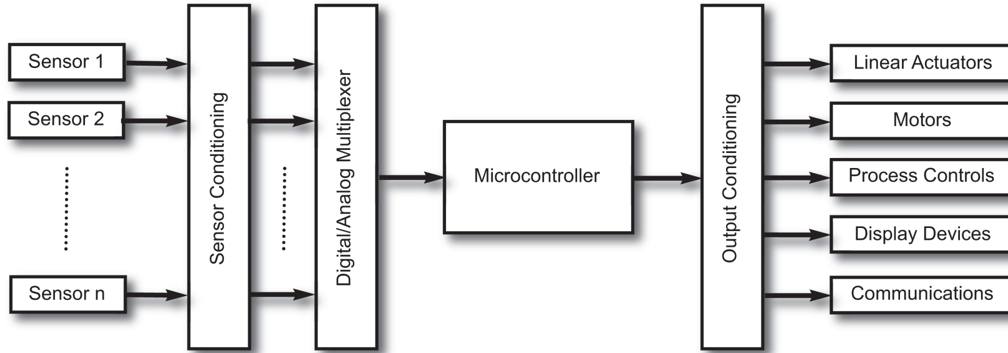


Figure 1.1: A typical embedded system architecture.

predecessor of the modern day microcontroller which was called VERDAN (Versatile Digital Analyzer), shown in Figure 1.2. This system, known by various names (VERDAN, MARDAN, D-9), evolved from work in the latter part of the 1940s by Autonetics, into a fully transistorized, flight control system consisting of some 1500 germanium transistors, 10,670 germanium diodes, 4,500 resistors and 670 capacitors. This was a particularly amazing accomplishment in light of the then known problems with manufacturing germanium transistors and their associated, and for some, infamous thermal "runaway"[3] problems.

The resulting "macro-computer-based" system was able to navigate, and control, airframes capable of operating at speeds well in excess of Mach 3, i.e., greater than 2000 mph. The computer portion of the system consisted of three sections: 1) a General Purpose (GP) section based on a 24-bit data formats and fifty-six 24-bit instructions, and integral multiply/divide[4] hardware, 2) an I/O section capable of handling multiple shaft encoder and resolver I/O channels and 3) an 128 integrator, Digital Differential Analyzer (DDA). The equations of motion, a set of partial differential equations, were solved in the DDA section, based on continual input from an inertial platform. The GP Section interacted with the DDA to update various parameters in the equations of motion. The DDA then solved the equations of motion in real time and the solutions were subsequently passed to the GP section which communicated with the I/O section to output control information and commands to the system's actuators.

VERDAN's architecture was to reappear in several incarnations, e.g., VERDAN II (aka MARDAN) as part of The US Navy's Ships Inertial Navigation System (SINS) and the D17 which was employed in Minute Man Missiles. Each of these consisted of multiple plug-in cards sharing a common bus. Autonetics may have also been the first to consider a floppy disk as a rotating memory device, but ultimately decided on, what appears to have been, the first "rotating disk" memory. It had fixed heads and rotated at speeds comparable to present day hard disks.

VERDAN was repackaged to facilitate field access to the circuitry and renamed MARDAN (Marine Digital Analyzer). The bus structure remained the same and most of the printed circuit boards used in VERDAN, of which there were approximately 75, were fully compatible with

[3]Some transistors exhibit a phenomenon known as "thermal runaway" which results from the fact that the number of free electrons is a function of temperature. Thus as the temperature rises, the current increases leading to additional Ohmic heating and therefore further temperature increases. This type of problem can lead to the destruction of the transistor but also means that circuit behavior based on such devices can become "unpredictable".

[4]Later systems, e.g. Minuteman, were based on the same architecture but supported only hardware multiply. Division was carried out by inverting the divisor and multiplying. Functions such as square root, if needed, were handled programmatically. Multiplication and division results are each 48 bits.

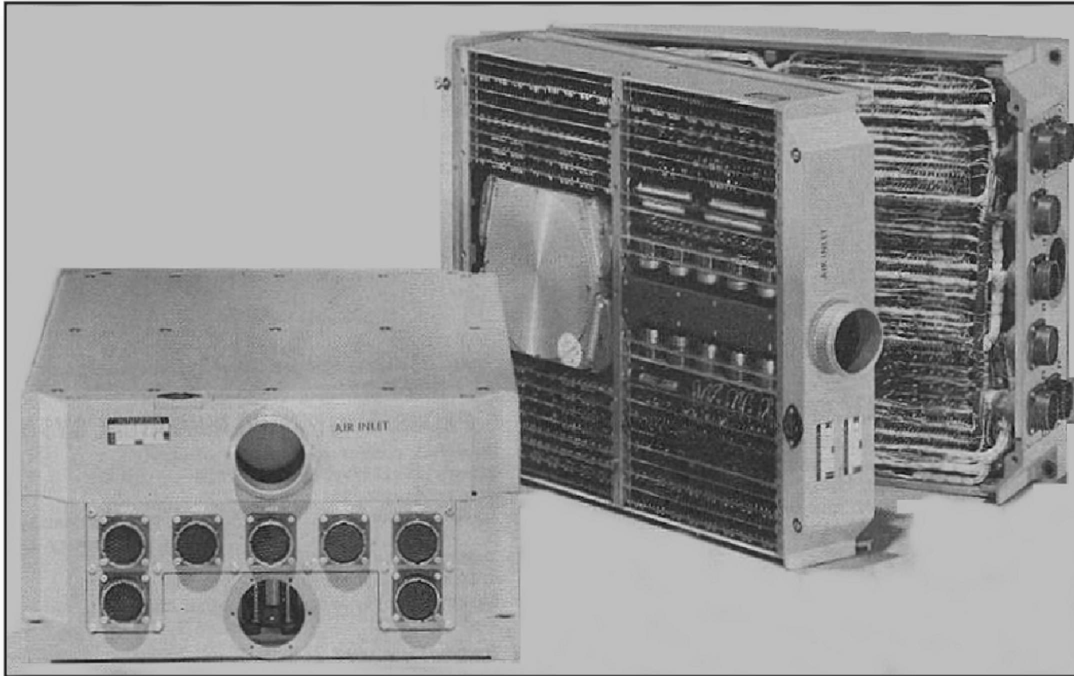


Figure 1.2: The VERDAN Computer.

MARDAN's bus architecture. MARDAN was to remain in service from 1959 until 2005 on the Polaris, Trident, etc., submarines. VERDAN was repackaged for a third time as the navigational computer in the later versions of the Minute Man systems. In addition, some aspects of VERDAN's design were incorporated into the Apollo guidance system and the Apollo simulator.

1.2 Evolution of Microprocessors

The first microprocessor (Intel 4004) was introduced by Intel in 1971. It was followed by the Intel 8008 (1972) and the Intel 8080 (1974). All of these microprocessors required a number of external chips to implement a "useful" computing system and required multiple operating voltages (-5, +5 and +12vdc), cf. Figure 1.3.

In 1977, Intel introduced the 8085, which was a modified form of the 8080, but pt relied less on external chips and required only +5 volts for operation. As the number of microprocessor-based, embedded system applications grew it soon became apparent that utilizing a simple microprocessor, such as the Intel 8080, with its requirement for multiple, associated support chips and the limitations of available external peripheral devices made microprocessors inadequate for many potential and actual embedded system applications.

In 1976, Intel introduced its first true microcontroller with on-board memory and peripherals, known as the "Intel 8048", an N-channel, silicon-gate, MOS device, which was a member of Intel's MCS-48 family of 8-bit microcontrollers. The 8048 had 27 I/O lines , one timer/counter, hardware reset, support for external memory, an 8-bit CPU, interrupt support, hardware single step support and a crystal controlled clock.

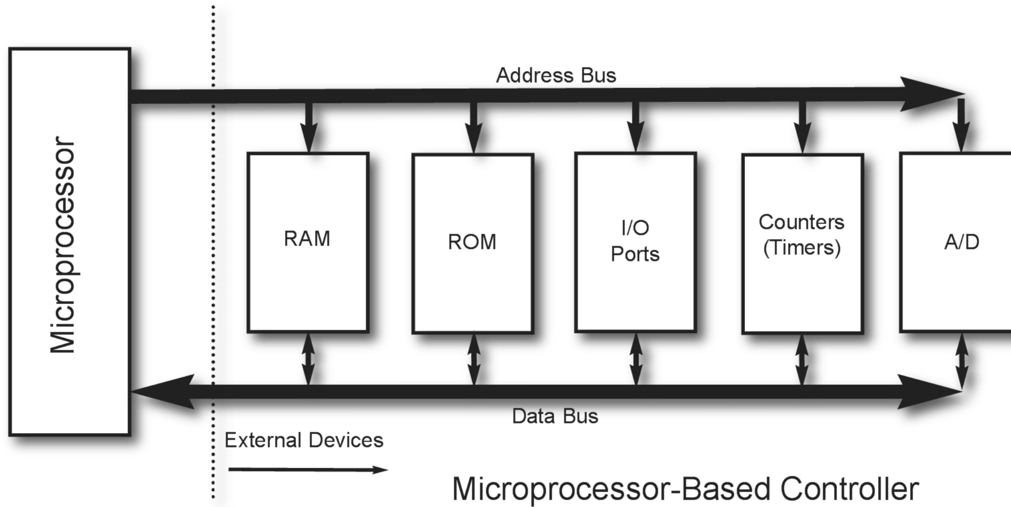


Figure 1.3: External devices required microprocessor-based (micro)controller.

Embedded system microcontrollers, and microprocessors, have historically employed either a Von Neumann, or Harvard, memory architectures, as shown in Figure 1.4. The Von Neumann memory configuration has only one "zero" location for both data and memory and therefore data and program code must reside in the same memory space. Since data and addresses each have their own bus in the Harvard configuration, program and data are stored separately in different regions of memory to allow instructions to be fetched while data is being processed and stored/accessed. Instruction address zero is distinct from data address zero, in the Harvard architecture, which makes it possible for address and data to have different bit sizes, e.g., 8 bit data versus 32 bit instructions.

A third configuration, sometimes referred to as a "modified Harvard" architecture, allows the CPU to access both data and instructions in separate memory spaces, but allows program memory to be accessed by the CPU as if it were data to permit instructions and text to be treated as data, so they can be moved and/or modified.

The 8048, shown schematically in Figure 1.5, had a "modified" Harvard architecture, 64-256 bytes of on-chip RAM, 96+ instructions with 90% being single-byte instructions (all instructions required either one or two cpu cycles). The 8048 handled both binary and BCD arithmetic, had an 8-bit counter/timer, twenty seven I/O lines, an on-chip oscillator which served as the clock. two single-level interrupts, and support for both internal and external ROM/RAM.

I/O was memory-mapped in its own address space, i.e. separated from program and data locations. Memory-mapped I/O (MMIO) and port I/O (also called port-mapped I/O or PMIO) are two complementary methods for performing input/output between the CPU and peripheral devices in an embedded application.[5] An external crystal was required to allow the 8048 to operate at a clock speed of 3-4 MHz, which resulted in the 8048 functioning at rate of approximately .33 -.5 MIPS[6]. The 27 I/O lines were bidirectional, with two groups of eight lines forming Ports 1 and 2.

[5]Another approach is to employ dedicated I/O processors or channels.

[6]MIPS is a figure of merit for a CPU and is a method of rating that is expressed in terms of the number of Millions of Instructions (executed by the CPU) Per Second.

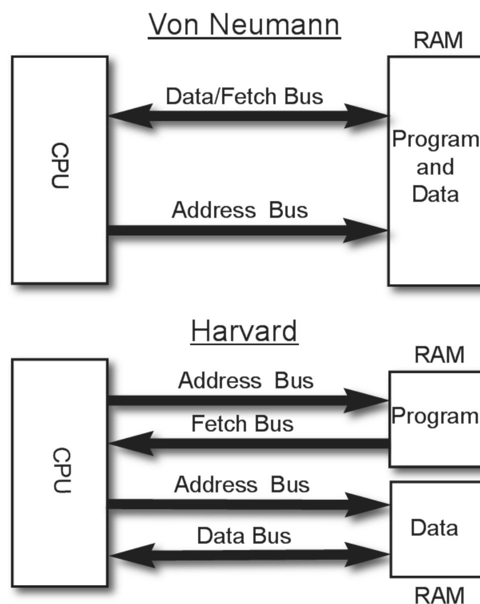


Figure 1.4: Comparison of Von Neumann and Harvard architectures.

Output data written to Ports 1 and 2 was “statically latched”, so that it remained unchanged, until overwritten. However, input data to these ports was not latched and had to be held by some external means until the data could be read by an 8048 input instruction. Maintaining statically latched I/O data continues to be one of the requirements of many of the modern microcontrollers.

An internal 8 bit bus was used to interconnect the accumulator, accumulator latch, a temporary 8 bit register, the flags register, the instruction register and decoder, the RAM address register, Ports 1 and 2, the Program Status Word (PSW), resident EPROM/ROM, lower program counter, as shown in Figure 1.10.

As shown in Figure 1.6, a microcontroller incorporates, within a single chip, all of the basic functionality embodied within a microprocessor and the various external (peripheral) devices required by a microprocessor to allow it to function as a “microcontroller”. However, there were many applications in which even this type of microcontroller was used in conjunction with external RAM, oscillator, UARTS/USARTS, A/Ds. D/As, PWMs, etc.

The 8048 included an arithmetic section consisting of an Arithmetic Logic Unit (ALU), accumulator, carry flag and instruction decoder. The arithmetic logic unit held 8-bit values obtained from either an accumulator latch or a temporary register, and produced a result defined by the instruction decoder. The stack was implemented using pairs of registers in the data memory area. The program counter was an independent counter, but the Program Counter stack used pairs of registers, also in data memory. Eight registers R0-R7, an eight level stack, an optional second register bank and the data store were all accessible by the ALU. The ALU could perform the operations shown Figure 1.7, and if any ALU operation resulted in an overflow of the most significant bit, the carry flag in the program status byte was set. The Program Status Word (PSW) was maintained in an 8 bit register, consisting of flip-flops, with the bit positions defined as shown in Figure 1.8.

Input/Output was performed using one of the 27 I/O lines provided. These lines were organized as three ports of 8 bits each and 3 test inputs that could be used to alter program

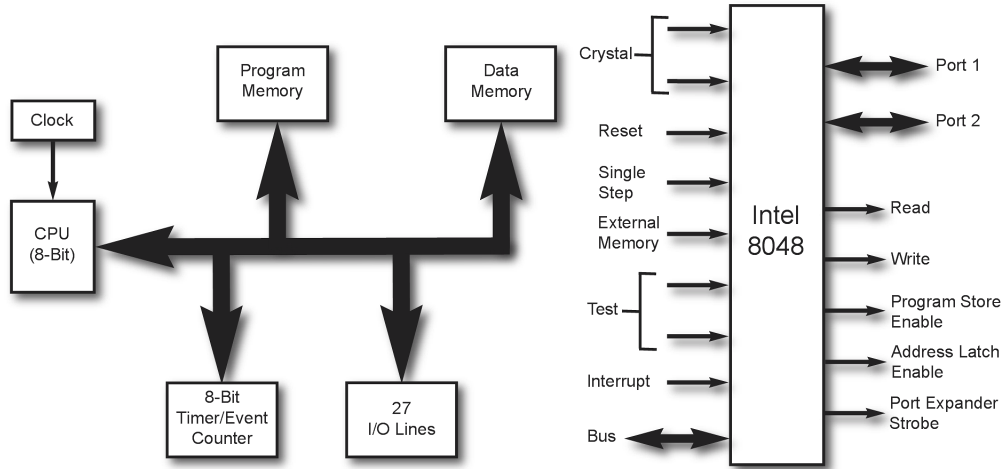


Figure 1.5: Block diagram of the Intel 8048.

execution, when tested by the applicable conditional jump instructions. Data written to a port was statically latched and remained so, until rewritten, but the input lines were non-latching and therefore inputs were required to remain static, until read by an input instruction.

Since an 8-bit address bus is limited to addressing a maximum of 256 bytes of memory, some other method must be employed to address additional RAM. Early microprocessors sometimes employed multiplexing of address lines to allow a larger memory space to be employed, e.g., 64K addressable by 16-bits, i.e., two bytes. In such applications, the lower byte of the address was placed on the address bus and latched in an external register. The upper byte was then placed on the address bus and latched in a second external register. The data byte held in the external memory location, and addressed by the two bytes, was then placed on the bus for retrieval by the microcontroller.

Another memory addressing technique employed by both microprocessors and microcontrollers is based on the concept of "paging" and utilizes a special register that holds a "page number" which serves as a pointer to a particular page, or segment of memory, of predefined size and location. Thus a CPU can for example, in principle at least, address an arbitrary number of pages of 256 bytes, or larger, by reading/writing a byte from/to a memory location pointed to by this special register. This type of memory structure is sometime referred to as "segmented" or "paged". With the advent of the Intel 8080, the address bus became 16-bits wide and therefore the page size became 64K.

However, paged/segmented memory structures imposed additional overhead on the CPU and increased the complexity of writing applications software. Microprocessors such as Motorola's 68000 had a memory structure referred to as linear/sequential/contiguous[7] that made it possible to directly address all available memory, which could be as large as 16 Mbytes.

Program memory, for the 8048, was 8-bits wide, addressable by the program counter and was available in 1024, 2048 and 4096 configurations. The memory was ROM-based and mask programmable, during manufacturing of the device[8]. Data memory was organized as 64, 128

[7]Linear, sequential contiguous memory architectures use a memory space in which memory is addressable physically and logically in the same manner.

[8]Versions of the 8048 were also available that had EPROM (Electrically Programmable, Read-Only Mem-

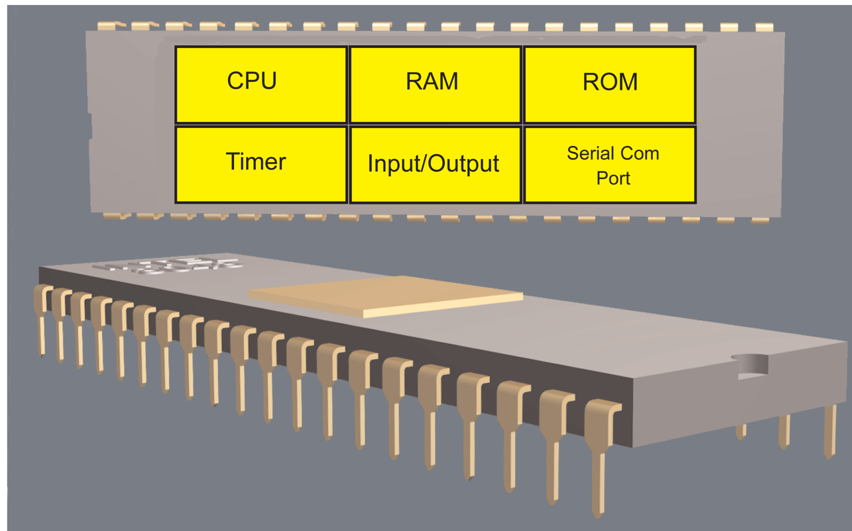


Figure 1.6: Architecture of a basic microcontroller, e.g.,the Intel 8048.

ADD (with or without carry)
AND, OR, Exclusive OR
Increment/Decrement
Rotate Right/Left
Swap Nibbles
Complement Bit
BCD Decimal Adjust

Figure 1.7: Accumulator operations

or 256 bytes and the stack size was restricted to 16 bytes. Even though, by current standards, the available memory was quite small, bank-switching was used to page memory which made it possible to address sufficient memory for many of the early applications for embedded systems. System memory consisted of so-called "Read/Write memory" (R/WM) and "Read-Only" memory (ROM). The latter was either static, which is fast but expensive, or dynamic which is slow(er), but cheaper. Read-Only memory was either erasable, or permanent, with erasable memory being EPROM (electrically programmable), EEPROM (electrically erasable),or ultimately Flash.

Permanent memory was masked ROM (created at the time of manufacture of the microcontroller) or programmable ROM, a one-time program capability that in some cases involved the burning of links, or fuses. Intel introduced the 8749, shown in Figure 1.9, a member of the 8048 family, that had an integral quartz window to allow the internal memory to be EPROM to be

ory)which allowed the device to be programmed/erased in the field to facilitate prototype development. The device could be erased by ultraviolet light applied to a quartz window on the 8047.

Bit Positions	PSW Bit Definitions
0	S_o
1	S1
2	S2
3	Unused
4	If 0, Bank 0; If 1 Bank 1
5	Flag 0, F(0-)
6	Auxillary Carry (AC)
7	Carry Flag (CY)

Figure 1.8: PSW bit positions.

erased by ultraviolet (UV) light.

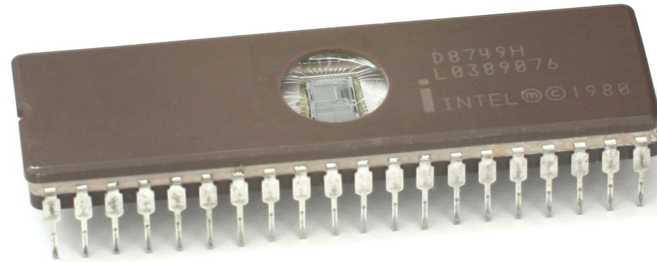


Figure 1.9: An ultraviolet (UV) erasable microcontroller.

Memory-mapped I/O, which is not to be confused with memory-mapped file I/O[9], uses the same address bus to address both memory and I/O devices. In such cases, the CPU instructions used to access the memory are also used for accessing external devices. In order to accommodate the I/O devices, areas of CPU's addressable space must be reserved for I/O. The reservation might be temporary in some systems to enable them to bank switch between I/O devices and permanent and/or RAM. Each I/O device monitored the address bus and responded to any access of device-assigned address space, by connecting the data bus to an appropriate device hardware register.

Port-mapped I/O uses a special class of CPU instructions specifically for performing I/O. This is generally the case for most Intel microprocessors, specifically the IN and OUT instructions which can read and write a single byte to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O. Thus inputs/outputs were accomplished by writing/reading to a predefined memory location. The 8048's internal architecture is shown in Figure 1.10.

The 8051 microprocessor was introduced by Intel in 1980 as a "system on a chip" and has subsequently become something of a worldwide standard in the fields of microcontrollers and

[9]Memory mapped file I/O refers to a technique of treating a portion of memory as if data is organized in a file format.

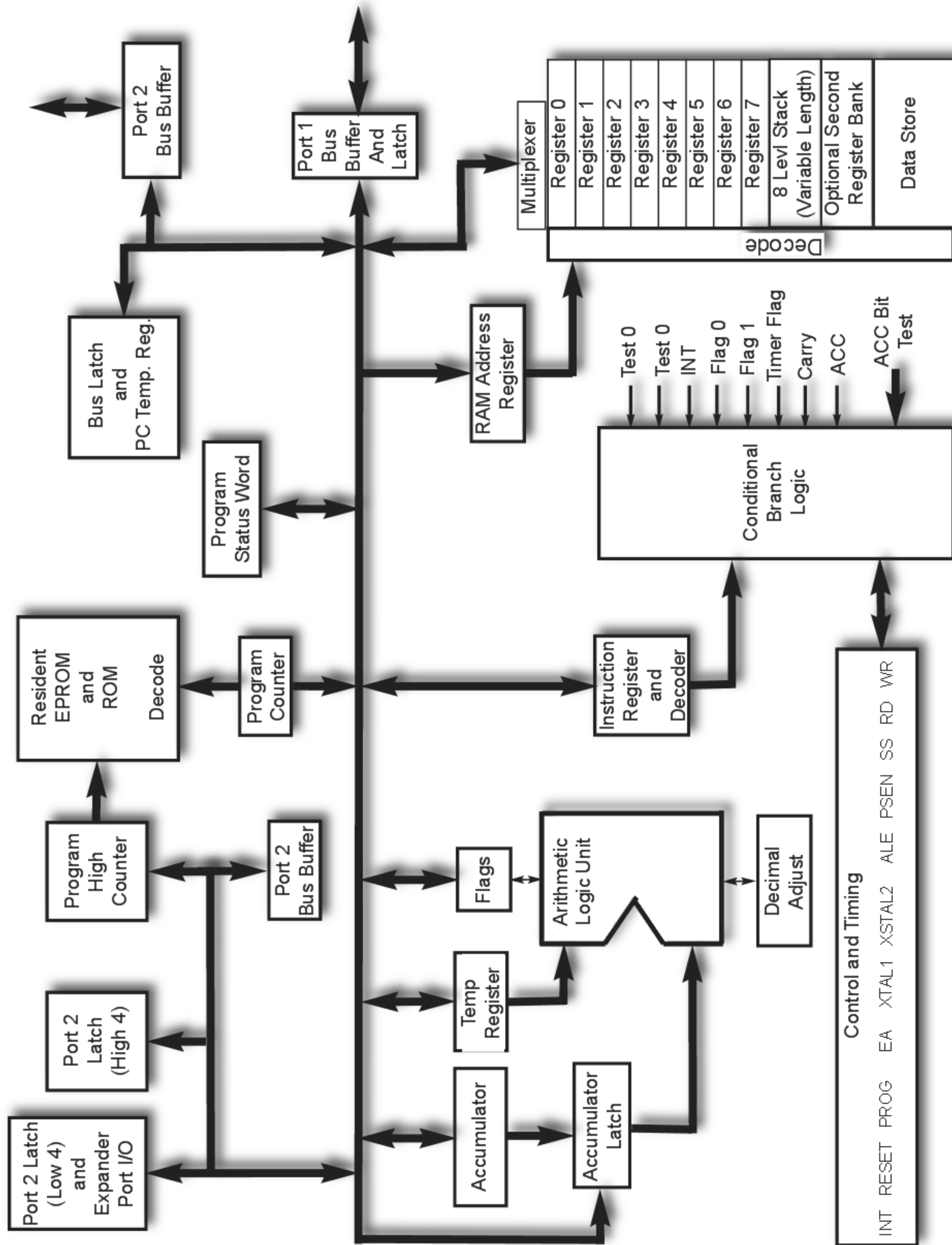


Figure 1.10: Intel 8048 internal architecture.

embedded systems. It was a refinement and extension of the basic design of the Intel 8048 and in its simplest configuration had the following:

- A Harvard memory architecture
- An ALU
- Seven on-chip registers
- A serial port (UART)
- A power saving mode
- Two 16 bit counter/timers
- Internal memory consisting of GP bit-addressable storage, register banks and special function registers
- Support for 64K of external memory (code)[10]
- Support for 64K external memory (data)
- Four 8-bit, bidirectional I/O ports[11]
- Two 10 bit-addressable locations[12]
- 128 bytes of internal RAM
- 4k bytes of internal ROM
- Multiple addressing modes - indirect/direct to memory, register direct via the accumulator
- 12 clock cycles per machine cycle
- Very efficient execution since most instructions required only one or two machine cycles.
- .5-1 MIPS performance at a clock speed of 12 MHz.
- An on-chip clock oscillator
- Six source, five vector interrupt handling
- 64K program memory address space
- 64K data memory address space
- Extensive Boolean handling capability

Intel ceased production of the 8051 in 2007, but a significant number of chip manufactures continue to offer 8051-like architectures, many of which have been very substantially enhanced and extended compared to the original design, e.g., Atmel, Cypress Semiconductor, Infineon Technologies, Maxim (Dallas Semiconductor), NXP, ST Microelectronics, Silicon Laboratories, Texas Instruments and Winbond. Although the original 8051 was based on NMOS technology, in recent years CMOS versions of the 8051, or similar architecture, have become widely available.

In addition to supporting memory-mapped I/O, the 8051's registers were also memory-mapped and the stack resided in RAM which was internal to the 8051. The 8051's ability to access individual bits made it possible to set, clear AND, OR, etc., individual bits utilizing a single 8051 instruction. Register banks were contained in the lowest 32 bytes of the 8051's internal memory. Eight registers were supported, viz, R0-R7, inclusive, and their default locations were at addresses 0x00-0x07. Register banks could also be used to provide efficient context switching

[10]The 8051 utilizes a separate 64K for data and code, respectively.

[11]I/O ports in the 8051 are "memory-mapped", i.e. to read/write from/to an I/O port the program being executed must read/write to the corresponding memory location in the same manner that a program would normally read/write to any memory location.

[12]128 of these are at addresses 0x20-0x2F with the remaining 73 being located in special function registers

and the active register bank was selected by bits in the Program Status Word (PSW). At the top of the internal RAM there were 21 special registers located at addresses $0x80 - 0xFF$. Some of these registers were bit- and byte-addressable, depending upon the instruction addressing the register.

As designers became increasingly more comfortable with microcontrollers they began to take on more and more complex embedded system applications. This resulted in the need for a much wider variety of peripheral devices. PWMs, additional UARTs, A/D converters and D/A converters were some of the first modules to be available "on-chip". The demand continued to grow for more memory, CPU functionality, faster clock speeds, better interrupt handling, support for more levels of interrupt, etc., and with the introduction of OpAmps "on-chip' and typically interoperable ', the demand for analog devices increased, as well. On-chip real estate has always been an extremely valuable asset and while additional digital functionality was also desirable, the need for more analog support was greater.

Thus a compromise was required and resulted in so-called "mixed-signal"[13] techniques being introduced into embedded system applications space. As a result, there are now a large number of manufacturers of microcontrollers utilizing a variety of cores, many of which are 8051 derivatives, and various combinations/permutations of analog and digital "peripherals" that are provided by the manufacturer, on-chip.

Some of the representative types that are currently available are listed below with a brief description:

68HC11 (Motorola) - CISC, 8-bit, two 8-bit accumulators, two sixteen-bit index registers, a condition code register, 16-bit stack pointer, 3-5 ports, 768 bytes internal memory, max of 64k external RAM, 8051 8-bit ALU and registers, UART, 16-bit counter/timers, four byte (bidirectional) I/O port, 4k on-chip ROM, 128-256 bytes of memory for data, 4 Kbytes of memory for the program, 16-bit address bus, 8-bit data bus, timers, on-chip oscillators, bootloader code in ROM, power saving modes, in-circuit debugging facilities, i2C/SPI//USB interfaces, reset timers with brownout detection, self-programming Flash ROM (program memory), analog comparators, PWM generators, support for LIN/CAN busses, A/D and D/A converters, non-volatile memory (EEPROM) for data, etc.

at91SAM3 (ATMEL at91SAM series) - ARM Cortex-M3 revision 2.0, core, max clock rate 64 MHz, Memory Protection Unit (MPU), Thumb-2 instruction set, 64 to 256 Kbytes embedded Flash, 128-bit wide access memory accelerator (single plane), 16 to 48 Kbytes embedded SRAM, 16 Kbytes ROM with embedded bootloader routines (UART, USB) and IAP routines, 8-bit static memory controller (SMC): SRAM/PSRAM/NOR and NAND Flash support, embedded voltage regulator for single supply operation, Power-on-Reset (POR), brown-out detector (BOD) and watchdog, quartz or ceramic resonator oscillators: 3 to 20 MHz main power with Failure Detection and optional low power 32.768 KHz for RTC or device clock, high precision 8/12 MHz factory trimmed internal RC oscillator with 4 MHz default frequency for device startup, (in-application trimming access for frequency adjustment),slow clock internal RC oscillator as permanent low-power mode device clock, two PLLs up to 130 MHz for device clock and for USB, temperature sensor, 22 peripheral DMA (PDC) channels, low power modes (sleep and backup modes, down to $3\mu A$ in backup mode), ultra low power RTC, USB 2.0 (12 Mbps, 2668 byte FIFO, up to 8 bidirectional endpoints, on-chip transceiver), 2 USARTs with ISO7816 (IrDA, RS-485, SPI, Manchester and modem Mode), two 2-wire UARTs, 2 two-wire I2C compatible interfaces SPI (1 Serial Synchronous Controller (I2S), 1 high speed multimedia card interface (SDIO/SD Card/MMC)), 6

[13]Mixed-signal refers to an environment in which both analog and digital signals are present and in many cases being processed individually by analog and digital modules that are interoperable.

three-channel 16-bit timers/counters with capture/waveform/compare and PWM mode, quadrature decoder logic and 2-bit Gray up/down counter for driving a stepper motor, 4-channel 16-bit PWM with complementary output/fault input/12-bit dead time generator counter for motor control, 32-bit real-time timer and RTC with calendar and alarm features, 15-channel 1Msps ADC with differential input mode and programmable gain stage, one 2-channel 12-bit 1Msps DAC, one analog comparator with flexible input selection/window mode and selectable input hysteresis, 32-bit Cyclic Redundancy Check Calculation Unit (CRCCU), 79 I/O lines with external interrupt capability (edge or level sensitivity)/debouncing/glitch filtering and on-die series resistor termination, three 32-bit parallel input/output controllers, and peripheral DMA assisted parallel capture mode.

ST92F124xx (STMicroelectronics ST92F Family) Single Voltage Flash 256 Kbytes (max), 8Kbytes RAM (max), 1K byte E^3(Emulated EEPROM), In-Application Programming (IAP), 224 general purpose registers (register file) available as RAM, accumulators or index pointers, Clock, reset and supply management, register-oriented 8/16 bit CORE with RUN, WFI, SLOW, HALT and STOP modes, 0-24 MHz operation (Int. Clock), 4.5-5.5 V range, PLL Clock Generator (3-5 MHz crystal), minimum instruction time: 83 ns (24 MHz int. clock), 80 I/O pins, 4 external fast interrupts + 1 NMI, 16 pins programmable as wake-up or additional external interrupt with multi-level interrupt handler, DMA controller for reduced processor overhead, 16-bit timer with 8-bit prescaler, and watchdog timer (activated by software or by hardware), 16-bit standard timer that can be used to generate a time base that is independent of the PLL clock generator, two 16-bit independent Extended Function Timers (EFTs) with prescaler, two input captures and two output compares, two 16-bit multifunction timers, with prescaler, two input captures and two output compares, Serial Peripheral Interface (SPI) with selectable master/slave mode, one multiprotocol Serial Communications Interface with asynchronous and synchronous capabilities, one asynchronous Serial Communications Interface with 13-bit LIN Synch Break generation capability, J1850 Byte Level Protocol Decoder (JBLPD), two full IC multiple master/slave interfaces supporting the access bus, two CAN 2.0B active interfaces, 10-bit A/D converter (low current coupling).

ATmega8 (Atmel AVR 8-bit Family) - Low-power AVR 8-bit Microcontroller, RISC Architecture, 130 instructions (most are single-clock cycle execution), 32 x 8 general purpose registers, fully static operation to 16 MIPS throughput at 16 MHz, on-chip 2-cycle multiplier, High 8K-32K Bytes of In-System Self-programmable Flash program memory, 512 Bytes EEPROM, 1K Byte SRAM, 10,000 Flash/100,000 EEPROM write/erase cycles(20 years data retention, optional boot code section with independent lock bits, in-system programming by on-chip boot program, true Read-While-Write operation, programming lock for software security, two 8-bit timer/counters with separate prescaler and one compare mode, one 16-bit timer/counter with separate prescaler and compare/capture mode, a real time counter with a separate oscillator, three PWM Channels, 6-8 channel ADC with 10-bit Accuracy, byte-oriented two-wire Serial Interface, programmable Serial USART, master/Slave SPI Serial Interface, programmable watchdog timer with separate on-chip oscillator, analog comparator, power-on Reset and programmable brown-out detection, calibrated RC Oscillator, external and internal interrupt sources, five sleep modes (Idle, ADC noise reduction, power-save, power-down, and standby), 23 programmable I/O lines, operating voltages: $2.7 - 5.5V$, clock speeds: $0 - 16MHz$, current drain $(4Mhz, 3V, 25C)$ - active: $3.6mA$/idle: $1.0mA$/power-down: $0.5\mu A$

80C51 (Atmel) - 8051 Core Architecture, 256 Bytes of RAM, 1K Bytes of XRAM, 32K Bytes of Flash, Data Retention: 10 years at 85C, Erase/Write cycle: 100K, Boot code section with independent lock bits, 2K Bytes Flash for Bootloader, in-system programming by Boot program, CAN, UART and IAP Capability, 2K Bytes of EEPROM, Erase/Write cycle: 100K, 14-sources 4-level interrupts, three 16-bit timers/counters, full duplex UART, maximum crystal frequency

of 40 MHz (X2 mode)/20 MHz (CPU Core, 20 MHz), five ports: 32 + 2 digital I/O Lines, five-channel 16-bit PCA with: PWM (8-bit)/high-speed output /timer and edge capture, double data pointer, 21-bit watchdog timer (7 Programmable Bits),10-bit analog to digital converter (ADC) with 8 multiplexed inputs, on-chip emulation logic (enhanced hook system),power saving modes: idle and power-down. Full CAN controller (CAN Rev2.0A and 2.0B), 15 independent message objects: each message object programmable on transmission or reception, individual tag and mask filters up to 29-bit identifier/channel, 8-byte Cyclic Data Register (FIFO)/message object, 16-bit status and control register/message object, 16-bit time-stamping register/message object, CAN specification 2.0 Part A or 2.0 Part B programmable for each message object, access to message object control and data registers via SFR, programmable reception buffer length up to 15 message objects, priority management of reception of hits on several message objects at the same time, priority management for transmission message object, overrun interrupt, support for time-triggered communication autobaud and listening mode, programmable automatic reply mode, 1-Mbit/s maximum transfer rate at 8 MHz crystal frequency in X2 mode, readable error counters, programmable link to timer for time Stamping and network synchronization, independent baud rate prescaler, data/remote error and overload frame handling,

PIC (MicroChip PIC 18F Family) - 8 and 16 bit, Harvard architecture, one or more accumulators, small instruction set, general purpose I/O pins, 8/16/32 bit timers, internal EEPROM, USART/UART, CAN/USB/Ethernet support, internal clock oscillators, hardware stack, capture/compare/PWM modules, A/D converter, etc., multiple power managed modes (Run: CPU on, peripherals on, Idle: CPU off, peripherals on, Sleep: CPU off, peripherals off, multiple power consumption modes (PRI_RUN: 150μA, 1MHz, 2V, PRI_IDLE: 37μA, 1MHz, 2V SEC_RUN: 14μA, 32kHz, 2V SEC_IDLE: 5.8μA, 32kHz, 2V RC_RUN: 110μA, 1 MHz, 2V RC_IDLE: 52μA, 1 MHz, 2V sleep: 0.1μA, 1 MHz, 2V) timer1 oscillator: 1.1μA, 32kHz, 2V watchdog timer: 2.1μA two-Speed Oscillator Start-up, four Crystal modes(LP, XT, HS: up to 25 MHz) HSPLL: 4-10 MHz (16-40 MHz internal), two External RC modes, up to 4 MHz, two External Clock modes, up to 40 MHz, internal oscillator block (8 user-selectable frequencies: 31 KHz, 125 KHz, 250 KHz, 500 KHz, 1 MHz, 2 MHz, 4 MHz, 8 MHz), 125 KHz to 8 MHz calibrated to R1%, two modes select one or two I/O pins, OSCTUNE allows user to shift frequency, secondary oscillator using Timer1 @ 32 KHz, fail-safe clock monitor (allows safe shutdown if peripheral clock stops), high current sink/source 25 mA/25 mA, three external interrupts, enhanced Capture/Compare/PWM (ECCP) module (One, two or four PWM outputs, selectable polarity, programmable dead time, auto-Shutdown and Auto-Restart, capture is 16-bit, max resolution 6.25 ns (TCY/16), compare is 16-bit, max resolution 100 ns (TCY)), compatible 10-bit, 13-channel Analog-to-Digital Converter module (A/D) with programmable acquisition time, enhanced USART module: supports RS-485, RS-232 and LIN 1.2 auto-wake-up on start bit, auto-Baud Detect, 100,000 erase/write cycle Enhanced Flash program memory typical 1,000,000 erase/write cycle Data EEPROM memory typical Flash/Data EEPROM Retention: ¿ 40 years self-programmable under software control priority levels for interrupts, 8 x 8 Single-Cycle Hardware Multiplier, extended Watchdog Timer (WDT)(Programmable period from 41 ms to 131s with 2% stability), single-supply 5V In-Circuit Serial Programming, (ICSP) via two pins, In-Circuit Debug (ICD) via two pins, wide operating voltage range: 2.0V to 5.5V.

MSP430(Texas Instruments MSP 430F Family) RISC[14] set computer instruction set, Von Neumann architecture, 27 instructions, maximum of 25 MIPS, operating voltage: 1.8V to 3.6V, internal voltage regulator, constant generator, program storage: 1KB - 55KB, SRAM: 128 -5120

[14]Computer are typically categorized as either Complex Instruction Set Computers (CISC), or as Reduced Instruction Set Computers. The basic concept is that RISC instructions require fewer machine cycles per instruction than CISC instructions. In RISC machines the instructions are typically of fixed length, each is responsible for a simple operation, general purpose registers are used for data operations not memory, data is moved via load and store instructions, etc.

Bytes, I/O: 14-48 pins, multi-channel DMA, 8-16 channel ADC, watchdog, real time clock (RTC), 16 bit timers, brownout circuit, 12 bit DAC, Flash is bit/byte and word addressable, maximum of twelve 8-bit bidirectional ports (Ports 1 and 2 have interrupt capability), individually configurable pull-up/pull-down resistors, supporting static/2-mux/3-mux and 4-mux LCDs. integrated charge pump for contrast control, single supply OpAmps, rail-to-rail operation, programmable settling times, OpAmp configuration includes: unity gain mode/comparator mode/inverting PGA/non-inverting PGA, differential and instrumentation modes, hardware multiplier supports 8/16 bit x 8/16 bit, signed and unsigned with optional "multiply and accumulate", DMA accessible, maximum of seven 16-bit sigma delta A/D converters, each has up to 8 fully differential multiplexed pinouts including a built-in temperature sensor, supply voltage supervisor, asynchronous 16-bit timers with up to 7 capture/compare registers, PWM outputs, a USART, SPI, LIN, IrDA and I2C support, programmable baud rates, USB 2.0 support at 12 Mbps, USB suspend/resume and remote wakeup,

PSoC1 (Cypress CY8C29466) - programmable system on a chip, Harvard architecture, M8C Processor Speeds up to 12 MHz, two 8x8 Multiply with 32-Bit accumulate, 4.75V to 5.25V operating voltage, 14-Bit ADCs, 9-Bit DACs, programmable gain amplifiers, programmable filters, programmable comparators, 8- to 32-Bit timers/counters, PWMs, CRC/PRS, four full-duplex or eight half-duplex UARTs, multiple SPI masters/slaves (connectable to all GPIO pins), Internal ± 4and PLL, Optional External Oscillator, up to 24 MHz, Internal Low Speed, Low Power Oscillator for Watchdog and Sleep Functionality, 24 MHz Oscillator high accuracy 24 MHz clock, optional 32.768 KHz crystal and PLL support, external oscillator support to 24 MHz, watchdog and sleep functionality, flexible on-chip memory, 32K Bytes Flash program storage, 100k erase/write cycles 2K Bytes SRAM data storage, In-System Serial Programming (ISSP), partial Flash updates, flexible protection modes, EEPROM emulation in Flash, programmable pin configurations: 25 mA sink, 10 mA drive on all GPIO Pull Up, Pull Down, High Z, Strong, or Open Drain Drive Modes on All GPIO, Up to 12 Analog Inputs on GPIO[1], Four 30 mA Analog Outputs on GPIO Configurable Interrupt on all GPIO, I2C master/slave or multi-master operation to 400 KHz, watchdog/sleep timers, user-configurable low voltage detection, integrated supervisory circuit and precision voltage reference.

PSoC3 (Cypress CY8C34 Family) - single cycle 8051 CPU core, DC to 48 MHz operation, multiply/divide instructions, Flash program memory to 64 KB, 100,000 write cycles, 20 years retention, multiple security features, max of 8 KB Flash ECC or configuration storage, max of 8 KB SRAM, max of Up to 2 KB EEPROM (1M cycles, 20 years retention), 24 channel DMA with multi-layer AHB bus access, programmable chained descriptors and priorities, high bandwidth 32-bit transfer support, operating voltage range from 0.5V to 5.5V, high efficiency boost regulator (0.5V input to 1.8V-5.0V output), current drain $330\mu A$ at 1 MHz, $1.2mA$ at $6MHz$, $5.6mA$ at $40MHz$, $200nA$ hibernate mode with RAM retention and LVD, $1\mu A$ sleep mode with real time clock and low voltage reset, 28 to 72 I/O channels (62 GPIO, 8 SIO, 2 USBIO[1]), any GPIO to any digital or analog peripheral routability, LCD direct drive from any GPIO (max of 46x16 segments), 1.2V to 5.5V I/O interface voltages (max of 4 domains), maskable independent IRQ on any pin or port, Schmitt trigger TTL inputs, all GPIO configurable (open drain high/low, pull up/down, High-Z, or strong output), configurable GPIO pin state at power on reset (POR), 25 mA sink on SIO, 16 to 24 programmable PLD-based Universal Digital Blocks, full CAN 2.0b 16 RX, 8 TX buffers, USB 2.0 (12 Mbps) using an internal oscillator, max of four 16-bit configurable timer, counter, and PWM blocks, 8, 16, 24, and 32-bit timers, counters, and PWMs, SPI, UART, I2C, Cyclic Redundancy Check (CRC), Pseudo Random Sequence (PRS) generator, LIN Bus 2.0, Quadrature decoder, configurable Delta-Sigma ADC with 12-bit resolution (programmable gain stage: x0.25 to x16, 12-bit mode, 192 ksps, 70 dB SNR, 1 bit INL/DNL), two 8-bit 8 Msps IDACs or 1 Msps VDACs, four comparators with 75 ns response time, two uncommitted OpAmps with

25 mA drive capability, two configurable multifunction analog blocks. (configurable as PGA, TIA, Mixer, and Sample/Hold), JTAG (4 wire), Serial Wire Debug (SWD) (2 wire), Single Wire Viewer (SWV) interfaces, Bootloader programming supportable through I2C, SPI, UART, USB, and other interfaces, precision, programmable clocking (1 to 48 MHz (±1% with PLL), 4 to 33 MHz crystal oscillator for crystal PPM accuracy, PLL clock generation to 48 MHz, 32.768 KHz watch crystal oscillator, Low power internal oscillator at 1 KHz and 100 KHz.

PSoC5 (Cypress CY8C53 Family) - 32-bit ARM Cortex-M3 CPU core, DC to 80 MHz operation, Flash program memory (max 256 KB, 100,000 write cycles, 20 year retention), multiple security features, 64 KB SRAM (max), 2 KB EEPROM (1 million cycles, 20 years retention), 24 channel of DMA with multi-layered AHB bus access, programmable chained descriptors and priorities, high bandwidth 32-bit transfer support, operating voltage ranges: 0.5V to 5.5V, high efficiency boost regulator (0.5V input to 1.8V to 5.0V output, current drain of 2 mA at 6 MHz), 300 nA hibernate mode with RAM retention and LVD, $2\mu A$ sleep mode with real time clock and low voltage reset, 28 to 72 I/O channels (62 GPIO, 8 SIO, 2 USBIO), any GPIO to any digital or analog peripheral routability, LCD direct drive from any GPIO (max of 46x16 segments), 1.2V to 5.5V I/O interface voltages (max of 4 domains), maskable independent IRQ on any pin or port, Schmitt trigger TTL inputs, all GPIO configurable (open drain high/low, pull up/down, High-Z, or strong output), configurable GPIO pin state at power on reset (POR), 25 mA sink on SIO, 20 to 24 programmable PLD based Universal Digital Blocks, full CAN 2.0b 16 RX, 8 TX buffers, full-USB 2.0 (12 Mbps using internal oscillator), max of four 16-bit configurable timer, counter, and PWM blocks, 8, 16, 24, and 32-bit timers, counters, and PWMs SPI, UART, I2C, Cyclic Redundancy Check (CRC), Pseudo Random Sequence (PRS) generator, LIN Bus 2.0, Quadrature decoder, SAR ADC (12-bit at 1 Msps), four 8-bit 8 Msps IDACs or 1 Msps VDACs, four comparators with 75 ns response time, four uncommitted OpAmps with 25 mA drive capability, four configurable multifunction analog blocks (PGA, TIA. Mixer and Sample and hold), JTAG (4 wire), Serial Wire Debug (SWD) (2 wire), Single Wire Viewer (SWV), and TRACEPORT interfaces, Cortex-M3 Flash Patch and Breakpoint (FPB) block, Cortex-M3 Embedded Trace Macrocell (ETM) for generating an instruction trace stream. Cortex-M3 Data Watchpoint and Trace (DWT) for generating data trace information, Cortex-M3 Instrumentation Trace Macrocell (ITM) for printf-style debugging, DWT, ETM, and ITM blocks that can communicate with off-chip debug and trace systems via the SWV or TRACEPORT, Bootloader programming supportable through I2C, SPI, UART, USB, and other interfaces, Precision, programmable clocking from 1 to 72 MHz with PLL, 4 to 33 MHz crystal oscillator for crystal PPM accuracy, PLL clock generation up to 80 MHz, 32.768 KHz watch crystal oscillator support, low power internal oscillator at 1 KHz and 100 KHz.

1.3 Embedded System Applications

Embedded systems can be found in an ever increasing number of applications including: televisions, cable boxes, satellite boxes, cable modems, routers, printers, microwave ovens, surround sound systems, computer monitors, digital cameras, zoom lenses, cars and trucks (some vehicles have 100+ such systems), stereos, dishwashers, dryers, washing machines, cell phones, digital multimeters, calculators, air conditioners, mp3 players, heaters, flight-control systems (fly-by-wire), running shoes, tennis rackets, traffic lights, elevators, telecommunications systems, medical equipment, airplanes, automotive cruise controls, ignition systems, personal digital assistants, pleasure boats, motorcycles, children's toys, oscilloscopes, ships, industrial and process control applications, railway systems, laboratory equipment, personal computers, data collection/logging equipment, numerical processing applications,"smart" shoes, robotics, fire/security alarms, biometric systems, proximity detectors, inertial guidance systems, GPS devices, UAVs, etc. The

major markets for embedded systems include automotive, medical, avionic, communications, industrial and consumer electronics.

For example, increasing numbers of automobile manufacturers produce products that utilize embedded systems that control their vehicle's major functions, such as powertrain management, air conditioning, (heating/cooling systems), seat positioning mechanisms, fuel systems, braking mechanisms, dashboard instrumentation, GPS systems, etc. Automakers must also continue to respond to steadily growing requirements for advanced safety, environmental protection and driver convenience, thus increasing the number of microelectronics components, in a vehicle, all of which continue to require more and more "lines of code".

In the last two decades, the total number of lines of code employed by automobile manufacturers has reportedly grown from approximately one million lines to close to one hundred million lines of proprietary and third party code. Thus the ease with which new code can be developed and reusability in future designs becomes of paramount importance. This is particularly true as microcontrollers evolve with increasingly more complex architectures in an attempt to meet market demands.

The need to continually:

- reduce the time to market for new designs,
- introduce less expensive microcontrollers with ever increasing capability and in some cases more specialization,
- support ever increasing application complexity,

and,

- support lower and lower power consumption,

has, in turn, increased the demand for more and more generic and specialized microcontrollers and substantially advanced the state of the available microcontroller technology and associated peripherals.

Automotive Electronics - vehicle manufacturers continue to move aggressively in implementing more and more embedded system technology into new vehicles to increase their competitive strengths in meeting the new challenges of their competitors and public demand for more efficient, reliable and feature-rich transportation.

Currently the number of microprocessors/microcontrollers in automobiles ranges from 10, to more than 100, with current estimates suggesting that as much as 40% of the value of some automobiles is invested in the electronics systems and networking. Some modern vehicles employ three, or more, network protocols, e.g., LIN (10 kbits/sec), CAN (1 Mbits/sec) and FlexRay[15] (10 Mbits/sec) to address the wide range of realtime responses needed in contemporary vehicles.

High speed networking, utilizing FlexRay and high-speed CAN, is required to handle fuel ignition and exhaust systems, spark/valve timing, fuel injection systems, anti-lock braking systems, cruise control, air bags, active suspension, steer-by-wire, brake-by-wire, and other "x-by-wire" systems. Low speed networking, utilizing LIN and low-speed CAN are employed to handle less demanding real time requirements, such as the instrument panel (dash board), air conditioning, windshield wipers, power windows, mirror adjustments, seat controls, alarm systems, door locks,

[15]FlexRay is an open, scalable network protocol created by a consortium consisting of Philips Semiconductor, BMW, DaimlerChrysler, Motorola, BMW, Ford Motor Company, General Motors Corporation and Robert Bosch GMBH specifically for automotive applications. It supports both synchronous and asynchronous data transfers and is capable of operating in either a single channel or double channel mode, if redundancy is required.

head lights, internal lighting systems, stop/tail/fog lights and high/low beams, seat temperature controllers, etc.

Avionic Electronics - Private, commercial and military avionic systems make extensive use of embedded systems for fly-by-wire systems, GPS-based and other navigational systems such as inertial navigation systems. Heads-up displays, power plant monitoring and control, instrument displays, communication systems, transponders, instrument panels, transponders, communications equipment, internal/external lighting systems, offensive and defensive weapon systems, etc., are also increasingly controlled and/or monitored by embedded systems.

Consumer Electronics - Since the advent of the microprocessor, consumer electronics have continuously taken more and more advantage of semiconductor technology and most particularly of microcontrollers. Modern homes make extensive use of embedded systems in the form of security systems, lighting control systems, stereo systems, telecommunications systems, cable TV and Internet systems, personal computers, MP3 players, etc.

Communications Electronics - cell phones, telephone switches, GPS, routers, microwave and satellite systems, etc., make extensive use of embedded systems.

Industrial Electronics - process control systems, numerically controlled milling and drilling machines, robotics, automated inspection systems, etc., are heavily dependent on embedded systems particularly for high volume, close tolerance manufacturing processes and systems.

Medical Electronics - blood pressure, young child/adult heart rate, fetal heart rate, pulse oximetry, blood glucose, electrocardiogram, ventilation/respiration, electronic stethoscopes, vital signs and anesthesia monitors are all embedded systems that are used in homes as well as, clinics, doctors offices and hospitals. Imaging systems, e.g., acoustic (sonograms), X-Ray, CT(X-Ray Computed Tomography), MRI(Magnetic Resonance Imaging), SPECT(Single photon emission computed tomography), and PET(Positron Emission Tomography), powered patient beds, patient monitoring systems, operating room systems, robotic surgery systems are also important embedded system application spaces.

Each of these relies on one, or more, embedded systems to gather input data from devices called “sensors”, and/or other data sources. Based on the information gathered, they then engage in numerical/logic processing of the input data, subject to certain predefined constraints and/or operating modes (states), make decisions based on the input data and subsequently provide outputs to various types of devices, such as other computer systems, display devices, actuators, motors, speakers, data transport channels, etc.

1.4 Embedded System Controlling

1.4.1 Types of Embedded System

Embedded systems are capable of functioning in a number of different modes, e.g.,

1. **Event-Driven Mode** (**EDM**) - perhaps the most common type of embedded system which is constrained to responding to previously defined events and providing pre-defined responses. The system waits for an event to occur in the form of a key depression, a parameter meeting some threshold level and thus representing an event, or other “triggering” events,
2. **Continuous Time Mode** (**CTM**) - such systems are continuously monitoring input channels and reacting to various input conditions,

3. **Discrete Time Mode (DTM)**- these systems "wake-up" at predetermined intervals, sample input data, carry out the appropriate responses and then go back to "sleep",

or some permutation thereof. For example a system may be required to "wake-up, respond to some set of input conditions on an event-driven basis and the go back to "sleep". Some systems employ "watch dog" functions that in the absence of the system responding within a pre-determined period of time, automatically reset themselves as a way of avoiding the system becoming "locked-up" because of some anomalous situation, or malfunction, and subsequently failing to function as described previously.

While some embedded systems are primarily involved in control functions, and to a lesser degree data processing, others are predominantly engaged in data processing/collection and some control functions. In such cases, the former are usually described as state machines that move from state-to-state as a result of certain events or input data conditions/values. In such cases, the embedded system remains in a given state until conditions arise in terms of events, or input data, that meets the criteria for a state transition. Resetting/setting such systems causes the state machine to enter a predetermined"home" state.

Whether functioning as:

- a controller designed to maintain certain parameters, or operating conditions, of a system, or process, within pre-defined ranges or contexts,
- part of a network of embedded systems engaged in making decisions, monitoring activity and/or exchanging information regarding the various systems, or processes, to be monitored or controlled and their respective states,
- an application-specific embedded system for image/video processing, graphics, multimedia processing,
- an embedded system for demanding computational applications and interfacing applications,
- a data logging system for applications such as remote sensing systems,

or,

- a specialized/custom digital communications processing system, such as part of a data link,

each consists of a CPU, memory, registers, address/data busses and various peripheral devices such as analog-to-digital converters, pulse-width modulators, digital-to-analog converters, various types of signal conditioners such as filters, comparators, etc.

Some embedded systems employ real time operating systems, while others are merely subsystems in a real time operating system environment. In the latter case, failure of one, or more, of the embedded subsystems might allow some portion of the total system to continue to function. In systems for which the embedded system has primary control, any failure could prove catastrophic, and therefore requires much more attention to failure modes and how best to address them by employing, e.g., fail-safe modes. However, real time operating systems add complexity, cost and processing overhead which, for some applications, is undesirable and can significantly degrade the system's performance.

1.4.2 Open Loop, Closed Loop and Feedback

Embedded systems may be implemented as open, or closed, loop systems. An "open loop" system, sometimes referred to as a "feedforward" system, as shown in Figure 1.11 acquires input information and produces appropriate outputs based on the acquired inputs, without any ability

Figure 1.11: Schematic view of an open loop system.

to determine whether or not, ultimately, the correct action, or actions have taken place[16]. Furthermore, such as system assumes that the input data is always correct and that there are no disturbances, or anomalies, to take into account. An open loop embedded system gathers information, reacts to the input parameter values in a predefined way and produce the appropriate output signals/commands, e.g., a thermostat senses temperature, (TempSensed), compares the temperature to a preset value, (TempUpperLimit), and if:

$$TempUpperLimit < TempSensed, \tag{1.1}$$

closes some switch contacts to turn on a fan. However, this simple system does not know whether the fan is actually operating, or if it is operating at a speed sufficient to return the temperature to an acceptable value, within a required period of time. Furthermore, if the (TempSense) exceeds (TempUpperLimit) the system will continue to attempt to providing cooling, but make no attempt to take further corrective action. This system is also representative of the open loop system shown in Figure 1.11.

Should the fan fail to be activated at the proper speed, the controller, in this example, would not initiate any further action, i.e., since there is no indication returned to the controller that cooling is, or is not, actually taking place. This type of open loop system is referred to as a "bang-bang" system since it does not provide proportional control of the device that it controls, i.e., the fan is either operating at a constant speed (RPMs), or is inactive. It would of course be possible to program the controller to monitor the input temperature as an explicit function of time so that if it, for example, found that the temperature was not changing it could take other actions, e.g., sounding of an alarm.

A similar example of an open loop system is shown in Figure 1.12, in which a motor is controlled by a sensor and an embedded system consisting of a microcontroller and a pulse width modulator (PWM)[17] driving an amplifier with sufficient output voltage and current to drive motor. In this case the speed of the motor is determined by the embedded system which controls the duty-cycle[18] of the pulse train produced by the PWM. Therefore the controller is able to provide proportional control of the fan by controlling the average amount of power provided to it.

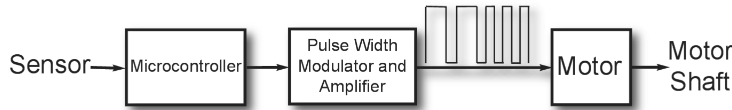


Figure 1.12: Embedded System motor controller.
Motor

[16]In some situations, no action may constitute the correct action.

[17]A Pulse Width Modular (PWM) is a device capable of producing pulses of variable width and frequency, in this case under the control of a microcontroller, that in the present example allows the speed of the motor to be varied over a wide range (Cf Section XX of Chapter ZZ)

[18]Duty cycle is defined as the ratio of time on to time off over some predefined period of time.

Some motors have integral tachometers and/or Hall effect sensors that can be used to produce an analog signal that can be returned directly to the summing junction to produce an error signal to be processed by the controller and to confirm that the motor is running at the appropriate speed. In Figure 1.13 the system returns a signal, e.g., an analog voltage/current, or digital data, that reflects the output state of the system, to the input for comparison.

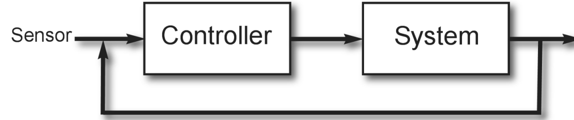


Figure 1.13: Schematic view of a closed loop system with direct feedback.

In some cases, the embedded system is provided with input parameters that are presumed to represent the state of the a system and the controller compares these values that characterize the current state of the system with predefined state conditions and makes decisions regarding what steps must be taken, if any, to bring the system into compliance with these conditions. Other embedded systems utilize a sensor, or sensors, to determine if the parameter values are consistent with the desired state of a system and additional sensors that represent the actual state of the process or system. In both cases, the input and output sensor signals are provided to a "summing junction", or equivalent, to produce an error signal as shown in Figure 1.14.

Note that in the former case, one sensor is used to establish a comparison between "setpoint", i.e, "desired input parameter value" and another sensor is employed to determine the output state, or "actual state", so that the desired versus actual state can be determined to allow the controller to establish what error, if any, exists and take such as action, or actions, as may be required to minimize the resulting "error" signal at the summing junction. Thermostats and cruise controls are example of such systems. This configurations allows the "equilibrium point" to be set externally while in the latter case that point is established programmatically within the controller, e.g., as in the case of inertial navigation systems, anti-lock braking systems, etc.

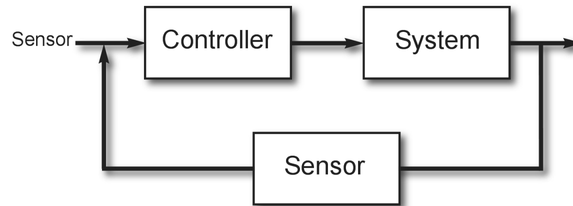


Figure 1.14: Schematic view of a closed loop system with "sensed" output feedback.

In designing embedded systems it is important for characteristics such as latency, phase shift and stability to be taken into account. Figure 1.14 shows a representation of a system with simple feedback represented symbolically. Note that the blocks representing the Controller and System transfer functions can be combined as shown in Figure 1.15. This is equivalent to combining the two transfer functions as follows:

$$G_1 = H_{Controller} H_{System} \tag{1.2}$$

where $H_{Controller}$ and H_{System} represent the transfer functions for the Controller and System, respectively.

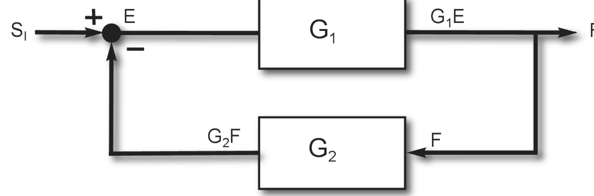


Figure 1.15: A generalized SISO feedback system.

Thus the embedded system represented in Figure 1.14 can be represented by the following:

$$E = S_I - G_2F \tag{1.3}$$

$$F = G_1E \tag{1.4}$$

$$G_2F = G_2G_1E \tag{1.5}$$

which leads to the result that:

$$\frac{f(t)}{s(t)} = \frac{G_1}{1 + G_1G_2} \tag{1.6}$$

and assuming that this system is a LTI (linear, time-invariant) system, the corresponding Laplace Transform can be expressed, symbolically as:

$$\frac{F(s)}{S(s)} = \frac{(s - z_1)(s - z_2)(s - z_3)\ldots(s - z_{m-1})(s - z_m)}{(s - p_1)(s - p_2)(s - p_3)\ldots(s - p_{n-1})(s - p_n)} \tag{1.7}$$

where $s = \sigma + j\omega$, the z_m terms are the zeros of the transfer function and the p_n terms are the poles. The stability, of lack thereof, of this system can then be determined by an examination of the location of the system's poles, p_n, in the complex plane, or by other techniques. An embedded system is said to be "Bounded Input, Bounded Output" (BIBO) stable if any bounded input results in a bounded output. An embedded system's stability may be one of several different types, e.g., unstable, uniformly stable, marginally stable, conditionally stable, etc.

Although it is also beyond the scope of this textbook, a further refinement of this type of a mathematical model for an embedded system would be to include the impact of perturbations, i.e., various types of disturbances, that the embedded system may be subject to such as electromagnetic interference, vibration, frictional effects, variation in loading of motors and actuators, nonlinear effects, effects of stray magnetic and/or electric fields, etc. Note also that when using sensors in an embedded system, it is sometimes necessary to employ various types of signal conditioning, e.g., various filtering techniques, to maintain signal integrity to assure appropriate current/voltages limitations are imposed, etc.

Adaptive embedded systems[19] are employed, when required, to allow them to modify their characteristics to meet, often in real time, variable "environmental" conditions such as power

[19] An embedded system is considered "adaptive" if it is able to reconfigure its program and hardware resources in real time to continue to meets its functional and performance specifications. In some cases degradation in these specifications may be regarded acceptable as long as they remain within defined boundaries.

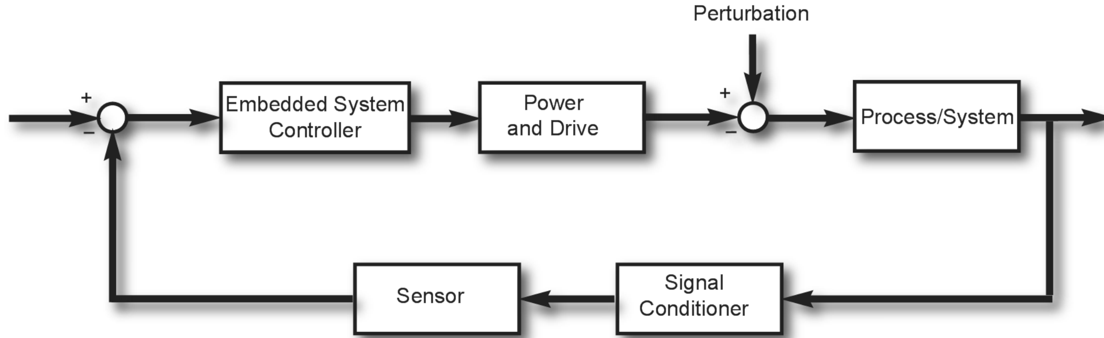


Figure 1.16: An embedded system that is subject to external perturbations

supply fluctuations/degradation and externally variable process and system conditions. Many embedded systems are able to reduce their clock frequencies, enter sleep modes, and operate at lower power levels in response to changes in environmental conditions. Others are able to move tasks between multiple cores to optimize performance and minimize hot spots.

Arguably any system that employs feedback could be considered "adaptive" since the embedded systems is "adapting" its responses based on input data. However, adaptability in the present context refers to adaptation within the controller itself in response, e.g., to changing environmental conditions. Adaptive systems may employ fuzzy logic, neural networks, Radial Basis Functions (RBFs), Kalman filtering, etc., which are often used for approximation, interpolation and to overcome limitations imposed by wavelets, polynomial interpolation, least square and other techniques when multidimensional parameters are involved, as for example, in the case of signal conditioning.

1.5 Embedded System Performance Criteria

Two of the most important considerations for embedded systems are 1) that they perform each tasks correctly and 2) that all of the reactions/responses by an embedded system occur in a timely manner. It is also important that na embedded system be robust[20]. Timeliness, or lack thereof, in the present context can be characterized as Soft,Firm or Hard.[14] A "Hard Real-Time System" (HRTS) is one in which failure could produce a catastrophic result, e.g. failure of a fire alarm system, or a pacemaker.

A "Firm Real-Time System" (FRTS) failure might be an automotive cruise control for which the latest value of the current speed is not available in time for the cruise control algorithm to determine what, if any, corrective action is required. In such cases, the algorithm may be able to use the previously reported speed and still perform the necessary operations to maintain relatively constant speed. A "Soft Real-Time System" error(SRTS) is exemplified by the failure of an ATM which, while perhaps inconvenient, is hardly a firm, or hard failure. Failures of these three types (HRTS, FRTS and SRTS) are usually analyzed in terms of "deadline misses" and their respective impacts on the system. Because such systems are expected to react in real time they are, by their nature, typically asynchronous.

Latency, in the case of an embedded system, refers to the delay $(t_1 - t_0)$ between the time (t_0) when a condition exists requiring a response and the time that the response occurs (t_1). Such

[20]Robustness is defined as resistant to perturbations.

delays can arise as the result of hardware delays in sensors, microcontrollers, peripheral output devices and software delays produced by program-execution overhead, e.g. program execution speed and interrupt servicing[21].

Embedded systems are often asynchronous and receive input data from multiple sources. In such cases, this data must wait until the embedded system is available to accept it. Some input devices introduce a delay between the time an input parameter is sensed/updated and the time at which the data has been "latched" for input to the microcontroller. Latching is often employed to be sure that when data is available from an external device such as a sensor, the microcontroller has sufficient time to complete, or suspend, its current tasks. Suspension of on-going tasks occurs when an "interrupt" request, of sufficient priority, is received. For tasks of lower priority, than the currently running task, or when large data sets are involved, various techniques can be employed to "buffer" the inputs from sensors until they can be processed. Flip-flops, which are bistable devices, are commonly used to latch input, or output data, particularly at the byte level, while waiting for a device, such as a microcontroller, to enter a ready state and subsequently accept the data. Various "shared memory" techniques such as Dynamic Memory Access transfers can also be employed to provide the needed "buffering".

Embedded systems employ various techniques to minimize latency:

- Direct Memory Access (DMA) - this technique, while not normally involving any pre-processing of data, allows I/O to occur relatively transparently without requiring significant CPU overhead. I/O devices can transfer data to/from the embedded system by directly accessing the microcontroller's memory space. A DMA controller, such as that shown in Figure 1.17, is used to facilitate the transfer, once the CPU has defined where the data is located, or, to be stored within local memory.

  In some cases, a region of memory is predefined as assigned to the DMA controller and therefore the CPU's direct involvement in data transfers under DMA control is obviated. The DMA controller can set a flag, or flags, indicating whether new data is available for processing by the CPU, or has been transferred to one, or more, external devices. This technique addresses both latency and bandwidth overhead by allowing the data to be transferred at a rate most appropriate for the external device(s) and whenever it is available.

  In other cases the CPU, under software control, initializes the DMA controller and provides the data addresses for both source and destination and the amount of data to be transferred. Microcontrollers allow the DMA controller, upon request, to take control of the bus and transfer data in a so-called "burst mode". In doing so, the CPU's access to the address bus is usually "tri-stated"[22] to avoid bus conflicts (cf. Figure 1.18). When the transfer is complete the DMA controller returns control of the bus to the CPU. In other cases, a "cycle-stealing" mode is employed by the DMA controller in which case it relinquishes control of the memory bus after each transfer. Note that most DMA controllers have address and length registers that are of different sizes, so that if the address register is larger than the length register it can address a large portion of memory. If the size of the address register is 32 Kbits and the length register is 16 bits, then the DMA controller can transfer data in blocks of 64 Kbytes anywhere within 4 Gbytes of RAM. As shown in Figure 1.17, data may be transferred either to/from memory internal to the microcontroller, or, if necessary to external memory depending on the amount of data to be transferred, latency concerns

[21] Interupts are discussed in section XX

[22] Tri-stating refers to placing a device input or output in one of three states, e.g., high (1), low (0) or high impedance, the latter effectively removing it from a circuit, e.g., a bus. This technique prevents bus conflicts and the possibility of two subsystems attempting to "drive the bus", i.e., apply signals, at the same time.

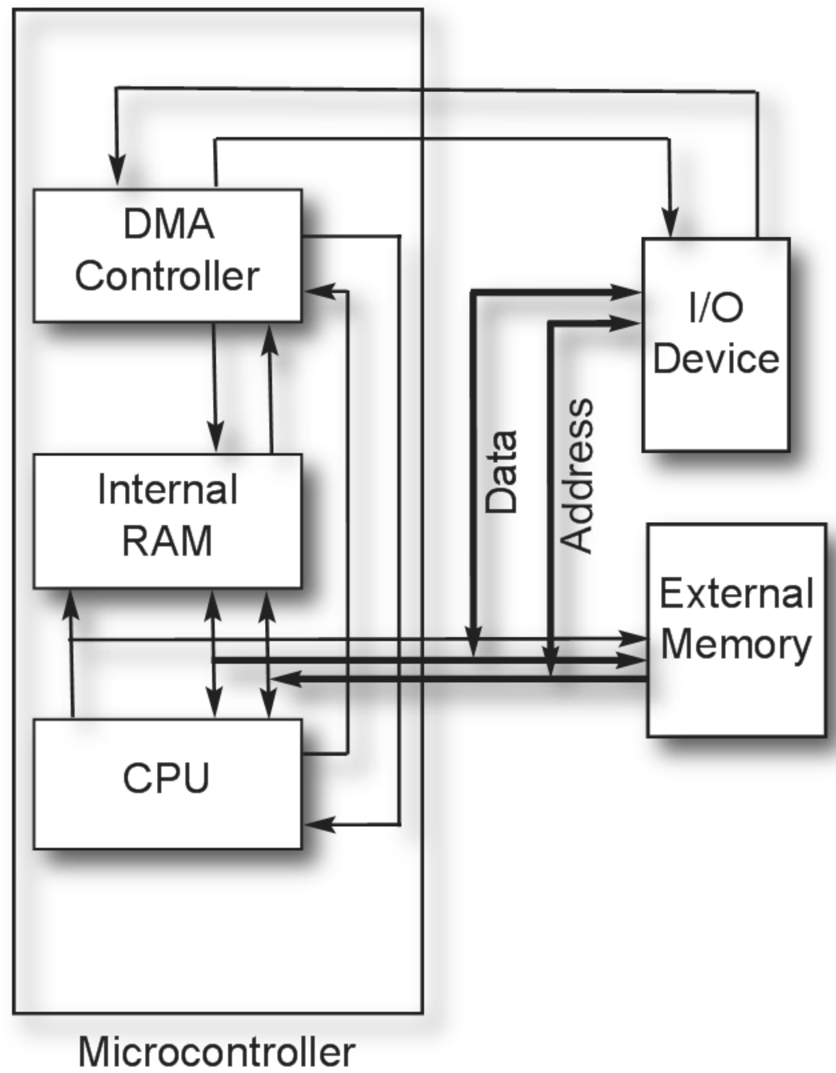
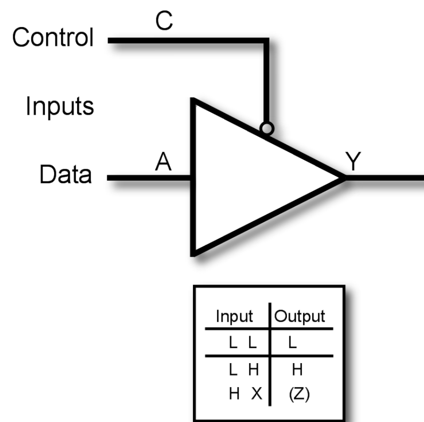


Figure 1.17: Block diagram of typical microcontroller/DMA configuration.

and the application.



Input	Output
L L	L
L H	H
H X	(Z)

Figure 1.18: An example of a tri-state device.

- Looping - Since an I/O device can itself be in an active (“busy”) or waiting (‘idle”) state, an embedded system can remain in a loop waiting for a flag to be set, before continuing with program execution. This has the advantage that having the microcontroller remain in an idle state waiting for data reduces at least some of the latency. But this technique has the limitation that the CPU is unable to accept data from other sources, while in this mode.
- Polling - Alternatively, an embedded system can“poll” status flags for I/O devices to determine if a device: 1) has data available for transfer to the microcontroller, 2) is available for receipt of data, or 3) is busy. Polling may be periodic, or aperiodic, depending on the application.
- FIFOs (First-In-First-Out buffers) and other forms of buffering can be used with external devices to buffer I/O until the microcontroller's resources are available. This technique, as in the case of the use of DMA, can be employed for occasions when data is being gathered faster than it can be handled by the microcontroller.
- Interrupts - interrupt schemes can be employed that interrupt the CPU only when I/O needs to occur.

1.5.1 Interrupts

An interrupt is a request initiated by a device requesting that the CPU be “interrupted” to service, i.e., process, some particular task. If the device initiating the interrupt has a task of sufficiently high priority, i.e., a higher priority than that of the task being conducted by the CPU at the time the interrupt request was received, and there are no other interrupt requests of higher priority waiting to be processed, then:

- the interrupt request is accepted,
- interrupts of the same, or lower, priority are blocked,
- the current task is suspended (which requires that the state of the CPU[23] be fully preserved to allow the interrupted task to be completed at a later time),

[23]The accumulator, Program Status Word,(PSW), program counter and any related registers are typically stored on the stack when a task is suspended to service an interrupt to allow the interrupted task to be fully restored.

and,

- the requested task is processed.

If other interrupt requests of higher priority than that of the original task exist, they will all be processed before the CPU returns to continue processing of its original task. If the microcontroller is engaged in a task and a series of increasingly higher priority interrupt requests occur, before the preceding interrupt has been fully serviced, then the stack will contain state information about each of the interrupted tasks that has been suspended by a higher priority task request and the original task, except for the highest priority interrupt, which will then be serviced by an interrupt service routine (ISR). It is important to fully preserve the state of each lower priority task on the stack, e.g., by storing the contents of the accumulator, program counter, program status word and any other registers involved.

The microcontroller can then restore the previous task(s) and continue program execution until the next interrupt occurs. In this case, the overall latency of the embedded system is the time between when the input data is ready (latched) and the time at which the microcontroller is able to input the data, process it and produce the required results. In the case of an output device, for example, a hard disk, UART, or device which has "busy" states, the microcontroller must wait until the device is ready to accept data/commands, i.e., is in a "non-busy state".

1.5.2 Latency

Thus the total maximum latency (L_{max}) for a system can be defined as:

$$L_{max} = L_{sensors} + L_{microcontroller} + L_{peripherals} + L_{actuators} + \dots \tag{1.8}$$

where, $L_{microcontroller}$ is a function of program execution times, time required to service interrupts, wake-up time[24], boot time[25], etc., and each of the Latency parameters represented in Equation (1.8) represent worst case conditions.

Interrupts introduce delays because of the time required to service a given interrupt and the fact that they are handled in order of priority. In the worst case, a low order priority interrupt will have to wait until all higher order interrupts have been serviced before it is serviced. In some applications, as long as the embedded system responds within a predefined period of time, the system is performing satisfactorily. In other cases, different response times are required depending on the state of the processes being monitored/controlled.

Thus for interrupt-driven I/O, an additional latency factor is "priority" which determines task precedence. While higher priority tasks, whether input or output, are addressed earlier than lower priority tasks, in some applications, all tasks may be assigned the same priority, so that no task takes precedence, over any other. Alternatively, as discussed previously, the embedded system may poll input/output devices to determine whether or not such devices are busy, have data available for input to the microcontroller, are available to transfer/receive data, etc. However, polling can result in the significant waste of machine cycles when polling for data that is not available and/or conditions that don't exist very often. Interrupts also make it possible to detect conditions internal to the microcontroller such as timer/counter overflow, data

[24]Some microcontrollers are programmed to go to "sleep" when nothing interesting is occurring in order to conserve power. They can be awakened periodically, or by the by the occurrence of an interrupt. In such cases, when the required tasks are completed the microcontroller can then be returned to a sleep state until needed again. It may be necessary in some applications to take the latency associated with returning from a sleep state to an active state into account.

[25]In some embedded systems, in the event that the embedded system becomes "locked-up", the embedded system re-boots itself after a predetermined period of time.

available in an internal UART, an internal UART being available for character transmission, that a multiplication product is available, etc.

Therefore, a microcontroller responds to interrupts by first determining if more than one interrupt has occurred. If so, the microcontroller then services the interrupts on the basis of priority by halting execution of the current task, storing all the information required to restore that task and then "servicing" the interrupt request[26], e.g., by collecting the latched input data, taking whatever action maybe required, such as storing the data, subjecting it to numerical processing and/or taking appropriate action such as setting/transmitting output parameters for actuators, data to transmission channels, data to display devices, etc. It should be noted that most microcontrollers have a reserved interrupt referred to as an None-Maskable-Interrupt (NMI) which has priority over all other interrupts. This interrupt is usually reserved for catastrophic events such as hard disk, or other serious failures.

[26]The routine responsible for responding to the interrupt request is referred to as an Interrupt Service Routine (ISR).

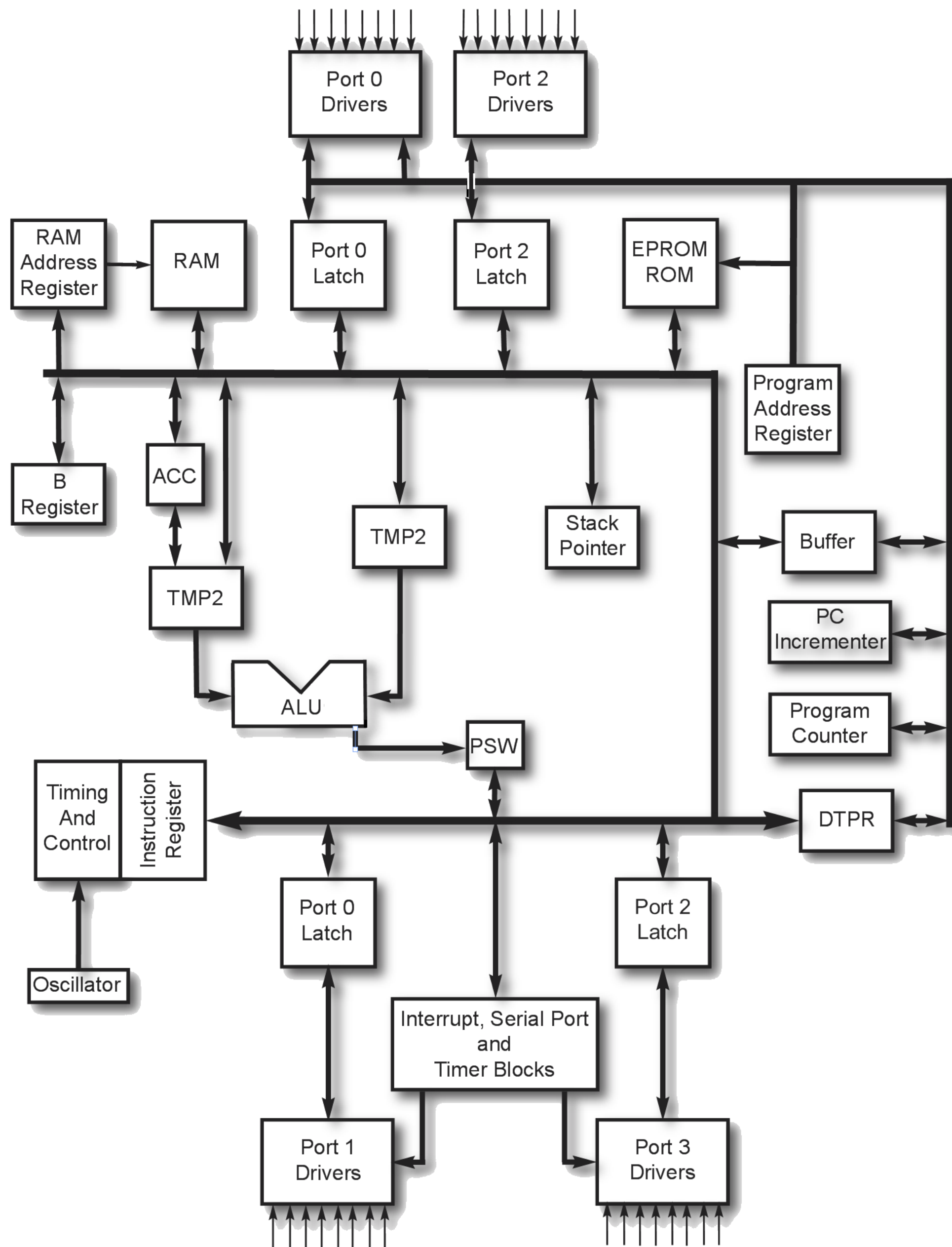


Figure 1.19: Intel 8051 Architecture.

1.6 Embedded Systems Subsystems

Microcontrollers need a wide range of subsystems if they are to be the basis for complex embedded system applications, e.g., voltage A/D and current/voltage D/A converters, mixers, pulse width modulators (PWMs), programmable gain amplifiers (PGAs), instrumentation amplifiers, etc., as shown in Table 1.1. In many applications, embedded systems involve multiple analog/digital data input channels. since data is often provided by a wide variety of sensors, communications channels, etc. The embedded system employs a microcontroller/microprocessor[27] for numerical computation and logic functions that are to be performed on the input data, e.g., numerical processing of input data and the decision-making based thereon. Output drivers for a variety of devices, e.g., motors, linear/rotary actuators, LCD and other types of display devices, communications devices for I2C, CAN, SPI, RS232[28], etc., are also required as part of the embedded system and interconnect directly with the microcontroller and the actuators, displays, PCs, networks, etc.

Typically, an embedded system involves a combination of analog and digital devices, under the control of a microcontroller, that collectively serve as a feed back/control system to monitor, and control, a wide variety of electromechanical, electro-optical systems, chemical process, etc. Embedded systems can be as simple as a fan controller, used to control one, or more, fans, to maintain pre-defined temperature(s) in a server, or consist of a complex network of embedded systems collecting and sharing data, as well as, handling various output/control functions.

In addition, embedded systems may also be required to provide real time actions in terms of responding within predefined time limits to certain critical input conditions, or lack thereof, with appropriate output responses as in the case of anti-lock brakes, deployment of air bags, response to failure of one or more devices, initiating critical shut down procedures, gathering data at sufficiently high rates to allow for data processing and appropriate control functions, etc.

Modern day embedded systems must of necessity, and to the extent feasible, also be adaptable to changing market requirements, avoid steep learning curves for the designer, etc. Furthermore, issues such as low component costs, minimal printed circuit board real estate requirements, ease of manufacture, minimal reliance on external components, in-circuit debugging/programming capability, support for standard communications protocols, and interoperability with other devices and systems are also important.

The advent of microprocessor/microcontroller technology led early adopters to conclude that embedded systems of the future would consist simply of one or more analog-to-digital and digital-to-analog converters conjoined with a microprocessor/microcontroller, as shown in Figure 1.20. The basic design philosophy was to immediately convert all input signals to their digital counterpart, process the resulting digital form of the inputs and then, if required, convert the digital results back to an analog signal, via digital-to-analog converters for connection to the external world.

This view was strengthened by the fact that analog signals can be degraded by component tolerances, undesirable nonlinearities, sensitivity to electrical noise (EMI), changes in environmental conditions such as temperature and humidity, vibration, limited current/voltage dynamic

[27] The distinction between microcontroller and microprocessor has become somewhat ambiguous in modern parlance and the two terms are sometimes used interchangeability with little regard for their differences.. In the discussion that follows the term microcontroller shall refer, at a minimum, to a microprocessor, memory and some form of I/O capability all within the confines of single chip, that functions as a "system on a chip".

[28] Within the family of RS232 type drivers are RS422 and RS485 protocols which are specific hardware protocols as opposed to data protocols and provide support for master/slave operation, as well as, significantly better noise immunity and longer transmission paths.

Table 1.1: Some of the types of subsystems available in microcontrollers.

Amplifiers	Prog. Gain	Instr	Transconductance	Comparators	OpAmp
A/D Converters	Delta-Sigma	SAR	Incremental		
D/A Converters	Multiplying	Current DAC	Voltage DAC	-	-
Dialer	DTMF	-	-	-	-
Counters	8 Bit	16 Bit	24 Bit	32 Bit	-
Timers	16 Bit Tach/Timer	8 Bit	16 Bit	24 Bit	32 Bit
Random Sequence	PRS8	PRS16	PRS24	PRS32	-
PWMs	PWM8	PWM16	PWM24	PWM32	-
Analog Muxs	AMUX4	AMUX8	RefMUX	Virtual	Sequencer
Filters	Low Pass	Bandpass	High Pass	Notch	Adaptable
Digital Comm	UARTs	USART	CRC Generators	-	-
Digital Comm	SPI	SPIM	SPIS	CAN	LIN
Digital Comm	IDaTX/IrDARX	I2Cm	I2CHW	USBFS	-
Digital Comm	One Wire	I2C	FlexRay	I2S	-
MAC	-	-	-	-	-
LCD	Character	Segment Static	Segment	-	-
LED	LED 7 Segment	-	-	-	-
Sleep Timer	-	-	-	-	-
LVDT	-	-	-	-	-
Logic	AND	OR	XOR	NAND	NOR
Logic	NOT	XNOR	Logic High	Logic Low	LUT
Logic	Digital MUX	De-Multiplexer	D Flipflop	-	-
Registers	Control	Status	-	-	-
DMA	-	-	-	-	-
Pins/Ports	Analog	Digital Bi-Direct.	Digital Input	Digital Output	-
Logic	AND	OR	XOR	NAND	NOR
Logic	NOT	XNOR	Logic High	Logic Low	
Logic	Digital MUX	De-Multiplexer	-	-	-
Registers	Control	Status	Shift	-	-
Mixer	-	-	-	-	-

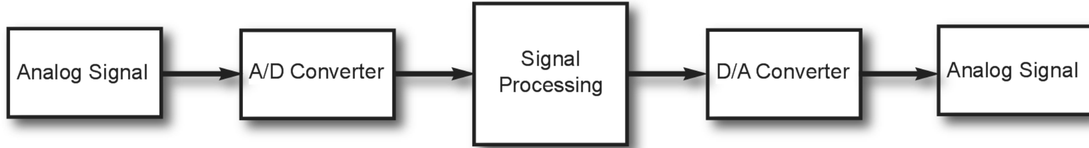


Figure 1.20: A simple example of analog signal processing.

range, storage of analog information in other than digital formats.

However, while it was soon realized that analog signal processing is an important part of many embedded systems (as is the need for minimal power consumption, fast response times and converting all data to a digital format) converting analog signals to a digital form before processing gave rise to other problems, e.g., aliasing. digital filtering overhead, etc.

When dealing with digital methods for gathering and processing data, careful consideration must be given to the amount of data gathered per unit time over a given period and the rate with which such data is gathered. It is assumed, in the following discussion, that the signal under consideration is a continuous time, "well-behaved" signal, and that the goal is to convert the analog signal into its digital equivalent under conditions sufficient to allow the original analog signal to be accurately reconstructed. Sampling at a rate below the highest frequency component of a given signal can give rise to a phenomenon known as "aliasing" as shown in Figure 1.21. In this case a fixed frequency sinusoidal signal is sampled at a rate of once per second while the impled signal derived by sampling, is seen to have a period of approximately 10 seconds.

The Nyquist-Shannon Sampling Theorem, also referred to as the Nyquist or Shannon criteria, requires that under these conditions the sample rate be equal to, or greater than, twice the highest frequency component of the signal, or equivalently, that if the frequency component is B Hertz, that the sample rate be:

$$f_s = 2\beta B \tag{1.9}$$

where f_s is the sampling frequency, B represents the bandwidth (highest frequency component of the signal) and β is a measure of the amount of oversampling, if any. Oversampling becomes important when attempting to minimize anti-aliasing effects particularly where A/D conversion are involved.

Sampling, in the present context, refers to the periodic, or aperiodic, collection of data resulting in a discrete time series. The rate of sampling in terms of samples per sec is usually determined by the application, the hardware used in the embedded system and Equation (1.9). The amount of data gathered per sample is obviously determined by the number of bits (bytes) gathered per sample and the rate is determined by how often a sample is taken.

For example, if each sample consists of two bytes, or equivalently 16 bits per sample, the sampling rate is 200 samples second, and the length of time over which samples are gathered at this rate is 24 hours, the size of the sampled data set, D, is given by:

$$D = (bits\ per\ sample)(\#\, of\ samples\ per\ second)(total\ sampling\, time) \tag{1.10}$$

and therefore:

$$D = \frac{(16)(200)(24)(3600)}{8} = 34.56 Mbytes \tag{1.11}$$

Note that in this example, it is tacitly assumed that the highest frequency component in the sampled signal is 100 Hz for a unit oversampling, i.e., for $\beta = 1$. Furthermore, “sampling” which is often introduced when relying on digital signal processing techniques such as A/D and D/A conversion can result in the loss of information (aliasing), adds additional CPU overhead and introduces potential quantization issues and round-off errors.

Digital filters, while capable of providing an excellent filter response, are an example of excellent characteristics at the expense of data processing time, and therefore latency, which can preclude their use in certain types of control systems. In such cases, analog filters may be employed that while perhaps offering much less sophisticated filtering capability are cheaper, fast and characteristically have a large dynamic range. As in the case of any optimized embedded system design, trade-offs are frequently required in order to provide the best overall solution in terms of response time, power consumption, cost, manufacturability, component count, printed circuit board (PCB) real estate, etc.

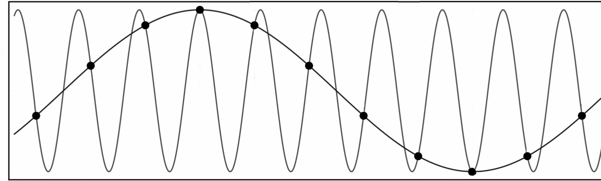


Figure 1.21: Example of aliasing

1.7 Sensors and Sensing

Sensors are devices that convert one or more physical parameters into digital or analog signals for processing and control applications. Such sensors, often referred to as transducers[29], typically convert physical parameters such as temperature, pressure, linear/curvilinear motion of objects (acceleration, velocity, displacement, etc.), salinity, hydrogen ion concentration (pH) wind speed, ocean currents, vibration, presence of toxic materials, fire, proximity of objects, force (linear, torque), fluid[30] flow (velocity, acceleration, displacement, etc.), radiation measurement (low frequency RF, high frequency RF, microwave, ultraviolet, visible light, infrared, etc.), heat flux, stress/strain, chemical signals (e.g., smells),quasi-static electric and magnetic fields (i.e. no radiative), altitude,metal, resistance, capacitance, inductance, electrical power, mechanical power, mass flow, volume flow, etc., into an analog voltage or current, or the digital equivalent.

In addition to “point” measurements utilizing a single sensor, sensors may also be used in groups in both inhomogeneous and homogeneous arrays to provide fields of data over a given surface or volume. Regardless of the physical parameter(s) being sensed (i.e. measured or detected, including the lack thereof) the converted parameter is provided as a proportional voltage, current or digital value which can then be processed by a microcomputer or microcontroller and used for feedback, or feed forward, information in a wide variety of applications.

Some of the more commonly encountered input sensors include microphones, tachometers, thermistors/thermocouples, sonar or other forms of acoustic sensors, pressure sensors, infrared

[29]Transducers may convert one physical parameter into another, but for the purposes of these discussions, the term ”transducer” refers explicitly to a device capable of converting the value(s) of an arbitrary physical parameter into corresponding voltage, current or digital value.

[30]Both and liquids are to be regarded, for the purposes of these discussions, as fluids.

sensors (passive and active), ultrasonic sensors, RFID readers, strain gauges, linear/rotary position sensors, mechanical switches of various configurations, distance (altimeters, ranging sensors, etc.), velocity, acceleration, roll, yaw, pitch, GPSs, and proximity detectors. Common output devices[31]include speakers, electric motors/positioners, wireless connections of various types and protocols, liquid crystal (LCD) displays, and PC connections.

Hundreds of types of sensors have been designed to detect acceleration, displacement, force, humidity, spatial position/orientation, temporal parameters, tactile contact or the lack thereof, biometrics (retinal, fingerprint, DNA, facial, etc.), proximity, speed and a host of other parameters. These devices provide resistance, capacitance, inductance current, voltage, amplitude, frequency, phase, quadrature modulation, and data in binary or other form, as output parameters. All of these devices are susceptible to noise (EMI), aging, temperature, vibration and other forms of degradation of their output signals. Thus it is important to carefully calibrate such devices, often against the manufacturers specifications under a variety of anticipated operating conditions. In some cases, the embedded system can utilize look-up tables (LUTs) and other means to apply corrections to data provided by such devices. Filtering of analog signals can also be used to maintain input data integrity.

In dealing with sensors, it is important to delineate between the "accuracy" of a sensor and its precision. The latter refers to the degree to which a sensor is measuring the quantitative value of a parameter and the latter to how close a series of measurements of a value of a parameter as measured value by a given sensor are grouped quantitatively. Thus it is possible for a sensor to make a series of measurements for a given value of a parameter and arrive at similar values and yet not be a particularly of the "accurate" assessment of its actual quantitative value. Note that precision is defined in terms of the number of significant figures to which the value is measured while accuracy is related to how close the value reflects the true value of the parameter being measured. Accuracy and in some cases precision are also affected by the conversion of analog data to a digital format. Typical microcontrollers have data "widths" of 8-16 bits which can affect both accuracy and precision, depending on the application.

Sensors also have inherent limitations with respect to input signal "dynamic range" and bandwidth, can introduce noise, may suffer from nonlinearities, may be affected by offsets, such to saturation effects and depending on their input and output impedances may require that proper "impedance matching"[32] for both input signals and interfacing to the microcontrollers I/O channels.

In the case of analog signals, "dynamic range", as applied to sensors, is a quantitative figure of merit defined as the ratio of maximum input signal to minimum discernable (detectable) input signal that can be applied to a sensor for which it can produce an output without distortion. Dynamic range for digital signals is defined in terms of the bit error ratio (BER) which is the ratio defined as:

$$BER = (number\, of\, altered\, bits)/(number\, of\, bits\, transmitted) \tag{1.12}$$

where "altered bits" refers to transmitted bits altered by adverse phenomena such as interference, noise, etc.

Care must be taken to insure that impedance "mismatches"[33] don't distort measurements

[31]Note these may also be viewed as transducers.

[32]Impedance matching considerations are an important when dealing with sensors it order to avoid adversely

[33]Impedance mismatch is often used to refer to differences between output impedance of one device and the input impedance of a second device when power transfer is a consideration. In such cases the output and input impedances should not be the same, i.e., "matched", i.e., it is important that the microprocessor not draw significant power from the sensor to avoid the possibility of distorting the value of the parameter being sensed.

and/or adversely effect the values input/output by the microcontroller from sensors/output to actuators, peripherals, motors, actuators, etc.

1.7.1 Types of Sensors

Currently, wireless sensors, ultra-low power, plug-and-play, MEMs-based[34], PWM output and sensor fusion are the most popular areas of sensor technology. Communications protocols used in conjunction with sensors include CAN/CANOpen, Devicenet, Ethernet IP, TCP/IPWireless, USB, and various proprietary networks. In terms of embedded system designer priorities, reliability, accuracy, durability (ruggedness), noise immunity, sensitivity, sensing range, resolution, ease of maintenance, ease of setup and environmental protection are the most import concerns and in descending order. Currently, for modern embeddem systems the most popular types of sensors are vision senors, wireless, rotary position, proximity, linear displacement and photoelectric.

1.7.1.1 Optical Sensors

Various types devices are available for measuring of optical parameters and the presence/absence of radiation in the optical portion of the electromagnetic spectrum. These devices

- Photomultipliers - these devices have multiples stages of light amplification that results in a current that is proportional to the intensity of illumination. These devices use low work function[35], e.g., alkali metal-based coatings, to convert photon impacts into electrons and hence currents.
- Pin Diodes/Photodiodes - these devices are semiconductors, e.g., PN junctions which produce a current which is proportional to the intensity of the illumination.

1.7.1.2 Capacitive Sensing

In recent years, capacitive sensing has become increasingly more common in applications such as automobiles, mobile phones, a wide range of consumer electronics including home appliances, stereos, televisions, a wide variety of consumer products as well as a broad range of military and industrial applications. Capacitive sensing offers a number of advantages over its mechanical counterpart, e.g., no mechanical parts, completely sealed interface, etc. Capacitive sensing is based on a very simple relationship between the area of a capacitor, the distance between two conducting surfaces of a capacitor, d, the permittivity of free space, ϵ_0, the relative dielectric constant, ϵ_r and its capacitance, C, as follows:

$$C = \epsilon_0 \epsilon_r \frac{A}{d} \tag{1.13}$$

Current estimates suggest that as many as 2.5 billion buttons and switches have been replaced by this technology. This "non-touch" sensing technology is sufficiently sensitive in some applications to allow it to be employed in applications requiring nanometer resolution. In addition to replacing the traditional "buttons", capacitive sensing techniques are also used to function as "sliders", proximity detection, LED dimming, volume controls, motor controls, etc. Capacitive sensing is capable of sensing the presence of conductive materials, including fingers and providing proximity sensing for a wide variety of touch pads and touch screens. Many capacitive sensing applications consist of a conducting surface often protected by glass or plastic that senses

[34] Micro-electromechanical or MEMs sensors can be on the order of 5x5x1 mm in volume, and are available as pressure, acceleration, gyroscopic, gas flow, temperature and other types of sensors.

[35] Work functions represent the minimal energy required to eject an electron from a solid and defined as $W = h\nu$ where ν is the minimum photon frequency required for photoelectric emission to occur fr a given solid surface.

A typical capacitive sensing arrangement involves two separate conducting surfaces often created by traces on a printed circuit board which represent a capacitance of 10 to 30 picofarads. Assuming that the traces are protected by an insulating material perhaps 1 millimeter in thickness, an approaching finger represents a capacitance in the range of 1-2 picofarads.

1.7.1.3 Magnetic Sensors

There are various forms of magnetic sensors which operate, in some cases, by closing or opening switch contacts (reed relays), utilizing the Hall effect to vary current flow, sensing current flow etc. Magnetic sensors also take advantage of the Curie Point[36].

They are used in in a wide variety of applications including:

- Brushless DC motors
- Pressure sensors
- Rotary encoders
- Tachometers
- Vibration sensors
- Valve position sensors
- Pulse counters
- Position sensors
- Flow meters
- Shaft position sensors
- Limit switches
- Proximity sensors

Magnetic sensors, unlike other types of sensors do not, for the most part measure a physical parameter directly. Instead magnetic sensors react to perturbations in local magnetic fields in terms of strength and direction to determine the state of electric currents, direction, rotation, angular position, etc.

The units of magnetic field are Gauss, Teslas and gammas and they are related by:

$$10^5\, gamma = 10^{-4}\, Tesla = 1\, gauss \tag{1.14}$$

Magnetic sensors can be classified in terms of the range of magnetic field strength which they sense as follows:

- Low Fields - magnetic fields whose strength is less than 1 gauss.
- Earth's Field - magnetic fields in the range from 1 microgauss to 10 gauss.
- Bias Magnetic Fields - magnetic fields of strength greater than 10 g.

and, since a magnetic field is a vector field, both magnitude and direction of the field, a magnetic sensor can use use director, strength or direction and strength to measure a particular parameter and therefore:

[36] The Curie point refers to the temperature at which the magnetic properties of a substance change from ferromagnetic to paramagnetic. If the temperature is subsequently reduced to below the Curie Point, the substance becomes ferromagnetic again.

- a vector magnetic sensor utilizes both magnitude and direction,
- an omnidirectional magnetic sensor uses magnetic field in one direction,
- a bidirectional magnetic sensor measures magnetic field in both directions,

and,

- a scalar magnetic sensor utilizes magnetic field strength only.

Finally, magnetic sensors can be further classified as:

- Anisotropic magneto-resistive (AMR) sensors - these are used for measuring position in terms of angular, linear position and displacement in fields comparable to that of the Earth. They consist of thin film resistors that are created by depositing nickel-iron on silicon whose resistance can be varied by several percent in the presence of a magnetic field.
- Bias magnetic field sensors - these sensors use Hall devices,
- Fluxgate sensors - these sensors are often used in navigation systems.
- Hall effect sensors - these sensors sense current in a small plate as a result of the Lorentz force $F = q(vxB)$ on electrons. This in turn produces a Hall voltage which is directly proportional to the magnetic field.
- Magneto-inductive sensors - these sensors utilize a single winding coil which has a ferromagnetic core in the feedback loop of an operational amplifier to form a relaxation oscillator. Changes in the the ambient magnetic field alter the frequency of the oscillator by as much as 100%. A shift in frequency can be detected by a microcontrollers "capture/compare"[37] functionality.
- Search coil sensors - this sensor relies on the fact that a changing magnetic filed induces a changing electric field in a coil. However, search coil sensors require that either the magnetic field is varying or the coil is moving.
- Squid sensors - Based on the Josephson junction is the most sensitive and is capable of sensing fields as low as 10^{-15} gauss and as high as $9x10^4$ gauss (9 tesla) which is equivalent to fifteen order of magnitude.

1.7.1.4 RF

RF sensors detect fluid viscosity, fluid contamination, fluid flow, linear and rotational speeds, displacement and position in automotive and aeronautical applications. They normally operate in ranges from DC to 1 Gigahertz, -170°C to 1000°C. This type of sensor can be used with both ferrous and non-ferrous materials as well as, glass, plastic, liquids and composites. This type of sensor measures the magnetic susceptability and electric permittivity within a predefined volume of space.

1.7.1.5 Ultraviolet

This type of sensor often relies on the physical characteristics of Zinc Oxide which is transparent when irradiated with visible light and opaque when irradiated with ultraviolet in the 220-400 nanometer range. Silicon photodiodes are also used for UV detection but silicon also absorbs UV which makes it less desirable as a sensor.

[37] "Capture" refers to a microcontroller's ability to time the duration of an event. Compare refers to its ability to compare the values in two registers and subsequently trigger an external event.

1.7.1.6 Infrared

InfraRed (IR) proximity sensors in applications such as TV remote controls, wireless connections between PCs and printers utilize light in the range from 600 - 1200 nanometers which is not visible to humans.Various optical techniques are employed including:

- Modulated IR - which modulates and IR bean to control devices remotely. Modulating the carrier provides better to signal to noise ratios (SNRs) which can be important in IR-noisy environments
- Reflective IR - this technique relies on measuring IR reflected from an object. However, it can be adversely affected by background thermal radiation, but is inexpensive to implement.
- Transmissive - detects objects located between an IR transmitter and receiver.
- Triangulation - offers the best performance for proximity detection using a focused beam and a receiver array to measure the angle of reflection from an object.

1.7.1.7 Ionizing Sensors

Smoke detectors use a chamber which contains a radioactive source, e.g., Americium-41, to provide alpha particles that ionize the oxygen and nitrogen present in the air in the chamber. There is also a set of plates, one positively charged, and the other negatively charged. The negatively charged plate attracts the ionized Oxygen/Nitrogen ions. Similarly, the electrons are attached to the positive plate. The net result is a small, but continuous current flow. However, in the presence of smoke, particulate matter in the smoke binds with the Oxygen/Nitrogen ions thus making the charge of each neutral and reduces the current. This reduction in current is then detected and triggers an alarm.

Photoelectric detectors are also used as detectors in smoke alarms by either monitoring the amount of light reaching a detector which is reduced in the presence of smoke, or by measuring the amount of light scattered from a beam in the presence of smoke.

1.7.1.8 Other Types of sensors

While there are obviously a great many types of sensors, the three most common forms of sensors are strain gauges, thermistors and thermocouples. The techniques used for making measurements using these three types of sensors are similar and shall be treated briefly.

Since the resistance of a length of wire is a function of length (L), cross-sectional area (A) and a physical quantity that is determined by the type of wire being considered referred to as resistivity (ρ) given by:

$$R = \rho \frac{L}{A} \tag{1.15}$$

and therefore,

$$dR = \rho \left[\frac{AdL - LdA}{A^2} \right] \tag{1.16}$$

so that,

$$\Delta R = \rho \frac{\Delta L}{A} \tag{1.17}$$

if the change in area is negligible the change in resistance is a linear function of the change in length (dL), assuming that ρ is not an explicit function of length (L) or area (A).

Strain gauges [38] are designed to measure a dimensionless parameter called “strain” which is defined as the deformation of an object when a load is applied, expressed as:

$$Strain = \frac{(\Delta L)}{L} \tag{1.18}$$

where ΔL is the length of deformation and L is the original length. It should be noted that strain can be either compressive or tensile (stretched). Strain gauges are designed to convert mechanical deformation into some form of electronic change, e.g., in resistance, inductance or capacitance, which is proportional to the strain.

Strain gauges measure deformation in only one direction, and therefore, if the deformation is in two or three dimensions, multiple strain gauges are placed such that they are orthogonal. In addition, most materials tend to be at least somewhat anisotropic[39] so that the same stress applied in orthogonal directions may result in different amounts of strain.

The three basic forms of strain:

- Bending strain, sometimes referred to as “moment strain”, is defined as the amount of strain resulting from a given force.
- Poisson strain is a measure of the elongation and thinning of an object that occurs as the result of stress applied to an object.
- Shear strain - Shear strain is a strain that is parallel to the face of a object that it is acting upon, as shown in Figure 1.23.

Stress is defined as:

$$\sigma = \frac{F}{A} \tag{1.19}$$

where F is orthogonal to the area, A, and can exist in five different states:

- Compression - caused by external forces applied to an object which cause adjacent particles within a material to be push against each other resulting in “shortening’ of the material.
- Flexure, also referred to as bending.
- Tension - caused by external forces applied to an object which cause adjacent particles within a material to be pulled away from each other resulting in ”stretching.
- Torsion - occurs when a material is “twisted’
- Shear - occurs when adjacent parts of a material “slide” away from each other as shown in Figure 1.23. Shear may be either vertical or horizontal and in the presence of bending, both occur. The shear angle is defined as:

$$\theta = arctan(\frac{\Delta x}{L}) \tag{1.20}$$

Many sensors use stress and strain to measure parameters such as angular displacement, linear displacement, pressure, compression, flexure, torque, force, acceleration, etc. Figures 1.24 and 1.25 show an application of a strain gauge to a duraluminum tensile test specimen.

[38] Strain gauges were invented by Simmons and Ruge in 1938.

[39] If the properties of a material are independent of direction the material is said to isotropic, otherwise the material is said to be anisotropic.

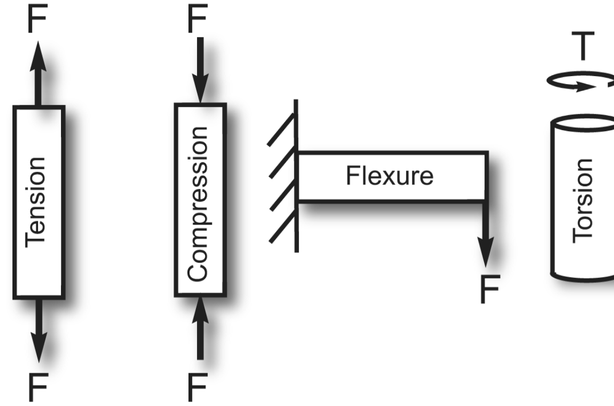


Figure 1.22: Examples of Tension, Compression, Flexure (Bending) and Torsion.

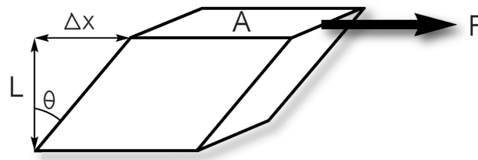


Figure 1.23: Example of shear force.

1.7.1.9 Thermistors

Various techniques are used to sense temperature, but the most common thermal sensors are thermistors, which are sintered semiconductors materials whose resistance is highly dependent on temperature. Simply stated, a thermistor is a semiconductor, with either a positive or negative temperature coefficient, whose resistance is a function of the ambient temperature. Modern thermistors are based on oxides of cobalt, copper, iron, manganese and nickel. Thermistors are sensitive to static charge and their use is typically restricted to temperature ranging from 0°-100°C. Carbon resistors are often used for extremely low temperature sensing, e.g., $-250° \leq -T \leq -150°$ and have a very linear negative temperature coefficient in this range.

In the simplest case, a thermistor can be characterized by the following relationship:

$$k = \frac{\Delta R}{\Delta T} \tag{1.21}$$

where k is referred to as the temperature coefficient and ΔR is the change in resistance for the corresponding change in temperature, ΔT. Depending on the type of thermistor, k can be either negative or positive.

However, most thermistors do not exhibit a linear relationship except over for small temperature ranges. The resistance of thermistors typically lies within a range from $1k\Omega - 100k\Omega$. Thus the resistance of leads attached to a thermistor need not be taken into consideration. Metal oxides are used to produce thermistors with negative temperature coefficients (NTCs) and barium/strontium compounds are used when positive temperature coefficients (PTCs).

A more accurate representation of the change in resistance of a thermistor can be approximated

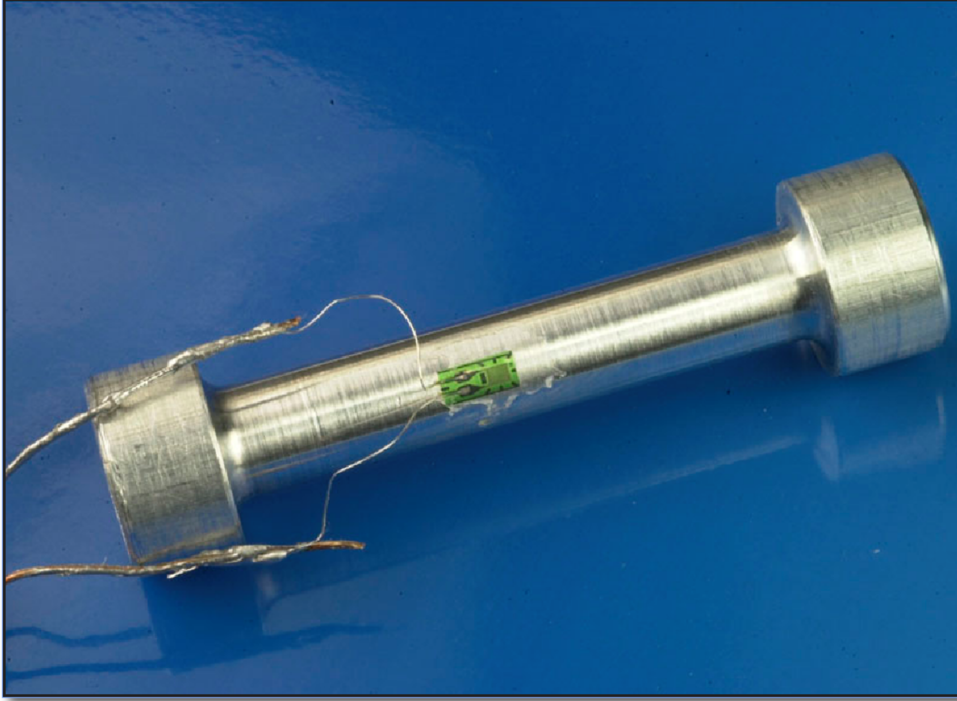


Figure 1.24: Strain Gauge applied to a duraluminum tensile test specimen.[40]

by:

$$\frac{R(T_1)}{R(T_2)} = A^{(T_1 - T_2)} \tag{1.22}$$

where T_1 and T_2 are temperatures, in degrees Kelvin, and A is an empirically derived value, less than 1. However an even better approximation to the relationship between resistance and temperature for a thermistor is given by:

$$ln(R) \approx a_0 + \frac{a_1}{T} + \frac{a_2}{T^2} + \frac{a_3}{T^3} \cdots + \frac{b_n}{T^n} \tag{1.23}$$

This is often further approximated as:

$$R = exp\left[a_0 + \frac{a_1}{T} + \frac{a_3}{T^3}\right] \tag{1.24}$$

The Steinhart-Hart[20] equation is an empirically-derived relationship with three constants that relate the resistance to a corresponding temperature:

$$\frac{1}{T_c + 273.15} = A + B\ln(R) + C\ln(R)^3 \tag{1.25}$$

The three unknowns can be easily determined by employing three data points, 1) the lowest temperature, 2) the highest temperature and 3) a value midway between the 1) and 2).

It can also be expressed as a 3rd order, logarithmic, polynomial with three constants, i.e.,

$$\frac{1}{T_K} = A + B\ln(R) + C\ln(R)^3 \tag{1.26}$$

Figure 1.25: Closeup of a Strain Gauge.[41]

where **A**, **B**, and **C** are empirical constants, **R** is the thermistor's resistance in Ohms. and $\mathbf{T}_K$ is the temperature in Kelvins. Generally speaking, the error in the Steinhart-Hart equation is less than 0.02°C.

A still more useful equation, that provides the temperature in Celsius, is given by:

$$T_C = \frac{1}{A + B\ln(R) + C\ln(R)^3} - 273.15 \tag{1.27}$$

Many thermistors are available with parameters **A**, **B**, and **C** defined. If for a particular thermistor these parameters are not available their respective values can be calculated by using three points in the conversion table, provided by the manufacturer, and solving for these constants. The minimum, maximum, and a middle value for the temperature range of interest are useful points to employ in determining the parameters. The cost of thermistors is primarily determined by the accuracy of the their resistance versus temperature characteristics and therefore the exponential nature of thermistors becomes an advantage.

For a thermistor with a tolerance of n, the possible temperature error is:

$$(1+n)\,R(T_k) = (1+n)\,A^{T_k} = \left[A^{\frac{\ln(1+n)}{\ln(A)}}\right]A^{T_k} \approx \left[A^{\frac{n}{\ln(A)}}\right]A^{T_k} = A^{\left[T_k + \frac{n}{\ln(A)}\right]} \tag{1.28}$$

which shows that a thermistor's resistance tolerance can be represented as a temperature shift. This shift can be removed by a single point calibration by subjecting the thermistor to 25 ° C and measuring its resistance, e.g., if its resistance represents a temperature of 26.2 ° C, then the

embedded system will need to impose an offset of 1.2° offset in order to determine the actual temperature. In some applications involving thermistors the user user has access to an offset register via the GUI and can make the necessary calibration, prior to making a measurement.

A useful heuristic is the fact that a thermistor resistance uncertainty of n% is equivalent to a temperature shift of approximately (n/3)$^\circ$C and can be used to determine if calibration is necessary. The decision as to whether to use (1.21), (1.22), (1.24), or (1.27) depends on the application and the required accuracy, cost cost constraints, available computation time and other factors.

1.7.1.10 Thermocouples

Thomas Johan Seebeck[42] discovered that when two dissimilar metals are in contact with each other in the presence of a thermal gradient, a voltage is produced that is a function of the types of metals and the temperature. The "Seebeck effect"[43] is also referred to as the "thermoelectric effect" and it can be expressed mathematically as:

$$\Delta V = \alpha \Delta T \tag{1.29}$$

where α is referred to as the Seebeck coefficient and T is measured in Kelvin. If two wires, e.g., copper and constantan[44] , are joined together the temperature can be determined by measuring the voltage between them as shown in Figure XX. Copper-Constantan thermocouples, also referred to as type "T" thermocouples, produce approximately $40\mu V$ per °C ($22\mu V$ per °F). However, in order to measure the voltage, metallic connections must be made to them. If the connections to the device measuring the voltage, e.g., a digital voltmeter, are made of copper, then two additional junctions, viz., copper-to-copper and copper to constantan, are introduced as shown. The copper-to-copper junction will not introduce an additional Seeback voltage since it does not involve dissimilar metals.

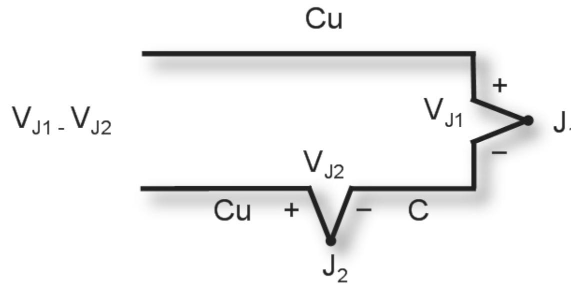


Figure 1.26: Seebeck potentials.

However the constantan-to-copper junction will introduce a voltage, V_{J2}. Therefore the measured voltage is:

$$V_{J1} - V_{J2} = \alpha T_1 - \alpha T_2 = \alpha(T_1 - T_2) \tag{1.30}$$

Since T is in Kelvin:

$$T_2 = t_2 + 273. \tag{1.31}$$

[42]Seebeck was a German-Estonian physicist.

[43]This effect is also sometimes referred to as the Peltier-Seebeck effect.

[44]Constantan is an alloy of copper (55%) and nickel (45%) with the property that its resistivity remains relatively constant over a wide temperature range. In addition to its use in thermocouples, it is also widely used in strain gauge applications.

where t_2 is the temperature in degrees Centigrade, Equation (1.28) becomes:

$$V_{J1} - V_{J2} = \alpha(t_1 + 273 - t_2 - 273) = \alpha(t_1 - t_2) \tag{1.32}$$

In practice, if J_2 is placed in ice, so that $t_2 = 0°$ C, and Equation (1.30) reduces to:

$$V_{J1} - V_{J2} = \alpha t_1 \tag{1.33}$$

It should be noted that the National Institute of Technology and Standards uses 0°C, as the reference junction temperature, in this case J_2, in NIST published tables for Type J thermocouples.

1.7.2 Use of Bridges for Temperature Measurement

Sensors such as strain gauges and thermistors are devices whose resistance is a function of strain and temperature, respectively. In order to collect temperature and strain data it is necessary to read the resistance of such devices. There are a number of techniques for making such measurements, two of which are based on Ohms Law[21] and have been widely employed:

- A sensitive current measuring device such as a Wheatstone bridge is used to determine the value of the resistance of a sensor whose resistance is a known function of temperature.
- A known current is passed through a sensor whose resistance is a known function of temperature and the resulting voltage is measured.
- A reference voltage is applied to a reference resistor in series with a sensor whose resistance is a known function of temperature and the resulting voltage is measured.

Once the value of sensor's resistance has been determined, the data can be compared to available conversion tables[45] that take nonlinearities and any other transducer-related dependencies into account.

A common technique for measuring temperature using thermocouples is to employ a Wheatstone bridge[46] as shown in Figure 1.27. The value of R_x can be determined by using Ohm's Law[47], as follows:

$$V_1 = i_1 R_1 \tag{1.34}$$
$$V_2 = i_2 R_2 \tag{1.35}$$
$$V_3 = i_3 R_3 \tag{1.36}$$
$$V_x = i_x R_x \tag{1.37}$$

If the voltage, V_w, is zero then:

$$i_1 = i_2 \tag{1.38}$$
$$i_3 = i_x \tag{1.39}$$

and therefore,

$$i_1 R_1 = i_3 R_3 \tag{1.40}$$
$$i_2 R_2 = i_x R_x \tag{1.41}$$

[45] Most sensors are provided with conversion tables or charts prepared by the sensor manufacturer.

[46] The Wheatstone bridge was invented by Samuel H. Cristie in 1833 but later became known as the "Wheatstone Bridge" as a result of the attention drawn to it by Sir Charles Wheatstone in 1843.

[47] Ohms Law, most commonly expressed as V=IR, states that the potential difference measured across a current carrying resistor is directly proportional to the value of the current through the resistor times the value of the resistance.

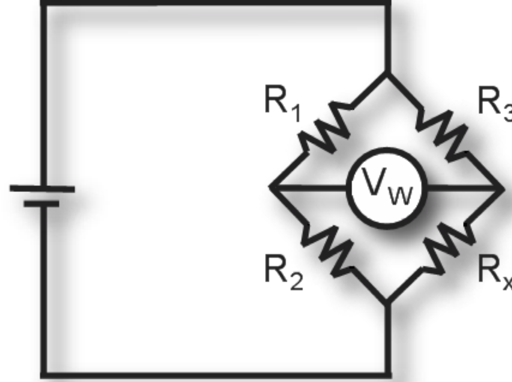


Figure 1.27: The Wheatstone Bridge.

$$\frac{i_1 R_1}{i_2 R_2} = \frac{i_3 R_3}{i_x R_x} = \frac{R_1}{R_2} = \frac{R_3}{R_x} \tag{1.42}$$

$$R_x = \frac{R_2 R_3}{R_1} \tag{1.43}$$

This technique while quite sensitive can be replaced by a much more cost defective and often desirable use of micrcontrollers such as PSoC[48] to gather data from one or more such sensors by employing a technique that supplies a known current to the sensor and then measures the resulting voltage across the sensor.

$$i = \frac{(V_1 - V_2)}{R_2} \tag{1.44}$$

$$V_2 = \frac{(V_1 - V_2)}{R_1} R_2 \tag{1.45}$$

$$R_2 = \left[\frac{V_2}{V_1 - V2}\right] R_1 \tag{1.46}$$

A simplified diagram of such an arrangement is shown in Figure 1.28. A constant current source is used to used to provide a known value of current to the sensor and the resulting voltage is measured utilizing an amplifier and analog to digital converter as shown in Figure. If necessary, an amplifier can be employed that has sufficiently high input impedance to avoid any significant perturbation of the voltage/current to be measured. Obviously the accuracy of this approach depends critically upon the accuracy of the current source and any errors introduced in measuring the resulting voltage, e.g., gain and offset errors.

A second technique is shown in Figure 1.29. In this case, sensors resistance can be determined from the following relationship:

$$\frac{V_{ref} - V_{response}}{R_{ref}} = \frac{V_{response}}{R_t} \tag{1.47}$$

[48]PSoC is a Programmable System on Chip manufactured by Cypress semiconductor that is referred to frequently in these discussions and treated in more detail in Chapter XX.

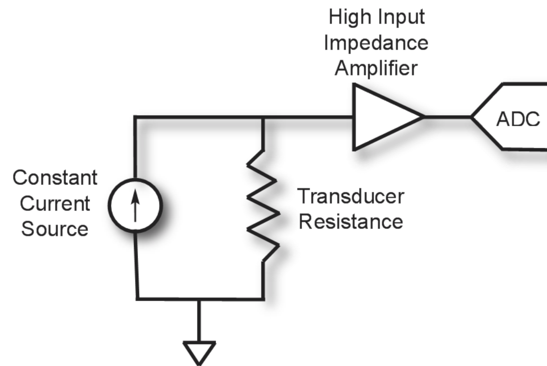


Figure 1.28: Constant current measurement.

and therefore,

$$R_t = \left[\frac{V_{response}}{V_{ref} - V_{response}} \right] R_{ref} \tag{1.48}$$

While this technique should be capable of making highly accurate measurements over a wide range, variances in the values for R_{ref}, amplifier gain and the amplifier's offset voltage can limit its accuracy. Selecting a high quality resistor for R_{ref} an OpAmp with minimal offset characteristics and a very stable reference voltage will allow accurate measurements of R_t to be made over a reasonable range for common resistive transducers.

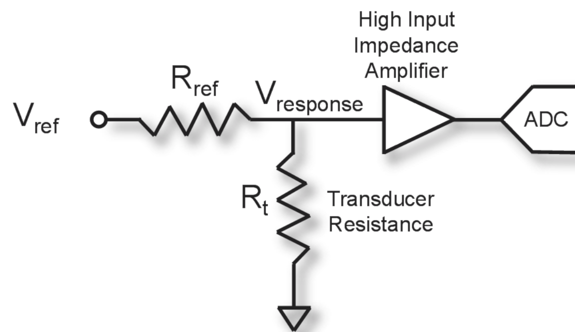


Figure 1.29: Resistive divider.

1.7.3 Sensors and Microcontroller Interfaces

Embedded systems often employ inputs/outputs in the form of analog voltages, currents and/or digital data in order to be able to obtain information from the widest variety of input devices such as sensors, and to output signals to devices such as motors, actuators, display devices, digital transmission channels, etc. This requires that microcontrollers be able to interface with both analog and digital signals of a wide variety. This is often facilitated in part by microcontrollers that have configurable I/O pins which allows some sensors to connect directly to the microconrollers I/O pins.

Input sensors can provide data in the form of analog signals that can be interpreted in terms of phase, amplitude, current, frequency, frequency shift, phase shift, other forms of modulation or

some combination thereof. The input signal may be converted to a digital format by employing an on-chip analog-to-digital converter for data processing, logging and/or retransmission to external devices. However, it is also possible to employ on-chip peripherals such as analog filters and/or various types of analog amplifiers used in conjunction with on-chip analog-to-digital converters and in subsequently output an analog signal without converting the input signal to a digital equivalent.

Handling these types of analog input signals requires the availability of a number of different analog circuits, e.g., analog multiplexers/demultiplexers, analog-to-digital converters, analog comparators, analog demodulators, amplitude/frequency detectors, analog mixers, analog filters, etc.

Similarly, digital input signals require digital multiplexers/demultiplexers, the ability to handle serial, parallel or both data formats, support for various protocols such as I2C, RS232 (UART, USART), CAN, SPI, Firewire, USB, etc. as well as, hardware variants of RS232 such as RS422 and RS485. Parallel data can be handled by some microcontroller's ability to simultaneously input data from a group of pins, e.g., P0-P7, for 8-bit parallel input, data transfers. In some applications multi-byte input data is transmitted to/from the microcontroller one byte at a time using such a technique.

Thus a microcontroller is essentially a CPU that communicates/interacts in a variety of ways, e.g., by responding to interrupts generated by the peripherals or to the state of the microcontrollers input pins, with a variety of on-chip analog/digital peripherals that in turn communicate with input devices such as sensors that provide inputs in the form of analog voltages, currents, frequencies, pulses, digital data etc. The microcontroller can apply the applicable numerical algorithms, invoke the appropriate logic and process this data regardless of original form, apply the appropriate logic sometimes based on the results of numerical processing of the input data and then transmit commands to other on-chip peripheral devices to provide the necessary output signals external devices.

1.8 Embedded System Processing

Embedded systems are capable of providing several different types of functionality, including but not limited to:

- Data collection/processing/transmission - since many embedded systems carry out data processing on data that either began as digital data, or was subsequently translated into the digital equivalent of an analog signal or signals, microcontrollers must be capable of carrying out a number of different low level computational tasks such as addition, subtraction, multiplication and division as well as bit manipulations, shift operations, bit testing. overflow and underflow handling, array manipulations, together with various loop and nesting functions. Computation of algorithms such as Fast Fourier Transforms, digital filtering, etc., are facilitated by special functions such as those provide by a MAC.

  In some cases, if extensive high speed computation of large amount of data must be subjected to complex algorithms where the time to compute is a concern, digital signal processors (DSPs) may be employed that are are optimized to perform high speed, often complex computations. The computational tasks are passed to the DSP or other specialized co-processor and the results of the computation are then made available to the microcontroller via shared memory, or other means. DSPs are specialized microprocessors designed to compute algorithms used in digital imaging, radar, seismic, sensor array, statistical, communications, biomedical signal processing. Some examples of such algorithms are shown in Table 1.2.

Table 1.2: Algorithms used in embedded systems.

Discrete Fourier Transforms (DFT)	
Bilinear Transform	
Real Time Convolution	
Z-Transforms	
Coordinate Rotations/Translations	
Quadratic & Higher-Order Polynomials	
Discrete Fourier Series	
Discrete Wavelet Transform (DWT)	
Least-Squares Computation (LMS)	
Speech Processing Algorithms	
Correlation Algorithms	
Computer Vision Algorithms	
Ray Tracing Algorithms	
Array Processing Algorithms	
Multimedia Algorithms	
Character Recognition Algorithms	
Speech Recognition Algorithms	
Image Processing Algorithms	
Video Processing Algorithms	
Target Detection Algorithms	
Compression Algorithms	
Fingerprint Processing Algorithms	
EEG/EKG Processing Algorithms	
Digital Filters	
	FIR
	IIR
	All-Pass
	Adaptive
	Comb

Field Programmable Gate Arrays (FPGAs) are also used as co-processors as a result of the availability of extremely fast ADCs which among other things have made it possible to apply a variety of digital algorithms to radio frequency (RF) applications. FPGAs are also capable of supporting parallel processing by employing multiple CPUs[49] (multicore), Multipliers-ACcumulators(MACs) and other special function devices such as graphics processors, DMA controllers, etc. on a single chip. MACs provide very high speed multiplications and have the have the capability of adding the products to previous products.

These algorithms, and others are used to select specific input signals to synthesize, compress, enhance, restore, recover and recognize signals, as well as, predict future values and/or interpolate missing values of a signal, In such cases, the microcontroller can act as the prime controller passing data to the DSP for processing and then carry out required operations on the result of the DSP calculations. Specialized math processors such as the Intel 80387 floating point co-processor, have the ability to full control of the address and data busses, are often highly optimized for performing certain specialized functions and therefore may have somewhat restrictive use.

Microcontrollers with integral MACs are very useful for carrying out multiplications for which the product is "accumulated" a common requirement for many digital signal processing algorithms such as vector-dot-products (audio, video, images) Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters, Fast Fourier Transforms (FFTs), discrete cosine transforms (DCTs), convolution algorithms, etc.

For vector-dot-product calculations a typical calculation can be represented by:

$$x = \sum a_i * b_i \tag{1.49}$$

and for convolution calculations,

$$y[n] = y[n] + x[i] * h[n-i] \tag{1.50}$$

similarly for matrix multiplication,

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{1.51}$$

where,

$$x_1 = a_{11}b_1 + a_{12}b_2 + a_{13}b_3 + a_{14}b_4 \tag{1.52}$$
$$x_2 = a_{21}b_1 + a_{22}b_2 + a_{23}b_3 + a_{24}b_4 \tag{1.53}$$
$$x_1 = a_{31}b_1 + a_{32}b_2 + a_{33}b_3 + a_{34}b_4 \tag{1.54}$$
$$x_1 = a_{41}b_1 + a_{42}b_2 + a_{43}b_3 + a_{44}b_4 \tag{1.55}$$

which requires 16 multiplications and 9 additions each time the vector $\vec{x}$ is calculated.

[49] Currently available technology is capable of supporting up to 16 CPUs per FPGA. But the goal is 1000+ per FPGA.

While reliance on co-processors for computation of algorithms, particularly for time critical applications, i.e. applications for which execution times are an important consideration, has proven successful, in recent years embedded systems have begun to be a synthesis of networking, transmission, sensors, data and signal processing, As a result, microcontroller manufacturers have begun to revise some of their standard microcontroller architectures to allow them to perform one, or more, of the functions formally assigned to co-processors, or other external hardware.

Systems processes and control algorithms, of the type addressed by embedded systems, can often be expressed in terms of one, or more, systems of linear, or in some cases partial differential equations that involve resistance, capacitance, inductance, OpAmps, etc., such as:

$$b_{11}\frac{d^2y(t)}{dt^2} + b_{12}\frac{dy(t)}{dt} + b_{13}y(t) = a_{11}\frac{d^2x(t)}{dt^2} + a_{12}\frac{dx(t)}{dt} + a_{13}x(t) \tag{1.56}$$

$$b_{21}\frac{d^2y(t)}{dt^2} + b_{22}\frac{dy(t)}{dt} + b_{23}y(t) = a_{21}\frac{d^2x(t)}{dt^2} + a_{22}\frac{dx(t)}{dt} + a_{23}x(t) \tag{1.57}$$

$$b_{31}\frac{d^2y(t)}{dt^2} + b_{32}\frac{dy(t)}{dt} + b_{33}y(t) = a_{31}\frac{d^2x(t)}{dt^2} + a_{32}\frac{dx(t)}{dt} + a_{33}x(t) \tag{1.58}$$

and equivalently in the form of a system of "difference equations" in the digital domain as:

$$b_{11}y[n-2] + b_{12}y[n-1] + b_{13}y[n] = a_{11}x[n-2] + a_{12}x[n-1] + a_{13}x[n] \tag{1.59}$$

$$b_{21}y[n-2] + b_{22}y[n-1] + b_{23}y[n] = a_{21}x[n-2] + a_{22}x[n-1] + a_{23}x[n] \tag{1.60}$$

$$b_{31}y[n-2] + b_{32}y[n-1] + b_{33}y[n] = a_{31}x[n-2] + a_{32}x[n-1] + a_{33}x[n] \tag{1.61}$$

where b_{mn} and a_{mn} are constants. Note that this system of difference equations involves decrementing of integer values, multiplication and addition only which are all functions easily performed by a CPU that has MAC support.

1.9 Microcontroller Sub-systems

In the following discussion attention shall be restricted to mixed-signal, microcontroller architectures, i.e., microcontrollers consisting of a microprocessor and some number of analog and digital subsystems, often referred to as "modules". In some cases, the functionality of the analog and digital subsystems will be found to be constrained to a limited range and configurability. In others, such as that of PSoC, Cypress' family of Programmable System(s) On Chip, an unusually high degree of variability, configurability and functionality is provided as shall be shown throughout this textbook.

Compared to a typical personal computer, microprocessors are rather limited in terms of memory resources, number of registers, clock speeds, program and data capacity, instruction sets, multitasking capability (if any), etc. Microcontrollers typically have, at a minimum, on- chip support for analog-to-digital, digital-to-analog and perhaps pulse width modulation depending on the manufacturer. Microcontroller instructions are typically 8-16 bits wide and interrupt support is relatively limited compared to personal computers.

Microcontrollers typically include the following subsystems:

- **CPU** - a Central Processing Unit, consisting of an Arithmetic Logic Unit (ALU) e.g. 8051- or ARM-based, microprocessor architectures. The ALU performs mathematical operations

such as addition, subtraction, multiplication and division and logic operations such as equality, less than, greater than, AND, OR, NOT, shift right, shift left, etc. The ALU has access to very fast, local registers that are used to carry out these operations.

The central processing unit, or as it is more commonly known, the CPU, is the heart of the embedded system and responsible for executing a series of predefined and stored instructions, known collectively as "the program". The CPU fetches instructions from memory, decodes them, performs the instruction and stores the results. A program counter (PC) keeps track of the location in memory from which the next instruction is to be fetched. In cases in which the previous instruction has been executed, and the results stored before the next instruction is available, the CPU must then wait for the new instruction to be loaded before it can begin to decode and execute it. This can result from the program residing in relatively slow memory compared to the CPU's execution speed.

In some systems, instructions are preloaded from slow memory to a small amount of fast memory, called "cache", to be fetched from cache as required. The moving of instructions to cache occurs as a "background" task, i.e., does not require CPU involvement and occurs at a rate sufficient to insure that instructions are available "as needed". Alternatively, so-called pipelining is sometimes used which is a technique that fetches instructions before the CPU has finished executing a previous instruction.

Once an instruction has been "fetched" it must be decoded to determine what actions are to be taken by the CPU. Instructions, typically contain specific CPU instructions, specific operands (or their locations) and locations to which the results are to be written. OpCodes might be ADD (addition), SUB (subtraction), MOV (move), etc., and the operands might be characters, numerics or their respective locations, e.g., in local registers within the CPU, or in memory.

- **Memory** - utilized by the CPU for program and data storage that may in fact consist of several types of memory such as SRAM[50], RAM[51] and EEPROM[52]. Some microprocessors/microcontrollers support both on-chip and off-chip memory. But support for off-chip memory is usually rather limited in terms of performance, and/or memory size. Microcontrollers utilizing paged memory[53] require the designer to keep track programmatically of what is stored on each page, accessing the various pages, etc.

Memory can be classified in terms of its read, write, programmability and erasability characteristics, as shown in Figure 1.30 viz:

- ROM - read only memory
- PROM - programmable read only memory
- EPROM - Erasable PROM (UV)
- EEPROM - electrically erasable PROM
- FLASH - Mostly read-only, non-volatile
- RAM - read-write, volatile

[50]SRAM - Static Random Access Memory (requires no refresh).

[51]Random Access Memory - memory whose storage locations can be arbitrarily accessed.

[52]EEPROM Electrically Erasable Programmable Read-Only Memory that is non-volatile used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, e.g., calibration tables or device configuration

[53]A system of memory management that causes non-contiguous parts of memory called pages, to treated as contiguous thus creating a "virtual memory" system.

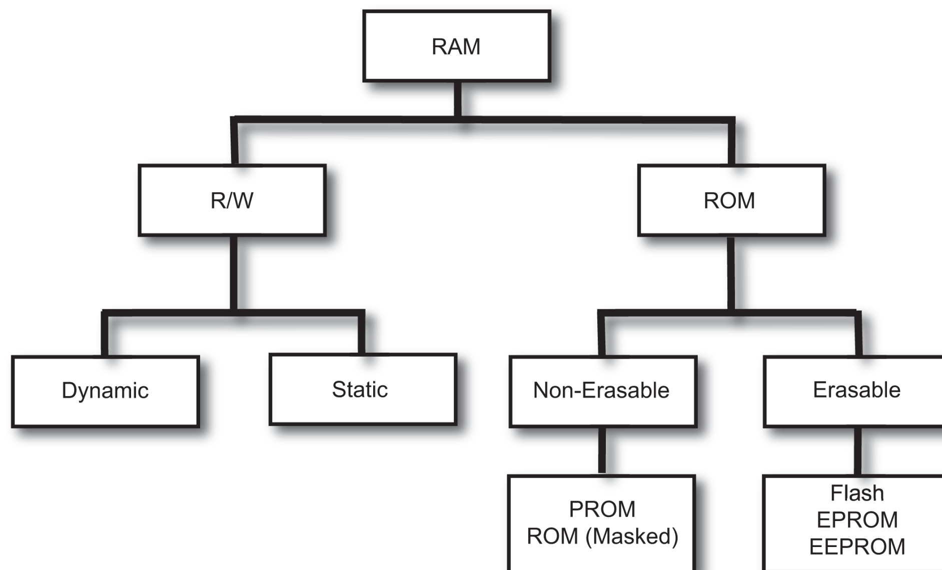


Figure 1.30: Classification of the types of memory used in/with microcontrollers.

Memory for microcontrollers and microprocessors falls into the two broad categories: read/write (R/W) and read-only memory (ROM) and as either volatile or non-volatile depending upon whether or not program and/or data is to be retained in memory, in the absence of supply voltages. Read/Write memory can be further categorized as either static (SRAM) or dynamic memory (DRAM).

Static ram (SRAM) consists of large number of so-called "cells" each of which consists of two inverters as shown in Figure 1.31. This combination of inverters creates a bi-stable device thus making it a viable memory device. A dynamic RAM cell consists of a transistor and capacitor combination for the storage of a bit as shown in Figure 1.32. Static memory is, generally speaking, much faster than dynamic memory but it is also more expensive since it takes as many as four to six transistors (MOSFETs), per bit of storage to implement, but unlike dynamic memory it does not need to refreshed. However, DRAM does provide higher density storage than SRAM.

On-chip Flash memory, which is non-volatile, is typically used for program storage and on-chip SRAM is employed to provide fast program execution, for cache RAM[54] and volatile data storage. Depending on the application, DRAM and Flash, SRAM and Flash, or mixtures of SRAM, DRAM and Flash may be used for off-chip storage. Dynamic memory utilizes as little as one transistor per bit but also requires a capacitor for the storage of each bit. Because of capacitive leakage, it is necessary to refresh dynamic memory periodically, e.g., thousands of times per second, in order to maintain memory integrity.

- **Analog Subsystem** - Microcontrollers employ various combinations of analog functions such as those provided by OpAmps, comparators, current/voltage analog-to-digital (A/D) and digital to analog (D/A) converters, mixers, analog multiplexers, programmable gain amplifiers, instrumentation amplifiers, transimpedance amplifiers, filters, etc.
- **Digital Subsystem** - Similarly, digital functions such those provided by as counters, timers,

[54]Cache memory is defined as memory that can be accessed much faster than main memory.

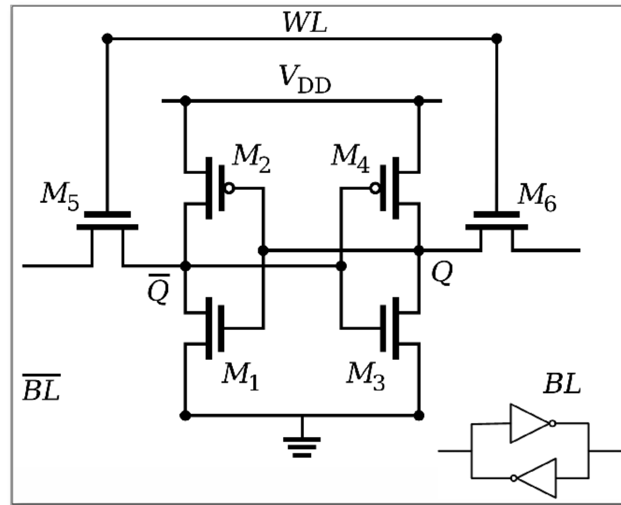


Figure 1.31: An example of an SRAM Cell.

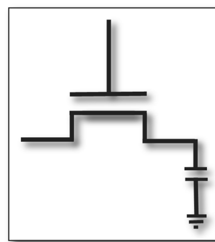


Figure 1.32: An example of a dynamic cell.

CRC, PRS, PWM, quadrature decoder, shift register, logic (AND, NAND, NOR, NOT, OR, Bufoe, D flipflop, logic high, logic low, multiplexer, de-multiplexer, virtual multiplexer, look-up table), precision illumination signal modulators (PRISMs), display (LCD) control and status registers are also to be found in some microcontrollers.

- **Internal Bus Structures** - Obviously, connections between the microprocessor and the various subsystems in a microcontroller are a combination of fixed and variable interconnections and serves as communications pathways between the subsystems, memory, CPU and external world devices and pathways. Microcontrollers that support programmatic changes in internal connections make it possible in some cases to actually reconfigure the internal "wiring" in real time so that optimal utilization of the microcontrollers internal resources can be achieved and the embedded system can adapt to changing conditions and functional requirements.
- **GPIO System** - A microcontroller's interface (General Purpose I/O system) communicates with external devices and peripherals via its pins. In some cases the pins are grouped in sets of 8 and referred to as a "port", e.g., for byte I/O transfers. Whether treated as a group of pins, or individually, all GPIO pins are usually configurable either as output,or input, pins. The impedance characteristics, sourcing and sinking capability of the pins are in some cases configurable depending on the device and the application. Some GPIO interfaces are also voltage tolerant so that a microcontroller operating at voltages below the voltages applied to one or more pins can operate normally, i.e., without being damaged or malfunctioning.

  Microcontrollers that allow groups of pins to be treated as 8-bit parallel ports so that each of the eight pins assigned to a given port serves as a General Purpose I/O interconnect also allow each pin to have its own input buffer, output driver one-bit register and associated configuration logic. In addition each pin is programmable with respect to the driving mode required, independent of whether or not it is part of a multi-bit port configuration.

  Various types of MOSFET-based pin configuration are available are shown in Figure 1.33. Configuration a) is the open drain mode in which both MOSFETs are in an OFF mode causing the output to be in a high impedance state, b) is referred to as the strong, slow drive mode and functions as an inverter, c) is the high impedance mode, d) is the open drain mode and is compatible with I2C interconnections, e) is the pull down mode (resistive) and provides strong drive capability, f) functions as an inverter with strong drive capability and g) is the strong pull up mode. It should be noted that the use of resistors with the MOSFET configurations can affect the rise and fall times of the various configurations.

  Output devices such as motors, actuators and other devices often require more power than can be provided by a microcomputer output channel. Also motors may react inductively to excitation by a microcontroller, so some form of transient protection may be required, or additional power stages may be required to interface such devices to the microcontroller. Mechanical switches are frequently used for this purpose, as are optically-coupled Darlington pairs in conjunction with protective diodes capable of handling inductive voltages. A typical microcontroller is capable of sourcing 10-25 ma to external devices at nominally five volt levels. Whether power amplifiers, power solid state devices such as Silicon Controlled Rectifiers (SCRs), Thyristors, high power MOSFETS, solid state relays, optically-coupled Darlington Pairs or other isolated solid state devices are used to drive motors, actuators, LCDs and other devices requiring significant power, care must be taken to protect the microcontroller and its peripherals, including input devices, from harmful potentials, currents, temperatures, etc.
- **Additional System functionality** - some microcontrollers have an internal boost converter that makes it possible to create voltage levels higher than the available input voltages

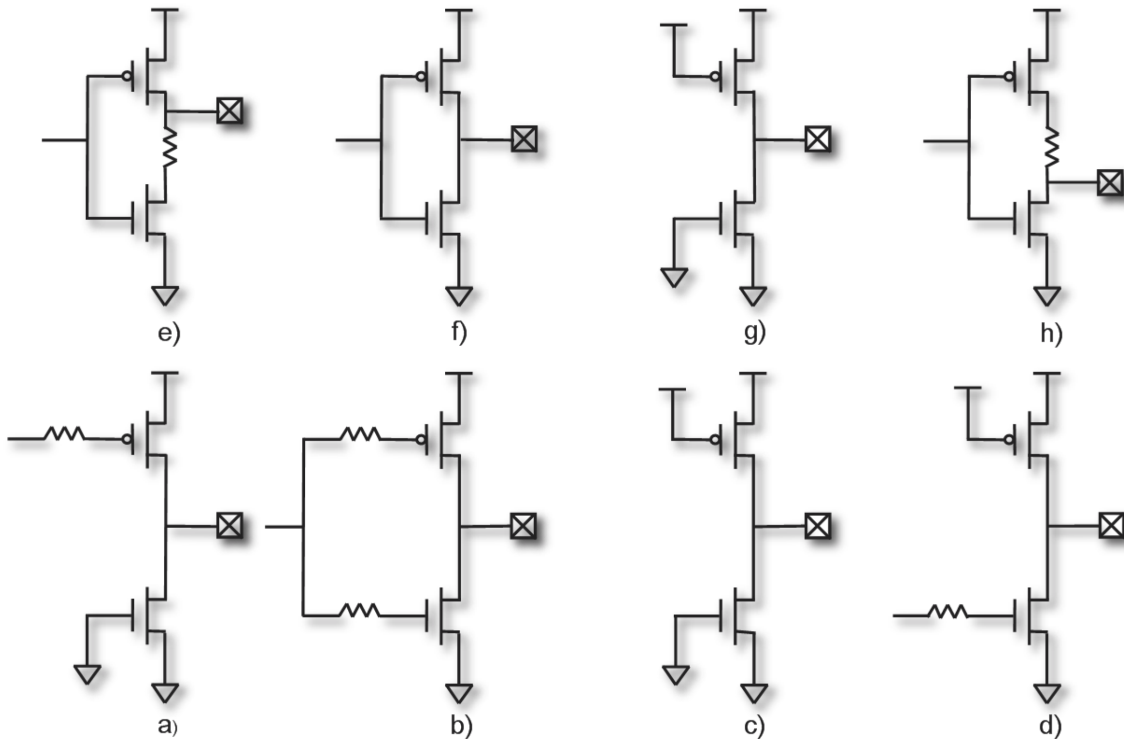


Figure 1.33: Driving modes for each pin are programmatically selectable.

to provide the "desired system voltage level", advanced microcontrollers can also provide additional clock functionality, the ability to monitor the Die temperature programmatically primarily in cases where it is necessary to write to internal EEPROM, an internal DMA controller[55], EEPROM (typically support is provided for erasing an EEPROM sector, writing to EEPROM, blocking reads while writing, and checking the state of a write), a sophisticated interrupt handler, Real Time Clocks (RTC), sleep timers, voltage references, etc.

1.10 Software Development Environments

In the early days of computers, programs were written in what was termed "machine" code. and each instruction was defined in terms of a unique combination of zeros and ones[56]. The next step was to assigning mnemonics, called "OpCodes" which resulted in the development of "assembly" language.[57] The mnemonics assigned to each machine code usually identified some aspect of the

[55]Dynamic memory transfer refers to the ability to transfer data to/from memory without requiring significant CPU overhead.

[56]Unique that is for a particular architecture. There is in principle, aside from any copyright issues, no prohibition on different architectures using the same combination of zeros and ones to represent the same or different instructions

[57]It should be noted that assembly language does not in and of itself offer any new functionality but merely substitutes mnemonics which are related to the instruction's specific function and was far more efficient to work

instruction's function. For example, NOP for No OPperation[58] or MVI for MoVe an Immediate value, e.g., MVI A, 0 represented the instruction code "00111110 00000000" and the action of moving the "immediate" value"0" to the accumulator.

With the advent of the C language[59], developers rapidly adopted it for its portability i.e., its ability to produce applications that could run on a large number of different hardware architectures, and the fact that it provides a somewhat higher level of abstraction than assembly language. Fortunately, C applications does not differ significantly with respect to code size or execution speed for most applications when compared with assembly code. Early development was carried out in Unix-based environments and at the command level using a wide array of text editors, preprocessors[60], compilers[61], assemblers, linkers, debuggers, profilers and a various, preexisting libraries of source and object code. Once graphical user interfaces became ubiquitous, they were soon followed by Integrated Development Environments (IDEs). These environments provided a graphical user interface, or GUI-based, system that supported virtually all of the tools required for development. These IDEs could be hosted in a variety of operating system environments including Microsoft Windows in its various incarnations and the many "flavors" of UNIX, MAC OS, Linux, etc.

Modern IDEs typically consist of preprocessors, compilers, assemblers, linkers, in some cases profilers, debugging tools of various levels of sophistication and collections of predefined functionality in the form of user-defined modules and/or so-called "standard libraries". The available debuggers usually provide, at a minimum, single-stepping of each line of executable source code and the setting of breakpoints, watch points, views of user-defined memory locations, views of registers, etc.

Profilers, while less common in such IDEs, are used to determine how much execution time is spent in a particular location, or locations, in a software program. Such knowledge of "hot spots"makes it possible to "tune", i.e. optimize, the hardware/software performance of an embedded system for efficient program execution. Debugging and profiling are generally best most effective in "single-tasking" environments. Microcontrollers running operating systems can sometimes prove difficult to debug in complex applications.

Typically a designer creates the required source code in an editor-environment (Notepad, VI, Ultra-Edit, EMacs, etc.), or IDE with an integral editor, and then invokes an assembler or C compiler to create an object or assembly language source file. This can result in the generation of warnings, and/or error messages[62], that may arise due to program inconsistencies, syntax errors, etc. If the compiler produces assembly language output, as opposed to an object file, an assembler is then invoked.

Then the "relocation" process, sometimes referred to as "Link-Editing", takes place. This

---

with from a software development standpoint and much easier to debug.

[58]One might well ask why the need for an instruction that did nothing. While it is true that the instruction did not result in any action, it did consume machine cycles which provides a way of introducing delays into a program.

[59]C, a general-purpose computer programming language, was developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories and soon found widespread use for developing portable application software. C is one of the most popular programming languages is use, especially for embedded system applications development..

[60]Preprocessors process the applicable include files, conditional compilation, and macros prior to the invocation of a compiler.

[61]Some C and C++ compilers produce assembly language which is then processed by an assembler to produce object code which is then processed by a linker which in turn produces an executable program for the target system.

[62]"Warnings" are indications issued by the compiler, or assembler, of potential problems with the program which may or may not, at the option of the designer, be ignored as opposed to "errors" which are serious defects in a program and should be corrected before attempting to use the application.

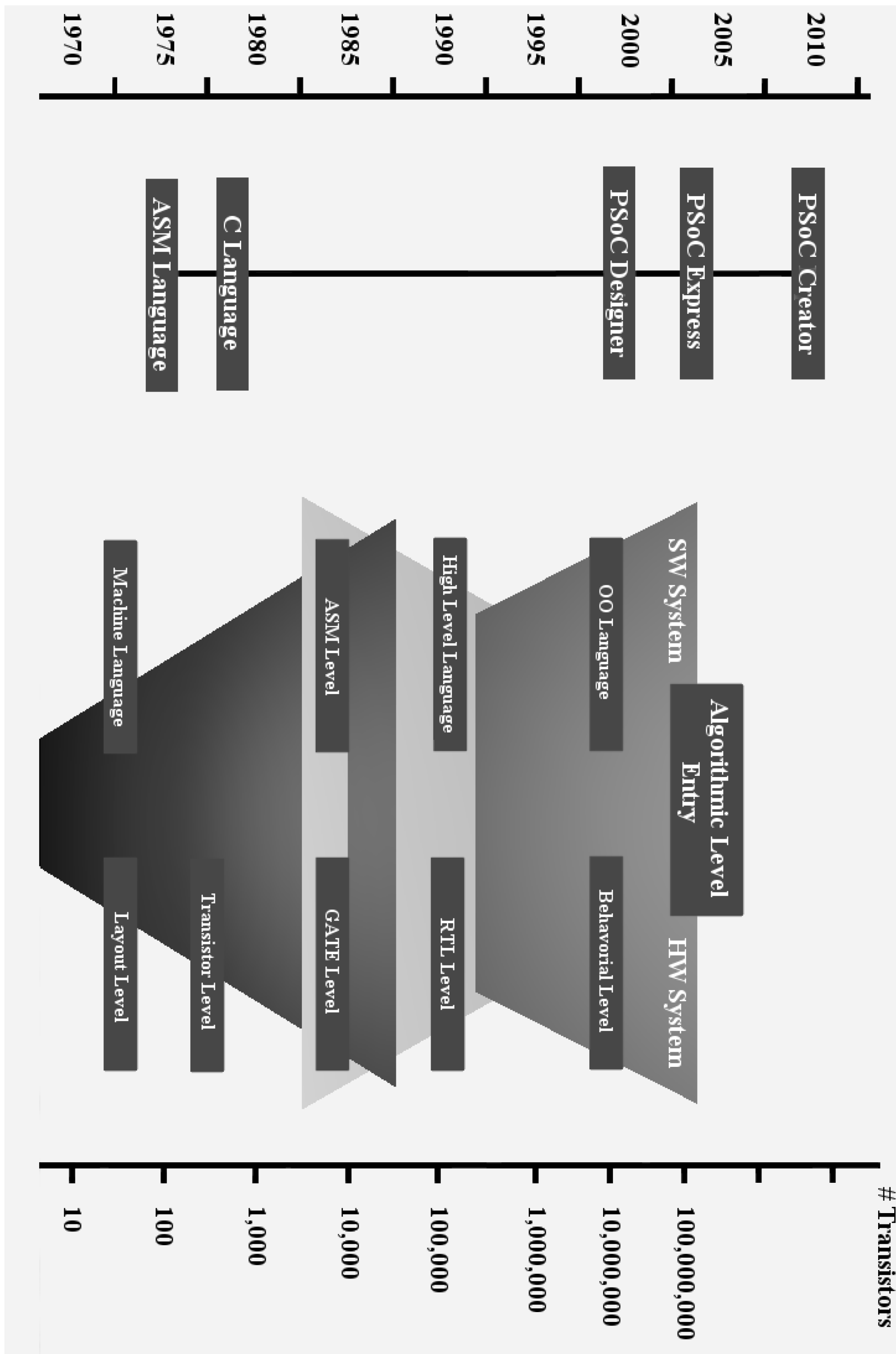


Figure 1.34: Development Tool and Hardware Evolution.

requires that the linker replace the symbolic references, or names, used in each library routine with the appropriate relocatable addresses. In the course of linking all of the applicable object files, the linker must also resolve all unresolved symbols and report unresolvable symbols and other errors and/or potential errors. Linkers also provide symbolic information to assist in the debugging of programs. The resulting file can then be "linked" with other object files to produce the resulting relocatable file. The linker produces a script which is contains information for the locator, e.g., the stack size and location and other information that is used to create an absolute file. The linker-locator phase represents the final stage of the compilation process and among other things determines where various aspects of the executable code will reside in the target's physical memory space. It should be noted that libraries are usually designed to be relocatable, i.e. there is no specific memory address dependance. When the linker-locator has determined where each portion of the code is to be physically located an executable file, referred to as "firmware", is produced which can then be downloaded to the target hardware.

It should be noted that a program may be running in what is, in effect, a "virtual" memory space which appears to the program to be a linear memory space but is actually non-contiguous portions of memory that are mapped into an apparently linear memory space. Debuggers typically provide support for break points, watch points and trace buffers so that a program can be interrupted and the last "n" instructions examined, as well as, the ability to monitor/trap registers and memory locations during execution. Some IDEs provide simulators, although they are often nothing more than programs that allow the developer to test the program's logic. In a later chapters, a modern IDE will be examined in detail and used to illustrate various aspects of embedded system design.

Linkers perform other tasks as well before producing the executable file referred to as "firmware". Once created the firmware is downloaded into the "target" microcontroller. Some microcontrollers support real time debugging via physical handshaking with external hardware and transferring information to external platforms for analysis. In other cases in-circuit emulators (ICEs) or logic analyzers are employed as debugging aids.

There are four basic types of problems encountered with embedded systems:

1. **Coding Problems** - problems results from coding and logic errors. These are the most common problems encountered by developers of embedded system.
2. **Runtime problems** - encountered only at runtime and therefore can be quite difficult to resolve. These problems require careful and often detailed analysis to resolve. Good use can sometimes be made of "isolate and eliminate" techniques. Other cases may require the use of diagnostic hardware such as logic analyzers that allow the system to operate at full speed while providing the ability to closely track the system's activities.
3. **Hard System Crashes** - in such cases the system fails to perform at all. This class of problem can be extremely difficult to resolve because all information leading up to the crash may have been lost. Many newer microcontroller-based systems have some form of hardware debugging that may be of help in debugging this class of problem. Hardware diagnostics that include so-called "deep memory" can sometimes be very effective in diagnosing hard failures by recording the system's history, prior to a crash
4. **Lock Up** - the embedded system gets stuck in some routine or mode and is unable to continue normal program execution. This can be a coding or other problem that only appears when the program is waiting, for some hardware condition to be met that will allow normal program execution to continue, or as the result of timing errors, etc.

Unfortunately, manufacturers have been slow, in many cases, to keep pace with hardware technology when it comes to evolving their respective development environments. Whenever possible

designers should be relieved of the necessity of dealing with many of the low level implementation details of the hardware involved and be able to engage in the development of designs at a much higher level of abstraction. Examination of Figure 1.41 shows that from the time of the advent of the first microprocessor in 1972, until 2001 there was virtually no significant advances in software tool development for embedded system designers other than some modest advances in compiler technology, and minor improvements in debugging functionality, text editors and linkers. Tools such as Source Code Control Systems (SCCSs) have advanced but they tend to be of primary use to large groups of developers working on the same source code. In recent years, IDE's such as PSoC Designer and PSoC Creator have made significant advances in IDE technology and make it possible for applications developers and designers to create the increasingly more complex embedded system applications that incorporate significantly more sophisticated, mixed-signal, embedded systems.

1.11 Embedded Systems Communications

An important component of embedded systems are the channels that support input and output, particularly as they relate to links between various aspects of a system, or a group or groups of systems, that employ standard communications protocols, such as CAN[63], I2C[64], RS232[65], SPI[66] and an ever increasing array of communications schemes and protocols.

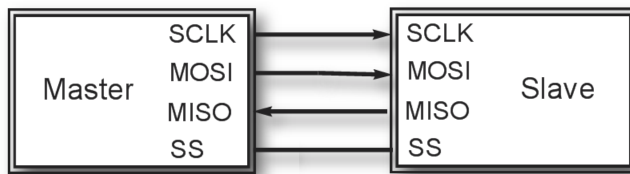


Figure 1.35: A graphical representation of the simplest form of SPI communication.

Communications protocols exist for information exchange within a chip, between chips and for both long and short distances for information transfer to and from an embedded system. They may be state-based, event-based, serial or parallel communication-based, and either point-to-point (data links) or shared media networks (data highways). Master-slave configurations may involve a single master and multiple slaves or multiple masters and multiple slaves, where as point-to-point is a peer configuration and therefore there are neither masters or slaves.

1.11.1 The RS232 Protocol

Early microcontrollers provided limited communications capability and tended to rely on the RS232 protocol operating at baud rates (bits per sec) varying from 60 to 115K baud. This serial data transmission protocol is fundamentally a three wire system, in which one wire is a dedicated transmission line (Tx) a second is a dedicated receive line (Rx) and the third is maintained as

[63] Controller Area Network (CAN or CAN-bus) is a message-based, standard protocol that allow microcontrollers and devices to communicate. It has been used in automotive, industrial automation and medical applications.

[64] The Inter IC bus (*I2C*, I^2C or IIC) is a two wire, bidiectional bus that was developed by Philips originally as a 100 kbit/sec bus. Currently, the protocol supports a maximum data rate of 3.4 Mbits/sec.

[65] RS-232 (Recommended Standard 232) is a standard hardware protocol for serial transmission of binary data signals most commonly used in conjunction with personal computer serial ports and external devices.

[66] SPI (Serial Peripheral Interface) is a full duplex, four wire serial bus serving as a synchronous serial data link. Communication occurs in a master/slave mode with the master devices initiating the data frame.

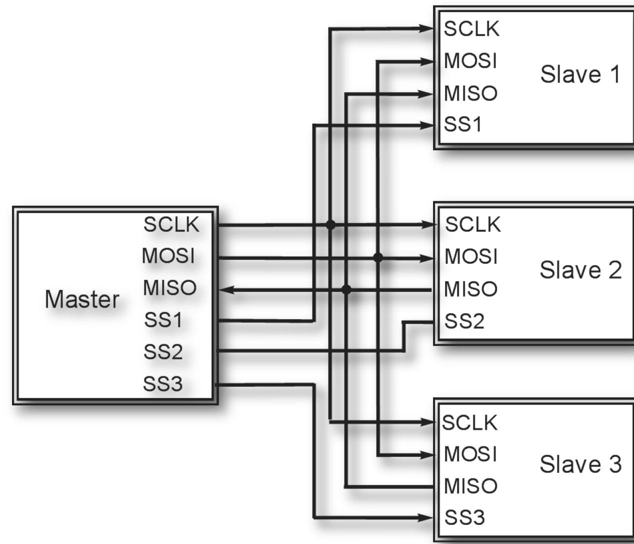


Figure 1.36: SPI - Single master multiple slaves.

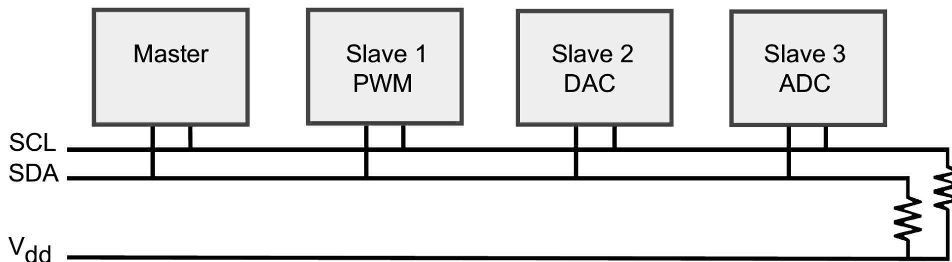


Figure 1.37: Hardware example of the SPI network.

a common ground for both TX and RX. Handshaking, a form of signaling between two systems linked by an RS232 connection is sometimes employed, utilizing additional "control" lines, e.g., Clear to Send, Data Terminal Ready, etc., when implementing RS232 communications in order to avoid collisions and lost data by making sure that when one system is transmitting the other is listening, and vice versa.

Data is commonly transmitted from one location to another in the form of "packets" which may be as little as a single byte. The format of these packets is based on a number of well known, standard protocols. Each packet may include a Cyclic Redundancy Check byte (CRC) or parity bit which are used to detect the receipt of a packet that was corrupted during transmission. This allows the receiver to ask that the packet be re-sent by the transmitter and provides a simple method for assuring some level of transmitted data integrity. Even if the receiver is not able to request a re-transmission of one or more packets, the receiving system is at least aware of the fact that it has received compromised data. While RS323 systems are still in use, they are rapidly being replaced by other protocols such as the Universal Serial Bus (USB).

1.11.2 USB

USB was originally designed to operate at speeds up to 12 Mbps but currently 480 Mbps second implementations are available and in widespread use. It is most commonly used to connect personal computers and a wide variety of peripherals. However, USB does have some significant limitations, e.g., it is limited to a cable length of approximately 5 meters as a result of timing limitations imposed by the USB specification. These limitations can be overcome in some respects, but related protocols such as RS422, and RS485 offer cable length support up to 4800 feet, and Master/Slave support. although at much lower baud rates than USB.

It is a four wire system consisting of Data Plus (D+), Data (-) (which form a differential pair), V_{bus} a five volt power line and ground. Data is transmitted in packets separated by idle states. Current drain is limited to 500 milliamps. Pullup resistors on D+ and D- enable a host such as PC to determine whether it is connected to a low or full speed. USB 1.0 (low speed mode) USB 1.1 (full speed mode) and USB 2.0 (high speed mode) operate at 1.5 Mbits/sec, 12 Mbits/sec and 480 Mbits/sec, respectively. Low speed, full speed and high speed data voltages are 3.5 volts peak-to- peak, 3.5 volts peak-to-peak and 400 millivolts peak-to-peak, respectively.

Communications takes place asynchronously with error detection/correction, and device detection/configuration occurring automatically USB supports several data flow types: Bulk (aperiodic, burst mode, large packets), Control (aperiodic, burst mode, host initiated response/request), Interrupt (bounded latency, low periodicity) and Isochronous (periodic, continuous data transfer, e.g., audio and video).USB data transfers employ packets and each block of data transferred which begins with the host transferring a token that identifies the type of transfer that will occur. Data is transferred in the direction identified in the token followed by a handshake packet is sent to determine whether or not the data was transferred successfully.

1.11.3 Inter-Integrated Circuit Bus (I2C)

The Inter-Integrated Circuit (bus) or I2C is effectively a small area network (SAN) protocol. It was created to facilitate communications between integrated circuits on a printed circuit board and is limited in terms of line distance to $\approx$ 4 meters. Both I2C and SPI rely on a clock signal (max 100KHz) on one wire (SCL), data (SDA) on a second wire and a third wire for common ground. I2C is a bidirectional system, with the direction of data determined by the I2C protocol and no limit on the length of a data transferred. Slave addresses are 7-10 bits and each byte transferred is acknowledged. I2C speeds fall in the range from 100Kbits/second to 3.4 Mbits/second. Distinct start and stop conditions are imposed and slaves each have a 7-10 bit address. The master generates the clock, sets the start/stop conditions, transmits a slave address and determines the direction of data transfer.

1.11.4 Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI), originally developed by Motorola, is a protocol that was created primarily for communications with peripheral devices. Data is transferred synchronously but the data is transferred along with the clock signal and therefore the clock rate is variable. SPI is a master-slave protocol and the master controls the clock signal. There can be multiple slaves but no data transmission unless a clock signal is present. SPI can be operated as either a "single wire" system or in full duplex mode (transmission in both directions simultaneously.

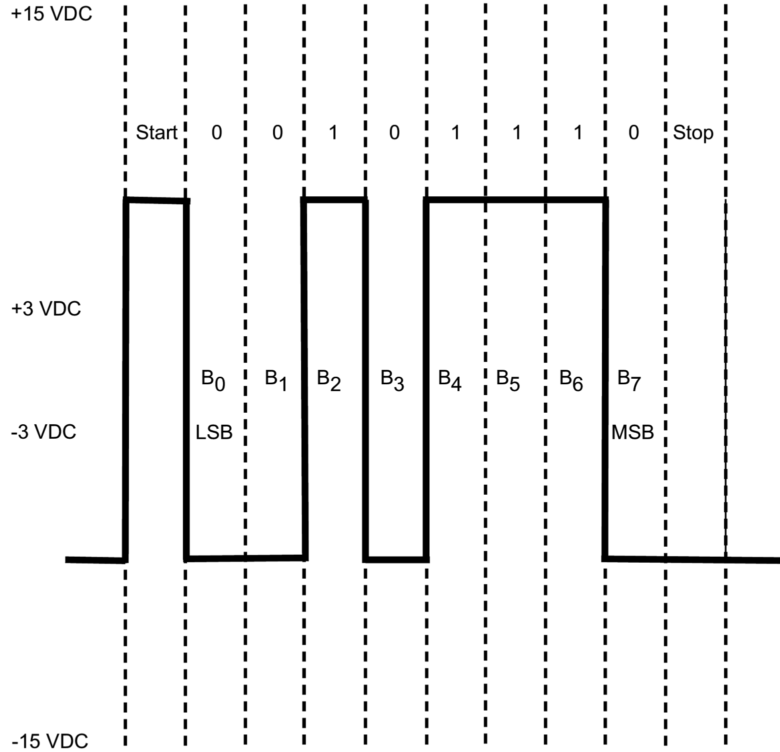


Figure 1.38: The RS232 protocol (1 start bit, 8 data bit, 1 stop bit).

1.11.5 Controller Area Network (CAN)[67]

The CAN bus was developed in the 1980s as a low cost, multi-master, serial bus specifically intended to be capable of operating in electrically noisy environments. CAN supports Master-Slave, Peer-to-Peer and Multi-Master operating modes. It was first used in an automotive environment by Mercedes-Benz in 1992. It has since become "the standard" in the automotive industry and is supported by a variety of controllers for controlling/monitoring air bags, door locks, the vehicle's power train, anti-locking brakes, windshield wipers, rain detection sensors, engine timing, the dash panel illuminators/indicators, seat heating, seat belt systems, seat position systems, navigation aids, automotive voice/data communications, cruise control, mirror adjustment, radio/CD player systems, etc.

The physical data path is provided by ribbon cable (RC), shielded twisted pair (STP) or unshielded twisted pair (UTP). Transmission is non-synchronous, so that any node on the bus is able to transmit provided that the bus is not then in use. However, it is also possible for multiple nodes to initiate transmissions contemporaneously, in which case bitwise arbitration is invoked to determine which message has the highest priority.

Four types of messages are supported:

1. Data Frame[68] - this is the frame used to transmit data.
2. Error Frame - this frame alerts the other nodes on the network that the data integrity of

[67] Refer to ISO 11898 for a detailed specification.
[68] Frame refers to the format used to package a message for transmission.

the data frame has been compromised and instructs the master to re-send the data frame.

3. Overload Frame - this message occurs when a device is unable to receive data.
4. Remote Frame - is a frame request for data to be transmitted.

Every node on the bus is able to listen to the bus traffic and therefore should a node detect an error in a transmission it is able to request that the transmitter, either a master or another node, resend the message. The transmission of a frame begins with the transmission of a Start Of Frame (SOF) bit which is then followed arbitration field, either 11 or 29 bits, which defines the type message and the node from which the message originated. Next the data is transmitted beginning with 4 bits to define the length of the data, then follows the data. Next the cyclic redundancy field is sent. The transmitter computes the CRC, places it in the data frame prior to transmission and then upon receipt by the receiver, the receiver calculates its own CRC and compares it to that of the transmitter. If they are the same value then the receiver assumes that the data integrity has been preserved, otherwise the receiver sends back a message indicating that the data frame needs to be re-sent.

1.11.6 Local Interconnect Network (LIN)

The LIN bus[69] is another bus employed by the automotive industry that functions in single-master/multiple-slave modes. It is typically used in conjunction with the CAN protocol in order to reduce costs. The LIN protocol is much cheaper to implement but has lower performance capability so that it often serves as a subnet to CAN.

LIN, a "single-wire"[70] serial communications protocol is based, in part, on the UART[71] and is designed to handle low demand, automotive applications such as power windows, lights, door controls, and other low demand applications. The maximum data rate for a LIN network is 20 kbits/second. The LIN message format is shown in Figure 1.40. The LIN architecture is self-synchronizing so that nodes do not require crystals or resonators. The master determines message priority and order, controls error handling and provides the systems clock reference. Slaves are limited to a maximum of sixteen and listen for messages with their respective IDs. Although there can be only one master, a slave can function as a master. It should be noted that both CAN and LIN can be interconnected with higher-level networks, if required. The message frames contain a synch byte followed by an ID byte that includes information about the sender, the intended receiver(s), the purpose of the message and the field length of the data.

The frame begin with a break consisting of 13 dominant bits[72] The next filed is the synch which is defined as x55. This field slaves to adjust their baud rates so that they are synchronized with that of the bus. After the synch field has been transmitted, 1 of 64 possible ID fields is transmitted. O through 59 are data frames, 60-61 contain diagnostic data, 62 is reserved for user-defined purposes, and 63 is reserved for future use. The byte representing this field contains two parity bits and the remaining lower six bits are reserved for the ID. Slave response is a field containing from one to eight bytes of data followed by an 8-bit checksum field. Two methods are employed in creating the checksum: 1) the bytes in the data field are summed or by summing the data bytes and the ID. The latter is referred to as the "enhanced checksum;".

---

[69] Refer to ISO9141 for full details.

[70] The phrase "single-wire" is somewhat misleading, but refers to a system in which data transmission takes place using a single wire referenced to ground.

[71] Universal Asynchronous Receive/Transit (UART) protocol - A start bit is followed by 7-8 data bits which in turn are followed by stop bit(s).

[72] Dominant bits are defined as zeros. Recessive bits are ones.

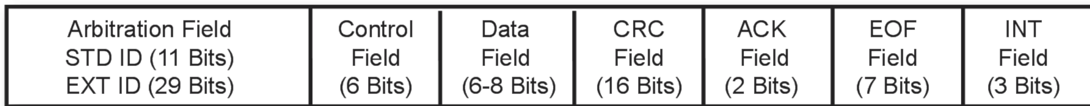


Figure 1.39: CAN frame format.

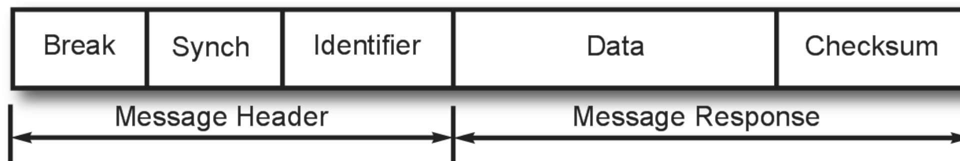


Figure 1.40: LIN frame format.

1.12 Programmable Logic

Because of the inherent cost in designing ICs and the sophistication of the equipment and techniques for manufacturing them, the most economic way of producing them is in large quantities. However, many IC designs are needed in relatively small quantities and ideally, an IC should be manufacturable in small quantities if required but producible in large quantities if needed. This has given rise to a family of programmable logic devices, as shown in Figure 1.41, which can be economically manufactured in large numbers, but can also be programmed to provide large numbers of various relatively low-volume configurations. Programmable devices are available that are field-programmable, some of which are erasable and reprogrammable to allow field updates and develop prototypes which if successful can then b manufactured in high volume as conventional integrated circuits. The The permanent form of programmable logic are either mask programmed, or employ either fuses or anti-fuses[73]. The primary manufacturers of such devices have been Actel, Altera, Atmel, Cypress, Lattice, Lucent technologies, QuickLogic and Xilinx.

The first programmable logic device appeared in 1984 and consisted os 320 gates, packaged as a 20-pin device capable of operating at speeds up to 10 MHz. The first filed programmable gate array (FPGA) appeared in 19895 and consisted of 100K gates represented more than 10 million transistors and was capable of speeds as high as 100 MHz.

There are three basic types of Programmable Logic Devices (PLDs):

1. Programmable Read-Only Memory (PROMs)
2. Programmbale Array Logic devices (PALs)
3. Programmable Logic Arrays (PLAs)

The earliest, user-programmable, solid state device that could be used to implement logic circuits in the field was the Programmable Read-Only Memory (PROM) . The address lines were used as the input and the data lines as the output. However, a PROM used for this type of application is inherently more complex from a hardware perspective than is really necessary. PROMs were subsequently followed by the Field-Programmable Logic Array (FPLA), also referred to as a PLA.

[73]Fuses are links, i.e., connections that can be opened and anti-fuses are potential connections that can be “linked’, i.e. connected.

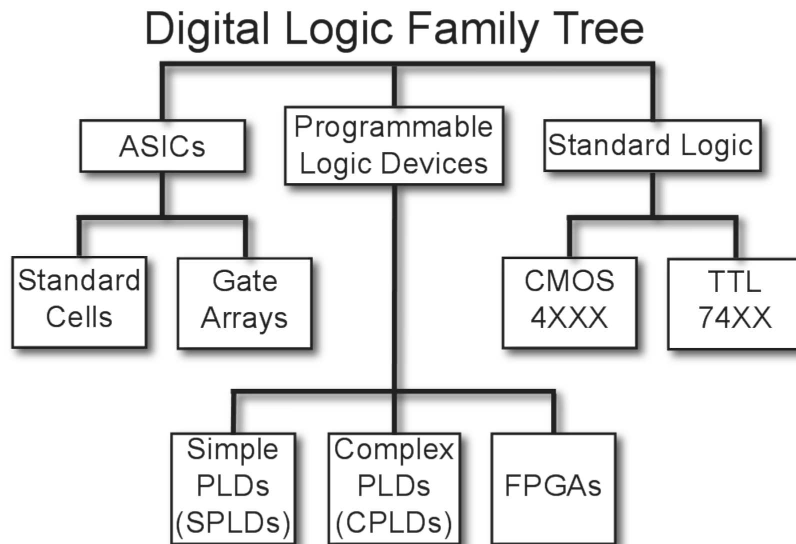


Figure 1.41: Digital Logic family tree.

A PLA consists of two planes, or levels, one with AND gates and a second with OR gates. The links in both the AND and OR arrays are programmable which has made PLAs very versatile. However, PALs only allow the AND plane to be programmed, i.e., the OR plane connections are fixed. This makes PLAs less versatile than PALs but has the advantage that the ORs switch faster than than their programmable link counterparts. In the case of PROMs, the AND array is fixed and the OR array are programmable

A PLD employed systems of so-called "MacroCells" consisting of simple combinations of gates and a flipflop. Each MacroCell can be configured to provide various Boolean equations in hardware and it has input and output connections that are used by the Boolean equation. The resulting equation combines the state of an arbitrary number of inputs to produce and output that, if necessary, can be stored in the integral flipflop until the appropriate clock signal occurs. PLAs and PALs are characterized by the number of AND gates, number of OR gates and the number of inputs.

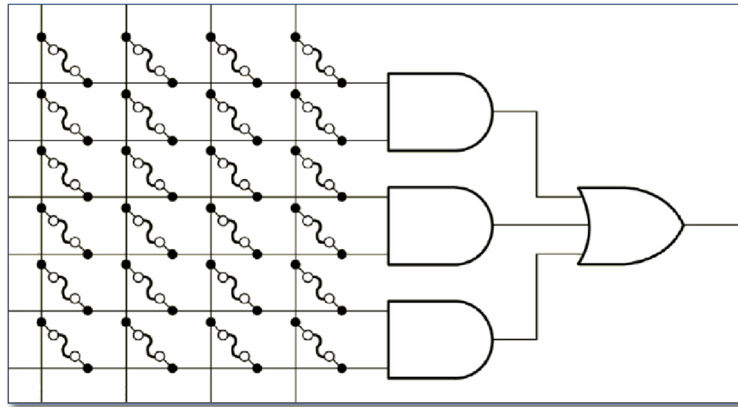


Figure 1.42: Unprogrammed PAL.

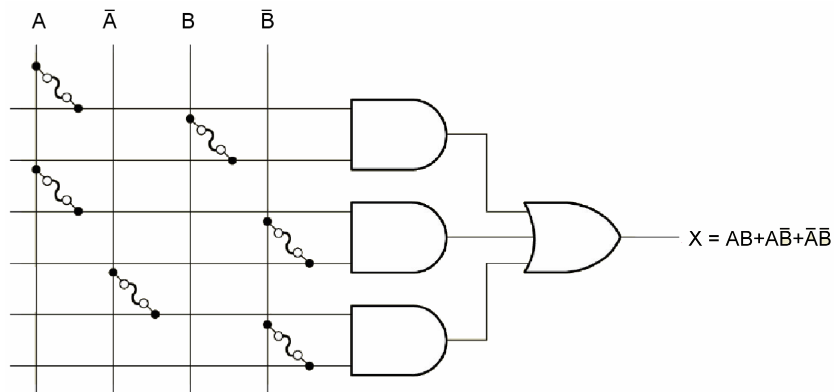


Figure 1.43: An example of a programmed PAL.

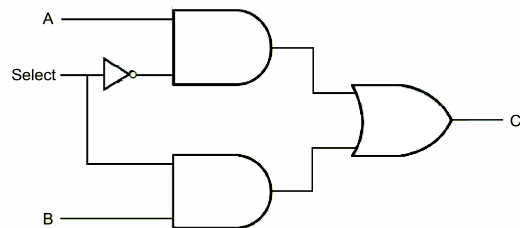


Figure 1.44: An example of a multiplexer based on a PLD

1.13 Mixed-Signal Processing

The earliest embedded systems were for the most part dealing with analog signals in the form of voltages and relied on analog-to-digital and digital-to analog converters to interface to the real world. In recent years

1.14 PSoC - Programmable System on Chip

Cypress Semiconductor's family of PSoC1/3/5 devices employ a highly configurable system-on-chip architecture for embedded control design, providing a flash-based equivalent of a field-programmable ASIC without imposing lead-time or NRE[74] penalties. PSoC devices integrate configurable analog and digital circuits, controlled by an on-chip microcontroller, providing both enhanced design revision capability and component count savings. A single PSoC device can provide as many as 100 peripheral functions , while requiring minimal board space and power consumption, improving system quality and reducing system cost.

All PSoC devices are also dynamically reconfigurable, so that their internal resources can be to "morphed" on-the-fly, utilizing fewer components to perform a given task. This text focuses on two particular PSoC-based architectures which provide excellent performance and unmatched time-to-market, integration, and flexibility across 8-, 16-, and 32-bit applications. These programmable, analog and digital, embedded design platforms are powered by an innovative development environment called the PSoC Creator Integrated Development Environment (PSoC Creator IDE), which has a unique, schematic-based, design-capture functionality and fully tested, libraries of pre-packaged analog and digital peripherals that are easily customizable by the use of user-intuitive wizards and APIs. PSoC Creator enables designers to develop new designs tin a highly intuitive manner that strongly reflects the manner in which designers think about their designs and dramatically shortens time-to-market.

The programmable analog and digital peripherals in PSoC3/5, high performance 8-bit and 32-bit MCU sub-systems and capabilities such as motor control, intelligent power supply/battery management and support for human interfaces with CapSense touch sensing, LCD segment displays, graphics controls, audio/voice processing, communication protocols, and much more, make it possible for designers to address a wide variety of mixed-signal, embedded applications, including all phases of the industrial, medical, automotive, communications and consumer markets.

The PSoC3/5 architectures include high-precision, programmable analog resources that can be configured as ADCs, DACs, TIAs, Mixers, PGAs, OpAmps, etc. and enhanced programmable logic-based digital resources that can be configured as 8-, 16-, 24- and 32-bit timers, counters, and PWMs and advanced digital peripherals such as Cyclic Redundancy Check (CRC), Pseudo Random Sequence (PRS) generators, and quadrature decoders. These resources allow designers to customize PSoC3/5's general purpose PLD-based logic. These architectures also support a wide range of communications interfaces, including Full-Speed USB, I2C, SPI, UART, CAN, LIN, and I2S.

The new PSoC3/5 architectures are powered by high performance, industry-standard processors. The PSoC3 architecture is based on a new, high-performance 8-bit 8051 processor provides up to 33 MIPS. The PSoC5 architecture includes a powerful 32-bit ARM Cortex-M3 processor and is capable of carrying out 100 MIPS. Both architectures meet the demands of extremely low power applications based on their availability to operate over a voltage range from 0.5 to 5 volts and hibernate current as low as 200nA. They provide a seamless, programmable design platform

[74] Non-Recurring Engineering.

from 8- to 32-bit architectures with pin and API compatibility between PSoC3 and PSoC5, along with programmable routing, allowing any signal, whether analog or digital, to be routed to any general-purpose I/O to ease circuit board layout. This capability includes the ability to route LCD Segment Display and CapSense signals to any GPIO pin.

PSoC3/5 architectures serve as scalable platforms with the computing power of high-performance MCUs, the precision of stand-alone analog devices and the flexibility of PLDs, all within the scope of PSoC3/5's powerful, easy-to-use design environment. Thus designers of 8-, 16- and 32-bit applications able to fully exploit the inherent flexibility and integration of PSoC3/5's true system-level programmability and extend the concept of programmability beyond instructions for the processor to configuring peripherals and customization of digital functions.

PSoC3/5's internal architecture, cf. Figure 1.45,. consists of 14 configurable digital modules (PWM, UART, A/D, etc.), and 10 analog modules (A/D, filter, etc.) and a CPU core (either 8051 or ARM Cortex-M3) with interrupt controller, internal oscillator, digital clocks, Flash, SRAM, I2C and USB controllers, switch mode pump, decimator, MAC. All of these resources are supported by an extensive programmable, interconnect and routing facility that provides virtually unlimited configurations and interconnections of digital and analog modules and resources. External interfacing is provided by 8 ports (0-7).

Key Features of the PSoC3/5 Architectures

- A programmable precision analog sub-system that provides up to 20-bit resolution for the integral Delta-Sigma ADC, sample rates up to 1 msps for the 12-bit SAR ADC, a reference voltage accurate to +/- 0.1 voltage range, up to four 8-bit, 8 Msps DACs;, 1-50x PGA, general purpose Op-amps with 25mA drive capability, up to four comparators with 30 ns response time, DSP-like digital filter implementation for instrumentation and medical signal processing, a large library of pre-characterized analog peripherals in PSoC Creator Software and CapSense functionality for all devices.
- A programmable, high-performance, digital array of Universal Digital Blocks (UDBs) each consisting of a combination of uncommitted logic (PLD), structured logic (datapath), and flexible routing to other UDBs, I/O or peripherals, a large library of pre-characterized digital peripherals in PSoC Creator Softwar, e.g., 8-, 16-, 24- and 32-bit timers, counters and PWMs.
- A customizable digital system is made possible by the full featured general purpose PLD-based logic provided on-chip.
- PSoC3/5 support high-speed connectivity support for full Speed USB, I2C, SPI, UART, CAN, LIN and I2S.
- A high-performance CPU sub-systems based on either PSoC3's 8-bit 8051 core with 33 MIPS performance (PSoC3) or PSoC5's 32-bit ARM Cortex-M3 core with 100 MIPS performance, 24-channel, multi-layer, Direct Memory Access (DMA) with simultaneous access to SRAM and CP,U on-chip debug and trace functionality with JTAG and Serial Wire Debug (SWD)and the availability of a wide variety of industry-standard compilers and real time operating systems.
- PSoC3/5's low power operation modes provide an an operating range from 0.5-5.0 volts with no degradation in analog performance. PSoC3/5's active power consumption is 1.2mA at 6 MHz for PSoC3 and 2mA at 6 MHz for PSoC5. Sleep-mode power consumption for PSoC3 is 1μA for PSoC3 and 2μA for PSoC5. Hibernate-mode power consumption for PSoC3 is 200nA and 300nA for PSoC5.

- PSoC3/5 provide programmable, feature-rich I/O & clocking by providing interconnection of any pin to any analog or digital peripheral, LCD segment display on any pin with up to 16-commons/736 segments, CapSense on any pin for replacing mechanical buttons and sliders. PSoC3/5 also support .2-5.5V I/O interface voltages, up to 4 domains for easy interface with systems running at different voltage domains, and a 1-66 MHz, internal, $+/-1\%$ oscillator with PLL over the full temperature and voltage range.

Summary: This chapter has provided a brief summary of the history of embedded systems, microprocessors and microcontrollers Also presented were basic concepts of programmable logic devices, overviews of the Intel 8048 and 8051 microcontrollers, brief descriptions of some of the more popular and currently available microcontrollers that are in widespread use and introductions to a number of subjects related to microcontrollers and embedded systems, e.g., types of feedback systems employed in embedded systems, microcontroller subsystems, microprocessor/microcontroller memory types, embedded system performance criteria, interrupts, introductory sampling topics, sensors and sensor types, strain gauge/thermocouple/thermistor sensing and measurement techniques, software development for embedded systems, embedded systems communications, brief overviews of PSoC3, PSoc5 and the PSoC3/5 development environment, PSoC Creator, etc.

In the chapters that follow, more detailed discussions and illustrative software and hardware examples are provided that are related to the topics in this chapter, as well as others, with particular emphasis on the PSoC3/5 family of programmable systems on a chip.

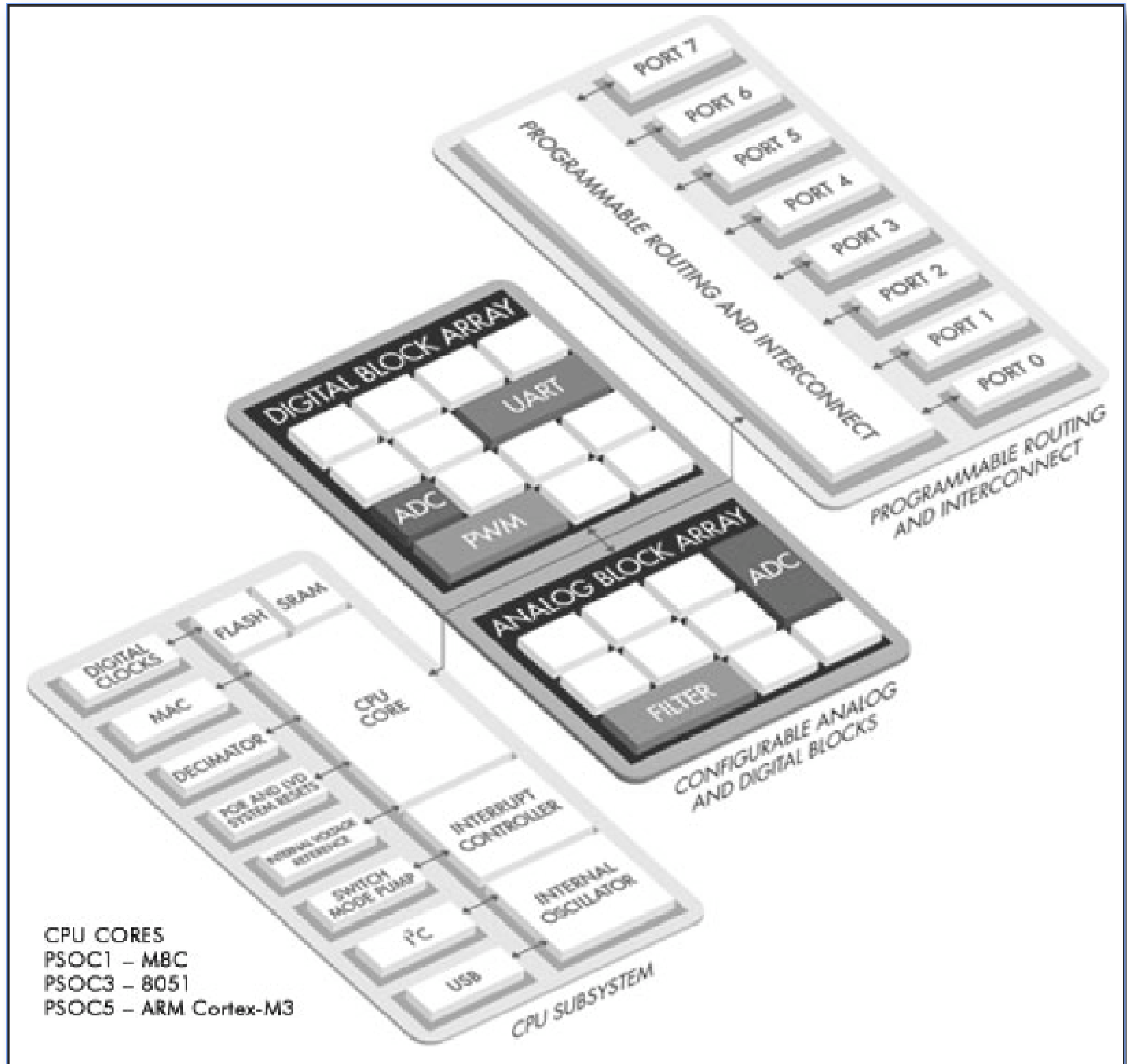


Figure 1.45: PSoC1/PSoC2/PSoC3 architectures.

Appendix A

Mnemonics

ADC - Analog to Digital Converter (also A/D)
ALU - Arithmetic Logic Unit
AMD - Analog Modulator
ARM - Advanced RISC machine Core
CAN - Controller Area Network
CDAC - Current Digital To Analog Converter
CapSense - Capscative3 Sensing
CLK - Clock
CLR - Clear
CMOS - Complementary Metal-Oxide Semiconductor
CPU - Central Processing Unit
DAC - Digital to Analog Converter (also D/A)
DDA - Diffrential digital analyzer
DEC - Decimator
DFB - Digital Filter Block
DOC - Debug On Chip
DUT - Device Under Test
DSM - Delta-Sigma Modulator
EPROM - Electrically Programmable Read-Only Memory
EEPROM - Electrically Erasable/Programmable Read-Only Memory
FIFO - First In First Out **GP** - General Purpose
GPIO - General purpose input/output
I2C - Inter-IntegratedCircuit
ILO - Internal Local Oscillator
IMO - Internal Main Oscillator
IPGA - Inverting Programmable Gain Amplifier
IAV - Interrupt Address Vector
INT - Interrupt
LCD - Liquid Crystal Display
LED - Light Emitting Diode
LUT - Look Up Table
MIPS - Millions of INstructins Per Second **MMIO** - Memory Mapped Input Output
MUX - Multiplexer
NMI - Non Maskable Interrupt
NOP - No Operation
PGA - Programmable Gain Amplifier

PLL - Phase-Locked Loop
PMIO - Port Mapped Input Output
PRT - Port (GPIO/SIO)
PWM - Pulse Width Modulator
RISC - Reduced Instruction Set Computer **SPISTK** - SPI Stack
SIO - Serial Input Output
SWD - Serial Wire Debugging
TMR - Timer
TST - Test
UART - Universal Asynchronous Receiver Transmitter
VDAC - Voltage Digital to Analog Converter
VLT - Low votlage reference
WDT - Watch Dog Timer

Appendix B

Definitions

Accumulator - In a CPU, a register in which intermediate results are stored. Without an accumulator it would be necessary to write the result of each calculation (addition, subtraction, shift, and so on.) to main memory and read them back. Access to main memory is slower than access to the accumulator, which usually has direct paths to and from the arithmetic and logic unit (ALU).
Active High - A logic signal having its asserted state as the logic 1 state. A logic signal having the logic 1 state as the higher voltage of the two states.
Active Low -

1. A logic signal having its asserted state as the logic 0 state.
2. A logic signal having its logic 1 state as the lower voltage of the two states: inverted logic.

Address - The label or number identifying the memory location (RAM, ROM, or register) where a unit of information is stored.
Algorithm - A procedure for solving a mathematical problem in a finite number of steps that frequently involve repetition of an operation.
Ambient Temperature - The temperature of the air in a designated area, particularly the area surrounding the PSoC device.
Analog - (See analog signals).
Analog Blocks - The basic programmable OpAmp circuits, i.e., SC (switched capacitor) and analog blocks CT (continuous time) blocks. These blocks can be interconnected to provide ADCs, DACs, multi-pole filters, gain stages, etc.
Analog Output - An output that is capable of driving any voltage between the supply rails, instead of analog output just a logic 1 or logic 0.
Analog Signal - A signal represented in a continuous form with respect to continuous times, as analog signals contrasted with a digital signal represented in a discrete (discontinuous) form in a sequence of time.
Analog-to-Digital Converter (ADC) - A device that changes an analog signal to a digital signal of corresponding magnitude. Typically, an ADC converts a voltage to a digital number. The digital-to-analog (DAC) converter performs the inverse operation.
AND - See Boolean Algebra.
Application Program Interface (API) - A series of software routines that comprise an interface between a computer application and lower-level services and functions (for example, user modules and programming interface libraries). APIs serve as building blocks for programmers that create software applications.
Array - Also referred to as a vector or list, is one of the simplest data structures in computer programming. Arrays hold a fixed number of equally-sized data elements, generally of the same data type. Individual elements are accessed by index using a consecutive range of integers, as

opposed to an associative array. Most high level programming languages have arrays as a built-in data type. Some arrays are multi-dimensional, meaning they are indexed by a fixed number of integers; for example, by a group of two integers. One- and two-dimensional arrays are the most common. Also, an array can be a group of capacitors or resistors connected in some common form.

Assembly - A symbolic representation of the machine language of a specific processor. Assembly language is converted to machine code by an assembler. Usually, each line of f assembly code produces one machine instruction, though the use of macros is common. Assembly languages are considered low level languages; where as C is considered a high level language.

Asynchronous - a signal whose data is acknowledged or acted upon immediately, irrespective of any clock signal.

Attenuation - The decrease in intensity of a signal as a result of absorption of energy and of scattering out of the path to the detector, but not including the reduction due to geometric spreading. Attenuation is usually expressed in dB.

Bandgap Reference - A stable voltage reference design that matches the positive temperature coefficient of VT with the negative temperature coefficient of VBE, to produce a zero temperature coefficient (ideally) reference.

Bandwidth -

1. The frequency range of a message or information processing system measured in Hertz.
2. The width of the spectral region over which an amplifier (or absorber) has substantial gain (or loss); it is sometimes represented more specifically as, for example, full width at half maximum.

Bias -

1. A systematic deviation of a value from a reference value.
2. The amount by which the average of a set of values departs from a reference value.
3. The electrical, mechanical, magnetic, or other force (field) applied to a device to establish a reference level to operate the device.

Bias Current - The constant low level DC current that is used to produce a stable operation in bias current amplifiers. This current can sometimes be changed to alter the bandwidth of an amplifier.

Binary - The name for the base 2 numbering system. The most common numbering system is the base 10 numbering system. The base of a numbering system indicates the number of values that may exist for a particular positioning within a number for that system. For example, in base 2, binary, each position may have one of two values (0 or 1). In the base 10, decimal, numbering system, each position may have one of ten values (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9).

Bit - A single digit of a binary number. Therefore, a bit may only have a value of "0" or bit "1". A group of 8 bits is called a byte. Because the PSoC's M8CP is an 8-bit microcontroller, PSoC's native data chunk size is a byte.

Bit Rate (BR) - The number of bits occurring per unit of time in a bit stream, usually expressed in bit rate (BR) bits per second (bps).

Block -

1. A functional unit that performs a single function, e.g., an oscillator.
2. A functional unit that may be configured to perform one of several functions, such as a digital or analog PSoC block.

Boolean Algebra - In mathematics and computer science, Boolean algebras or Boolean lattices, are algebraic structures which "capture the essence" of the logical operations AND, OR and NOT, as well as, the set theoretic operations, i.e., union, intersection, and complement. Boolean algebra also defines a set of theorems that describe how Boolean equations can be manipulated. For example, these theorems are used to simplify Boolean equations, which will reduce the number of logic elements needed to implement the equation. The operators of Boolean algebra may

be represented in various ways. Often they are simply written as AND, OR, and NOT. In describing circuits, NAND (NOT AND), NOR (NOT OR), XNOR (exclusive NOT OR), and XOR (exclusive OR) may also be used. Mathematicians often use + (for example, $A + B$) for OR and for AND (for example, $A * B$) (since in some ways those operations are analogous to addition and multiplication in other algebraic structures) and represent NOT by a line drawn above the expression being negated (for example, $A, A, !A$).
Break-Before-Make - The elements involved go through a disconnected state entering (break) before the new connected state (make).
Buffer -

1. A storage area for data that is used to compensate for a speed difference, when transferring data from one device to another. Usually refers to an area reserved for I/O operations, into which data is read, or from which data is writ- ten.

2. A portion of memory set aside to store data, often before it is sent to an external device or as it is received from an external device.

3. An amplifier used to lower the output impedance of a system.

Bus -

1. A named connection of nets. Bundling nets together in a bus makes it easier to route nets with similar routing patterns.

2. A set of signals performing a common function and carrying similar data. Typically represented using vector notation; for example, address[7:0]. 3. One or more conductors that serve as a common connection for a group of related devices.

Byte - A digital storage unit consisting of 8 bits.
C - A high level programming language.
Capacitance - A measure of the ability of two adjacent conductors, separated by an insulator, to hold a charge when a voltage differential is applied between them. Capacitance is measured in units of Farads.
Capture - To extract information automatically through the use of software or hardware, as opposed to hand-entering of data into a computer file.
Chaining - Connecting two or more 8-bit digital blocks to form 16-, 24-, and even 32-bit functions. Chaining allows certain signals such as Compare, Carry, Enable, Capture,and Gate to be produced from one block to another.
Checksum - The checksum of a set of data is generated by adding the value of each data word to a sum. The actual checksum can simply be the result sum or a value that must be added to the sum to generate a pre-determined value.
Clear - To force a bit/register to a value of logic "0″.
Clock - The device that generates a periodic signal with a fixed frequency and duty cycle. A clock is sometimes used to synchronize different logic blocks.
Clock Generator - A circuit that is used to generate a clock signal.
CMOS - The logic gates constructed using CMOS transistors connected in a CMOS complementary manner. CMOS is an acronym for complementary metal-oxide semiconductor.
Comparator - An electronic circuit that produces an output voltage or current whenever two input levels simultaneously satisfy predetermined amplitude requirements.
Compiler - A program that translates a high level language, such as C, into machine language.
Configuration - In a computer system, an arrangement of functional units according to their configuration nature, number, and chief characteristics. Configuration pertains to hardware, software, firmware, and documentation. The configuration will affect system performance.
Configuration Space - the PSoC register space accessed when the XIO bit, in the CPU_F configuration space register, is set to "1".
CPLD - Complex PLD consisting of multiple SPLDs. FPGA - Field-Programmable Gate Array a field programmable device capable of very complex logic functionality. Whereas CPLDs feature

logic resources with a wide number of inputs (AND planes), FPGAs offer more narrow logic resources. FPGAs also offer a higher ratio of flip-flops to logic resources than do CPLDs.

Crowbar - A type of over-voltage protection that rapidly places a low resistance shunt (typically an SCR) from the signal to one of the power supply rails, when the output voltage exceeds a predetermined value.

Crystal Oscillator - An oscillator in which the frequency is controlled by a piezoelectric crystal. Typically a piezoelectric crystal is less sensitive to ambient temperature than other circuit components.

Cyclic Redundancy Check (CRC) - A calculation used to detect errors in data communications, typically performed cyclic redundancy using a linear feedback shift register. Similar calculations may be used for a variety check (CRC) of other purposes such as data compression.

Data Bus - A bi-directional set of signals used by a computer to convey information from a data bus memory location to the central processing unit and vice versa. More generally, a set of signals used to convey data between digital functions.

Data Stream - A sequence of digitally encoded signals used to represent information in transmission.

Data Transmission - The sending of data from one place to another by means of signals over a channel.

Debugger - A hardware and software system that allows the user to analyze the operation of the system under development. A debugger usually allows the developer to step through the firmware one step at a time, set break points, and analyze memory.

Dead Band - A period of time when neither of two or more signals are in their active state or in dead band transition.

Decimal - A base-10 numbering system, which uses the symbols $0, 1, 2, 3, 4, 5, 6, 7, 8$ and 9 decimal (called digits) together with the decimal point and the sign symbols $+$ (plus) and $-$ (minus) to represent numbers.

Default Value - Pertaining to the pre-defined initial, original, or specific setting, condition, value, or default value action a system will assume, use, or take in the absence of instructions from the user.

Device - The device referred to in this manual is the PSoC chip, unless otherwise specified.

Die - An unpackaged integrated circuit (IC), normally cut from a wafer.

Digital - A signal or function, the amplitude of which is characterized by one of two discrete digital values: "0" or "1".

Digital Blocks - The 8-bit logic blocks that can act as a counter, timer, serial receiver, serial transmitter, CRC generator, pseudo-random number generator, or SPI.

Digital Logic - A methodology for dealing with expressions containing two-state variables that describe the behavior of a circuit or system.

Digital-to-Analog (DAC) - A device that changes a digital signal to an analog signal of corresponding magnitude. The analog-to-digital (ADC) converter performs the reverse operation.

Direct Access - The capability to obtain data from a storage device, or to enter data into a storage device, in a sequence independent of their relative positions by means of addresses that indicate the physical location of the data.

Duty Cycle - The relationship of a clock period high time to its low time, expressed as a percent.

Emulator- Duplicates (provides an emulation of) the functions of one system with a different system, so that the second system appears to behave like the first system.

External Reset (XRES) - An active high signal that is driven into the PSoC device. It causes all operation of the CPU and blocks to stop and return to a pre-defined state.

Falling Edge - A transition from a logic 1 to a logic 0. Also known as a negative edge.

Feedback - The return of a portion of the output, or processed portion of the output, of a (usually active) device to the input.

Filter- A device or process by which certain frequency components of a signal are attenuated.

Firmware The software that is embedded in a hardware device and executed by the CPU.

Flag - The software may be executed by the end user, but it may not be modified. Any of various types of indicators used for identification of a condition or event (for example, a character that signals the termination of a transmission).

Flash - An electrically programmable and erasable, volatile technology that provides users with the programmability and data storage of EPROMs, plus in-system erasability. Non-volatile means that the data is retained when power is off.

Flash Bank - A group of Flash ROM blocks where Flash block numbers always begin with "0" in an individual Flash bank. A Flash bank also has its own block level protection information.

Flash Block - The smallest amount of Flash ROM space that may be programmed at one time and the smallest amount of Flash space that may be protected. A Flash block holds 64 bytes.

Flip-Flop A device having two stable states and two input terminals (or types of input signals) each of which corresponds with one of the two states. The circuit remains in either state until it is made to change to the other state by application of the corresponding signal.

Frequency - The number of cycles or events per unit of time, for a periodic function.

Gain - The ratio of output current, voltage, or power to input current, voltage, or power, respectively. Gain is usually expressed in dB.

Gate - 1. A device having one output channel and one or more input channels, such that the output channel state is completely determined by the input channel states, except during switching transients.

2. One of many types of combinational logic elements having at least two inputs (for example, AND, OR, NAND, and NOR (Boolean Algebra)).

Ground -

1. The electrical neutral line having the same potential as the surrounding earth.
2. The negative side of DC power supply.
3. The reference point for an electrical system.
4. The conducting paths between an electric circuit or equipment and the earth, or some conducting body serving in place of the earth.

Hardware - A comprehensive term for all of the physical parts of a computer or embedded system, as distinguished from the data it contains or operates on, and the software that provides instructions for the hardware to accomplish tasks.

Hardware Reset - A reset that is caused by a circuit, such as a POR, watchdog reset, or external reset. A hardware reset restores the state of the device as it was when it was first powered up. Therefore, all registers are set to the POR value as indicated in register tables throughout this document.

Harvard Architecture: separate memory areas are used for program instructions and data. Two or more internal data buses are employed to provide contemporaneous access data and instructions. The CPU fetches program instructions are fetched by the CPU on the program memory bus.

HCPLD - high-capacity PLD, e.g., FPGAs and CPLDs. Field-Programmable Device (FPD) a type of programmable integrated circuit used for implementing digital hardware, where the chip can be configured by the end user. Programming of such a device often involves placing the chip into a special programming unit, but some chips can also be configured in-system. Another name for FPDs is programmable logic devices (PLDs); although PLDs encompass the same types of chips as FPDs, we prefer the term FPD because historically the word PLD has referred to relatively simple types of devices.

Hexadecimal - A base 16 numeral system (often abbreviated and called hex), usually written using the symbols 0-9 and A-F. It is a useful system in computers because there is an easy

mapping from four bits to a single hex digit. Thus, one can represent every byte as two consecutive hexadecimal digits. Compare the binary, hex, and decimal representations:

bin	hex	dec a
0000	0x0	0
0001	0x1	1
0010	0x2	2
...	...	...
1001	0x9	9
1010	0xA	10
1011	0xB	11
...	...	...
1111	0xF	15

So the decimal numeral 79 whose binary representation is $01001111b$ can be written as $4Fh$ in hexadecimal ($0x4F$).
High Time - The amount of time the signal has a value of "1" in one period, for a periodic digital high time signal.
I2C - A two-wire serial computer bus by Phillips Semiconductors. I2C is an Inter-Integrated Circuit. It is used to connect low-speed peripherals in an embedded system. The original system was created in the early 1980s as a battery control interface, but it was later used as a simple internal bus system for building control electronics. I2C uses only two bi-directional pins, clock and data, both running at +5V and pulled high with resistors. The bus operates at 100 kbits/second in standard mode and 400 kbits/second in fast mode. I2C is a trademark of the Philips Semiconductors.
ICE - The in-circuit emulator that allows users to test the project in a hardware environment, while viewing the debugging device activity in a software environment (PSoC Designer).
Idle State - A condition that exists whenever user messages are not being transmitted, but the idle state service is immediately available for use.
Impedance -
1. The resistance to the flow of current caused by resistive, capacitive, or inductive devices in a circuit.
2. The total passive opposition offered to the flow of electric current. Note the impedance is determined by the particular combination of resistance, inductive reactance, and capacitive reactance in a given circuit.

Input - A point that accepts data, in a device, process, or channel.
Input/Output - A device that introduces data into, or extracts data from, a system.
Instruction - An expression that specifies one operation and identifies its operands, if any, in a instruction programming language such as C or assembly.
Integrated Circuit (IC) - A device in which components such as resistors, capacitors, diodes, and transistors are formed on the surface of a single piece of semiconductor.
Interface - The means by which two systems or devices are connected and interact with each interface other.
Interrupt - A suspension of a process, such as the execution of a computer program, caused interrupt by an event external to that process, and performed in such a way that the process can be resumed.
Interrupt Service Routine - A block of code that normal code execution is diverted to when the M8CP receives a hardware interrupt. Many interrupt sources may each exist with its own priority tine (ISR) and individual ISR code block. Each ISR code block ends with the RETI instruction, returning the device to the point in the program where it left normal program execution.
Jitter -

1. A misplacement of the timing of a transition from its ideal position. A typical jitter form of corruption that occurs on serial data streams. 2. The abrupt and unwanted variations of one or more signal characteristics, such as the interval between successive pulses, the amplitude of successive cycles, or the frequency or phase of successive cycles.

Keeper - A circuit that holds a signal to the last driven value, even when the signal becomes un-driven.

Latency - The time or delay that it takes for a signal to pass through a given circuit or network.

Least Significant Bit (LSb) - The binary digit, or bit, in a binary number that represents the least significant least significant bit value (typically the right-hand bit). The bit versus byte distinction is made by using (LSb) a lower case for bit in LSb.

Least Significant Bit (LSB) - The byte in a multi-byte word that represents the least significant values (typically least significant byte the right-hand byte). The byte versus bit distinction is made by using an upper (LSB) case for byte in LSB.

Lttle-endian - the lower-order byte is stored at the lower address and the higher-order byte is stored at the upper address. (cf. Big-endian)

Linear Feedback Shift Register (LFSR) - A shift register whose data input is generated as an XOR of two or more elements in the register chain.

Load - The electrical demand of a process expressed as power (watts), current (amps), or resistance (Ohms).

Logic Function - A mathematical function that performs a digital operation on digital data and logic function returns a digital value.

Logic Block a relatively small circuit block that is replicated in an array in an FPD. When a circuit is implemented in an FPD, it is first decomposed into smaller sub-circuits that can each be mapped into a logic block. The term logic block is mostly used in the context of FPGAs, but it could also refer to a block of circuitry in a CPLD.

Logic Capacity - amount of digital logic that can be mapped into a single FPD. This is usually measured in units of equivalent number of gates in a traditional gate array. In other words, the capacity of an FPD is measured by the size of gate array that it is comparable to. In simpler terms, logic capacity can be thought of as number of 2-input NAND gates.

Look-Up Table (LUT) - A logic block that implements several logic functions. The logic function is selected look-up table (LUT) by means of select lines and is applied to the inputs of the block. For example: A 2 input LUT with 4 select lines can be used to perform any one of 16 logic functions on the two inputs resulting in a single logic output. The LUT is a combinational device; therefore, the input/output relationship is continuous, that is, not sampled.

Low Time - The amount of time the signal has a given value in one period, for a periodic digital signal.

Low Voltage Detect (LVD) - A circuit that senses Vdd and provides an interrupt to the system when Vdd falls below a selected threshold.

M8CP - An 8-bit, Harvard Architecture microprocessor. The microprocessor coordinates all activity inside a PSoC by interfacing to the Flash, SRAM, and register space.

Macro - A programming language macro is an abstraction, whereby a certain textual pattern is replaced according to a defined set of rules. The interpreter or compiler automatically replaces the macro instance with the macro contents when an instance of the macro is encountered. Therefore, if a macro is used 5 times and the macro definition required 10 bytes of code space, 50 bytes of code space will be needed in total.

Mask -

1. To obscure, hide, or otherwise prevent information from being derived from a mask signal. It is usually the result of interaction with another signal, such as noise, static, jamming, or other forms of interference.

2. A pattern of bits that can be used to retain or suppress segments of another pattern of bits, in computing and data processing systems.

Master Device - A device that controls the timing for data exchanges between two devices. Or when devices are cascaded in width, the master device is the one that controls the timing for data exchanges between the cascaded devices and an external inter- face. The controlled device is called the slave device.

Microcontroller - An integrated circuit chip that is designed primarily for control systems and products. In addition to a CPU, a microcontroller typically includes memory, timing circuits, and IO circuitry. The reason for this is to permit the realization of a controller with a minimal quantity of chips, thus achieving maximal possible miniaturization. This in turn, will reduce the volume and the cost of the controller. The microcontroller is normally not used for general-purpose computation as is a microprocessor.

Mixed Signal - The reference to a circuit containing both analog and digital techniques and components.

Mnemonic - A tool intended to assist the memory. Mnemonics rely on not only repetition to remember facts, but also on creating associations between easy-to-remember constructs and lists of data. A two to four character string representing a microprocessor instruction.

Mode - A distinct method of operation for software or hardware. For example, the Digital modulation PSoC block may be in either counter mode or timer mode. A range of techniques for encoding information on a carrier signal, typically a sine wave signal. A device that performs modulation is known as a modulator.

Modulator - A device that imposes a signal on a carrier.

MOS - An acronym for metal-oxide semiconductor.

Most Significant bit (MSb) - The binary digit, or bit, in a binary number that represents the most significant value (typically the left-hand bit). The bit versus byte distinction is made by using a lower case for bit in MSb.

Most Significant Byte (MSB) - The byte in a multi-byte word that represents the most significant values (typically most significant byte the left-hand byte). The byte versus bit distinction is made by using an upper case for byte in MSB.

Multiplexer (Mux) -

1. A logic function that uses a binary value, or address, to select between a number of inputs and conveys the data from the selected input to the output.

2. A technique which allows different input (or output) signals to use the same lines at different times, controlled by an external signal. Multiplexing is used to save on wiring and IO ports.

NAND - See Boolean Algebra.

Negative Edge - A transition from a logic 1 to a logic 0. Also known as a falling edge.

Net - The routing between devices.

Net - A signal that is routed throughout the microcontroller and is accessible by many blocks or systems.

Nibble - A group of four bits, which is one-half of a byte.

Noise -

1. A disturbance that affects a signal and that may distort the information carried by the signal.

2. The random variations of one or more characteristics of any entity such as volt- age, current, or data.

NOR - See Boolean Algebra.

NOT - See Boolean Algebra.

OR - See Boolean Algebra.

Oscillator - A circuit that may be crystal controlled and is used to generate a clock frequency.

Output - The electrical signal or signals which are produced by an analog or digital block.
Parallel - The means of communication in which digital data is sent multiple bits at a time, with each simultaneous bit being sent over a separate line.
Parameter - Characteristics for a given block that have either been characterized or may be defined by the designer.
Parameter Block - A location in memory where parameters for the SSC instruction are placed prior to execution.
Parity - A technique for testing transmitting data. Typically, a binary digit is added to the data to make the sum of all the digits of the binary data either always even (even parity) or always odd (odd parity).
Path -

1. The logical sequence of instructions executed by a computer.
2. The flow of an electrical signal through a circuit.

Pending Interrupts - An interrupt that has been triggered but has not been serviced, either because the processor is busy servicing another interrupt or global interrupts are disabled.
Phase - The relationship between two signals, usually the same frequency, that determines the delay between them. This delay between signals is either measured by time or angle (degrees).
Phase-Locked Loop (PLL) - An electronic circuit that controls an oscillator so that it maintains a constant phase angle relative to a reference signal.
Pin - A terminal on a hardware component. Also called lead.
Pinouts - The pin number assignment: the relation between the logical inputs and outputs of the PSoC device and their physical counterparts in the printed circuit board (PCB) package. Pinouts will involve pin numbers as a link between schematic and PCB design (both being computer generated files) and may also involve pin names.
Port - A group of input/output pins, usually eight.
Positive Edge - A transition from a logic 0 to a logic 1. Also known as a rising edge.
Posted Interrupts - An interrupt that has been detected by the hardware but may or may not be enabled by its mask bit. Posted interrupts that are not masked become pending interrupts.
Power On Reset (POR) - A circuit that forces the PSoC device to reset when the voltage is below a pre-set level. This is one type of hardware reset.
Program Counter - The instruction pointer (also called the program counter) is a register in a computer processor that indicates where in memory the CPU is executing instructions. Depending on the details of the particular machine, it holds either the address of the instruction being executed, or the address of the next instruction to be executed.
Protocol - A set of rules. Particularly the rules that govern networked communications.
Programmable Array Logic (PAL) - a small FPD that has a programmable AND-plane followed by a fixed OR-plane.
Programmable Logic Array (PLA) - a small FPD consisting of an AND-plane and an OR-plane, which are programmable.
Programmable Switch a user-programmable switch that can connect a logic element to an interconnect wire, or one interconnect wire to another.
PSoC - Cypress MicroSystems' Programmable System-on-Chip (PSoC) mixed signal array. PSoC and Programmable System-on-Chip are trademarks of Cypress MicroSystems, Inc.
PSoC Blocks - See analog blocks and digital blocks.
PSoC Designer - The software for Cypress MicroSystems Programmable System-on-Chip technology.
Pulse - A rapid change in some characteristic of a signal (for example, phase or frequency), from a baseline value to a higher or lower value, followed by a rapid return to the baseline value.
Pulse Width Modulator (PWM) - An output in the form of duty cycle which varies as a function of the applied measurand.

RAM - An acronym for Random Access Memory. A data-storage device from which data can be read out and new data can be written in.
Register - A storage device with a specific capacity, such as a bit or byte.
Reset - A means of bringing a system back to a know state. See hardware reset and software reset.
Resistance - The resistance to the flow of electric current measured in Ohms for a conductor.
Revision ID - A unique identifier of the PSoC device.
Ripple Divider - An asynchronous ripple counter constructed of flip-flops. The clock signal is fed to the first stage of the counter. An n-bit binary counter consisting of n flip-flops that can count in binary from $0 to 2n - 1$.
Rising Edge - See positive edge.
ROM - An acronym for read only memory. A data-storage device from which data can be read, but new data cannot be written in.
Routine - A block of code, called by another block of code, that may have some general or frequent use.
Routing - Physically connecting objects in a design according to design rules set in the reference library.
RPM - revolutions per minute.
Runt Pulses - In digital circuits, narrow pulses that, due to non-zero rise and fall times of the signal, do not reach a valid high or low level. For example, a runt pulse may occur when switching between asynchronous clocks or as the result of a race condition in which a signal takes two separate paths through a circuit. These race conditions may have different delays and are then recombined to form a glitch or when the output of a flip-flop becomes metastable.
Sampling - The process of converting an analog signal into a series of digital values or reversed.
Schematic - A diagram, drawing, or sketch that details the elements of a system, such as the elements of an electrical circuit or the elements of a logic diagram for a computer.
Seed Value - An initial value loaded into a linear feedback shift register or random number generator.
Serial -

1. Pertaining to a process in which all events occur one after the other.
2. Pertaining to the sequential or consecutive occurrence of two or more related activities in a single device or channel.

Set - To force a bit/register to a value of logic 1.
Settling Time - The time it takes for an output signal or value to stabilize after the input has changed from one value to another.
Shift - The movement of each bit in a word one position to either the left or right. For example, if the hex value 0x24 is shifted one place to the left, it becomes 0x48. If the hex value 0x24 is shifted one place to the right, it becomes 0x12.
Shift Register - A memory storage device that sequentially shifts a word either left or right to out- put a stream of serial data.
Sign Bit - The most significant binary digit, or bit, of a signed binary number. If set to a logic 1, this bit represents a negative quantity.
Signal- A detectable transmitted energy that can be used to carry information. As applied to electronics, any transmitted electrical impulse.
Silicon ID - A unique identifier of the PSoC silicon.
Skew - The difference in arrival time of bits transmitted at the same time, in parallel transmission.
Slave Device - A device that allows another device to control the timing for data exchanges between two devices. Or when devices are cascaded in width, the slave device is the one that allows another device to control the timing of data exchanges between the cascaded devices and an external interface. The controlling device is called the master device.

Software - A set of computer programs, procedures, and associated documentation concerned with the operation of a data processing system (for example, compilers, library routines, manuals, and circuit diagrams). Software is often written first as source code, and then converted to a binary format that is specific to the device on which the code will be executed.

Software Reset - A partial reset executed by software to bring part of the system back to a known state. A software reset will restore the M8CP to a know state but not PSoC blocks, systems, peripherals, or registers. For a software reset, the CPU registers (CPU_A, CPU_F, CPU_PC, CPU_SP, and CPU_X) are set to $0x00$. Therefore, code execution will begin at Flash address $0x0000$.

SPLD - Simple PLD, typically a PAL or PLA

Logic Density logic per unit area in an FPD.

SRAM - An acronym for static random access memory. A memory device allowing users to store and retrieve data at a high rate of speed. The term static is used because, once a value has been loaded into an SRAM cell, it will remain unchanged until it is explicitly altered or until power is removed from the device.

SROM - An acronym for supervisory read only memory. The SROM holds code that is used to boot the device, calibrate circuitry, and perform Flash operations. The functions of the SROM may be accessed in normal user code, operating from Flash.

Stack - A stack is a data structure that works on the principle of Last In First Out (LIFO). This means that the last item put on the stack is the first item that can be taken off.

Stack Pointer - A stack may be represented in a computer as inside blocks of memory cells, with the bottom at a fixed location and a variable stack pointer to the current top cell.

State Machine - The actual implementation (in hardware or software) of a function that can be considered to consist of a set of states through which it sequences.

Sticky - A bit in a register that maintains its value past the time of the event that caused its transition, has passed.

Stop Bit - A signal following a character or block that prepares the receiving device to receive the next character or block.

Switching - The controlling or routing of signals in circuits to execute logical or arithmetic operations, or to transmit data between specific points in a network.

Switch Phasing - The clock that controls a given switch, PHI1 or PHI2, in respect to the switch capacitor (SC) blocks. The PSoC SC blocks have two groups of switches. One group of these switches is normally closed during PHI1 and open during PHI2. The other group is open during PHI1 and closed during PHI2. These switches can be controlled in the normal operation, or in reverse mode if the PHI1 and PHI2 clocks are reversed.

Synchronous -

1. A signal whose data is not acknowledged or acted upon until the next active edge of a clock signal.
2. A system whose operation is synchronized by a clock signal.

Tap- The connection between two blocks of a device created by connecting several blocks/components in a series, such as a shift register or resistive voltage divider.

Terminal Count - The state at which a counter is counted down to zero.

Threshold - The minimum value of a signal that can be detected by the system or sensor under threshold consideration.

Transistor - A transistor is a solid-state semiconductor device used for amplification and switching, and has three terminals: a small current or voltage applied to one terminal controls the current through the other two. It is the key component in all modern electronics. In digital circuits, transistors are used as very fast electrical switches, and arrangements of transistors can function as logic gates, RAM-type memory, and other devices. In analog circuits, transistors are essentially used as amplifiers.

Tri-state - A function whose output can adopt three states: 0, 1, and Z (high-impedance). The function does not drive any value in the Z state and, in many respects, may be considered to be disconnected from the rest of the circuit, allowing another output to drive the same net.
UART - A Universal Asynchronous Receiver-Transmitter translates between parallel bits of data and serial bits.
User - The person using the PSoC device and reading this manual.
User Modules - Pre-built, pre-tested hardware/firmware peripheral functions that take care of managing and configuring the lower level Analog and Digital PSoC Blocks. User Modules also provide high level API (Application Programming Interface) for the peripheral function.
User Space - The bank 0 space of the register map. The registers in this bank are more likely to be modified during normal program execution and not just during initialization. Registers in bank 1 are most likely to be modified only during the initialization phase of the program.
V_{dd} - A name for a power net meaning "voltage drain". The most positive power supply. Usually 5 or 3.3 volts.
Volatile - Not guaranteed to stay the same value or level when not in scope.
V_{ss} - A name for a power net meaning "voltage source". The most negative power supply signal.
von Neumann Architecture: data and program instructions are stored in the same memory space. There is a single internal data bus that fetches Instructions and data are fetched over the same path.
Watchdog Timer - A timer that must be serviced periodically. If it is not serviced, the CPU will reset after a specified period of time.
Waveform - The representation of a signal as a plot of amplitude versus time.
XOR - See Boolean Algebra.

Bibliography

[1] Ball, Roy and Pratt, Roger. Engineering Applications of Microcomputers Instrumentation and Control. Prentice Hall (1984).

[2] Birkner, John M., Chua, Hua-Thye,“Programmable array logic circuit,” U. S. Patent 4124899 (Filed May 23, 1977. Issued November 7 (1978).

[3] Birkner, John M. PAL Programmable Array Logic Handbook. Santa Clara: Monolithic Memories, (1978)

[4] Birkner, John; Coli, Vincent, PAL Programmable Array Logic Handbook (2 ed.), Monolithic Memories, Inc, (1981).

[5] Borrie, John A.Modern Control Systems - A Manual of Design Methods. Prentice Hall (1986).

[6] Cavlan, Napoleone. “Field Programmable logic array circuit” U. S. Patent 4,422,072 (Filed July 30, 1981. Issued Dec. 20, 1983).

[7] Chassaing, Rulph and Reay, Donald. Digital Signal Processing.Wiley Interscience (2008).

[8] Cline, R. “A Single-Chip Sequential Logic Element,” IEEE International Solid Sate Circuits Conference, Digest of Technical Papers, 15-17, pp. 204-205, Feb (1978).

[9] Coppens, A.B. Simple equations for the speed of sound in Neptunian waters (1981) J. Acoust. Soc. Am. 69(3), pp 862-863.

[10] Doboli, Alex N., Currie, Edward H. “Introduction to Mix-Signal, Embedded Design”. Springer (2010).

[11] KLingman, Edwin E. Microprocessor Systens Design.Prentice Hall (1977).

[12] Labrosse, Jean J. Embedded Systems Buidling Blocks, Seconfd Edition. CMP Books (2002)

[13] Lenk, John D. Logic Designer’s Manual, Reston Publishing (1977).

[14] Lui, Donglin; Hu, Xiabo Sharon; Lemmon D. Michael; and Ling, Qiang. “Firm Real-Time System Scheduling Based on a Novel QoS Constraint”. IEEE Transactions of Computers, Vol. 55, No. 3, March (2006).

[15] Meador, Don. Aalog Signal Processing with Laplace trnsforms and Active Filter Design. Delmar (2002)

[16] Nekoogar, Farzad and Moriarty, Gene. Digital Control Using Digital Signal Processing. Prentice Hall (1999)

[17] Parr, E.A. The Logic Designer’s Guidebook.McGraw Hill (1984),

[18] Pellerin, David and Holley, Michael. “Practical Design Using Programmable Logic”. Prentice Hall, (1991).

[19] Smith, Carl H., Caruso, Michael J., and Schneider, Robert W. A New Perspective on Magnetic Sensing. www.sensorsmag.com (1998)

[20] Steinhart, I.S. and Hart, S.R. "Deep Sea Research" vol. 15 p. 497 (1968).

[21] Van Ess, David. "Ohmmeter". Application Note: AN2028. Cypress Semiconductor, pp.1-3, (2002)

Index

ΔL, 38
68HC11 (Motorola), 11
8 bit register, 5
8048's internal architecture, 8
8051 derivatives, 11

A/D converters, 11
Accumulator, 73
accumulator, 5
accumulator latch, 5
Active High, 73
Active Low, 73
ADC, 73
ADD, 50
Address, 73
addressable space, 8
Algorithm, 73
algorithms, 48
ALU, 5, 49
ALU , 10
Ambient Temperature, 73
Analog, 73
Analog Blocks, 73
Analog Output, 73
Analog Signal, 73
Analog Subsystem, 51
Analog-to-Digital Converter, 73
AND, 73
API, 73
Application Program Interface, 73
arithmetic logic unit, 5
Array, 73
Assembly, 74
Asynchronous, 74
ATmega8, 12
Atmel, 10
Atmel 80C51, 12
ATMEL at91SAM3 , 11
Atmel AVR, 12
Attenuation, 74
Automotive Electronics, 16
Avionic Electronics, 17
Bandgap Reference, 74
Bandwidth, 74
bank switch, 8
Bending strain, 38
Bias, 74
Bias Current, 74
BIBO, 21
Binary, 74
Bit, 74
bit manipulations, 46
Bit Rate, 74
bit testing, 46
bit-addressable, 11
bit-addressable storage, 10
Block, 74
Boolean Algebra, 74
Boolean handling, 10
BR, 74
Break-Before-Make, 75
Buffer, 75
Bus, 75
Byte, 75
byte-addressable, 11

C Language, 75
C language, 54
cache, 50
CAN, 46, 58
CAN bus, 60
Capacitance, 75
Capture, 75
Carbon resistors, 39
carry flag, 5
Central Processing Unit, 49
Chaining, 75
Checksum, 75
Clear, 75
Clock, 75
clock cycles, 10
Clock Generator, 75
clock oscillator, 10
CMOS, 10, 75

co-processors, 48
Coding Problems, 57
Communications Electronics, 17
Comparator, 75
Compiler, 75
complex algorithms, 47
compressive, 38
Configuration, 75
Configuration Space, 75
constantan, 42
constantan-to-copper, 42
Consumer Electronics, 17
Continuous Time Mode, 17
convolution calculations, 48
copper-to-copper , 42
cores, 11
CPU, 6, 49
CRC, 51, 76
Crowbar, 76
Crystal Oscillator, 76
CTM, 17
Cyclic Redundancy Check, 59, 76
Cypress CY8C29466, 14
Cypress CY8C34, 14
Cypress CY8C53, 15
Cypress Semiconductor, 10

D/A converters, 11
Darlington Pairs, 53
Data Frame, 61
Data memory, 6
de-multiplexer, 51
Debugging, 55
difference equations, 49
digital filtering, 46
Digital Subsystem, 51
Discrete Time Mode, 18
DMA, 47
DRAM, 51
DSPs, 47
DTM, 18
dynamic memory, 50

EEPROM, 7, 53
EEPROM , 50
eight level stack, 5
electrically erasable, 7
electrically programmable, 7
Embedded systems, 15
EPROM, 5, 7, 50
Error Frame, 61
etched, 50
external memory, 10
external RAM, 5
external register, 6

Fast Fourier Transforms, 46
feedforward, 18
Field Programmable Gate Arrays, 47
Firewire, 46
firm real-time system, 22
FLASH, 50
Flash, 7, 51
flip-flops, 5
FPGAs, 47
FRTS, 22

General Purpose I/O system, 53
GPIO, 53

hard real-time system, 22
Hard System Crashes, 57
Harvard memory, 4
Harvard memory architecture , 10
hot spots, 55
HRTS, 22

I/O pin, 8
IDE, 55
in-circuit emulators, 57
Industrial Electronics, 17
Infineon Technologies, 10
Input/Output, 5
instruction decoder, 5
Integrated Development Environments, 55
Intel 4004, 3
Intel 8008 , 3
Intel 8048, 3
Intel 8051, 8
Intel 8080, 3, 6
Intel 8085, 3
Intel 8749, 7
Intel microprocessors, 8
Inter-Integrated Circuit, 60
Internal memory, 10
interrupt handler, 53

known current, 43

latency, 20
LIN bus, 62
Link-Editing, 55
linker-locator, 55

Linkers, 55
Lock Up, 57
logic analyzers, 57
lower program counter, 5

MAC, 46
machine code, 54
machine cycle, 10
masked ROM, 7
matrix multiplication, 48
Medical Electronics, 17
memory address space, 10
memory-mapped file I/O, 8
Memory-mapped I/O, 4, 8
memory-mapped I/O, 10
memory-mapped I/O (8051), 11
microcontroller, 5
MIPS, 4
MMIO, 4
mnemonics, 54
modified Harvard, 4
modified Harvard architecture, 4
modules, 49
MOSFETs, 51
MOV, 50
Multiple addressing modes, 10
multiplexer, 51
multiplexing of address lines, 6

negative temperature coefficients, 39
Net, 80
NMI, 27
NMOS, 10
non-latching, 6
None-Maskable-Interrupt, 27
NTC, 39

on-chip, 11
on-chip registers , 10
OpAmps, 11, 48
open loop system, 19
operands, 50
oscillator, 5
overflow, 46
Overload Frame, 61

page number, 6
paged, 6
paging, 6
parity bit, 59
partial differential equations, 48
PC, 49
Permanent memory , 7
phase shift, 20
PIC MicroChip, 13
pipelining, 50
PMIO, 4
Poisson strain, 38
Port-mapped I/O, 8
port-mapped I/O, 4
positive temperature coefficients, 39
power saving mode, 10
precision illumination signal modulators, 51
PRISM, 51
Profilers, 55
program counter, 49
Program Counter stack, 5
Program Status Word, 5, 11
Program Status Word (8051), 11
programmable ROM, 7
PROM, 50
PRS, 51
PSW, 5, 11
PSW (8051), 11
PTC, 39
PWM, 5, 11, 51

quadrature decoder, 51

R/WM, 7
RAM, 5, 50
Read-Only, 7
Read-Only memory, 7
Read/Write memory, 7
Real Time Clocks, 53
recognize signals,, 48
reference resistor, 43
reference voltage, 43
Register banks, 10
register direct, 10
registers (8051), 10
relocatable, 57
relocation, 55
Remote Frame, 61
resistance tolerance, 42
resistive transducers, 45
ROM, 5, 7, 50
RS232 protocol, 59
RTC, 53
Runtime problems, 57

SCL, 60

SDA, 60
Seeback voltage, 42
Seebeck effect, 42
segmented, 6
segmented memory, 6
Sensors, 32
sensors, 17
Serial Peripheral Interface, 60
serial port, 10
Shear strain, 38
shift operations, 46
shift register, 51
single-stepping, 55
single-wire, 62
sleep timers, 53
soft real time system, 22
special register , 6
special registers, 11
SPI, 46, 60
SRTS, 22
stability, 20
stack, 5
standard libraries, 55
Start Of Frame, 61
Static memory, 50
statically latched, 5, 6
Steinhart-Hart, 40
Steinhart-Hart equation, 40
STMicroelectronics ST92F, 12
Strain gauges, 38
strain gauges, 43
SUB, 50

temperature coefficient, 39
temporary register, 5
tensile, 38
Texas Instruments , 10
The Program Status Word, 5
thermal sensors, 39
thermistors, 39, 43
thermoelectric effect, 42
Thomas Johan Seebeck, 42
TI MSP 430F, 13
transducers, 32
Type J thermocouples, 43

UART, 10, 11
UARTS, 5
ultraviolet light, 8
Universal Serial Bus, 59
USARTS, 5
USB, 59
useful heuristic, 42
UV, 8

vector interrupt handling, 10
vector-dot-product, 48
virtual memory, 57
virtual multiplexer, 51
voltage references, 53
Von Neumann, 4

Wheatstone bridge, 43
Winbond, 10

Chapter 2

Microcontroller Subsystems

In this chapter, discussion focuses on the various subsystems common to microcontrollers, viz., the CPU, interrupt controller, DMA functionality, busses, memories, clocking, general purpose I/O (GPIO), power management and hardware debugging support. PSoC3 and PSoC5 are used throughout this chapter to illustrate the key concepts involved in each of these topics.[1]

2.1 PSoC3 and PSoC5 - Basic Functionality

Before beginning a discussion of microcontroller subsystems it is important to discuss the functionality common to PSoC3 and PSoC5, e.g., they have:

- the same pin-out configuration, and are therefore pin and peripheral compatible,
- support for a variety of communications protocols, e.g., USB, I2C, CAN, UART, etc.,
- a common development environment, viz., PSoC Creator,
- high precision/performance analog functionality with up to 20-bit ADC and DAC support, in addition to comparators, OpAmps, PGAs, mixers, TIAs, configurable logic arrays, etc.,
- an easily configurable logic array,
- SRAM, Flash and EEPROM memory,
- analog systems that includes both switched-capacitance (SW) and continuous-time (CT) blocks, 20 bit sigma-delta converter(s), 8-bit DACs configurable for 12-bit operation, PGAs, etc.,
- digital systems that are based on Universal Digital Blocks (UDB) and specific function peripherals such as CAN and USB,
- programming and debugging support via JTAG, Serial Wire Debug (SWD) and Single Wire Viewer (SWV),
- a nested, vectored interrupt controller,
- a high performance DMA controller,

and,

[1] It should be noted that the basic architectures of both PSoC3 and PSoC5 are quite similar but because of the dramatic differences in the microprocessor cores employed in each case, implementation details of some aspects of these programmable systems on (a) chip are quite different. However, such differences are the not primary focus of this chapter and shall be treated, if at all, in detail elsewhere in this textbook.

- flexible routing to all pins.

However, there are some significant differences between PSoC3 and PSoC5, e.g.,

- PSoC3's CPU subsystem (core) is based on a single-cycle[2], 8-bit, 8051-based, Harvard architecture processor capable of operating at clock speeds up to 67 MHz, which permits it to outperform standard 8051 incarnations by as much as a factor of ten, or equivalently one order of magnitude.
- PSoC5's CPU subsystem (core) is based on a 32-bit, Harvard architecture, three-stage, pipelined, ARM Cortex-M3 processor capable of operating at clocks speeds up to 80 MHz. Its instruction set is the same as Thumb-2 and supports both 16- and 32-bit instructions. PSoC5 has a Flash cache that reduces the number of Flash accesses required and thereby lowers power consumption.

2.2 PSoC3 Overview

The fundamental approach to the PSoC architecture, and philosophy, has remained basically unchanged as it has evolved from being based on the proprietary M8C microprocessor to support for both 8051 and ARM cores. The latter processors, while based on quite different architectures, both control a standard set of analog/digital blocks and the system's I/O ports as shown in Figures 2.1 and 2.2, respectively.

PSoC3 integrates a single-cycle-per-instruction 8051 core, a programmable digital system, programmable analog components and configurable digital system resources together with a highly configurable I/O system. Internal communications is primarily based on the Arm Advanced High-Performance Bus (AHB) in conjunction with a multi-spoke bus controller called the Peripheral Hub (PHUB)[3]. This allows many of the functional blocks within PSoC3 to communicate with little or no CPU involvement. In addition, there is an Analog Global Bus (AGB) that can be used to connect to/from the I/O system. Secondary bus structures allow the CPU to communicate directly with the I/O ports. The EEPROM, Flash and SPC blocks are connected bus to enable SPC programming control. CPU subsystem connections to the cache and interrupt controller allow the CPU to communicate directly with both thereby minimizing the latency and any requirements for communicating with the peripheral controller.

The 8051 "core"[4] is capable of being clocked from DC to 67 megahertz, provides both hardware multiply and divide, 24 channels of Direct Memory Access (DMA), up to 8K each of Flash and SRAM and up to 2K of 1 million cycle, 20 year retention, EEPROM.

2.2.1 The 8051 CPU (PSoC3)

As discussed in Chapter 1, the 8051 microcontroller is something of a classic in the field of microprocessors and microcontrollers dating from 1980 when it was introduced by Intel Corporation.

In its simplest configuration it consisted of:

- An ALU
- Seven on-chip registers

[2]Single cycle refers to instructions being executed in a single machine cycle.

[3]The PHUB bus is based on the AMBA AHB protocol and consists of a central hub and radial spokes that are connected to one or more peripheral blocks.

[4]This core is fully compatible with the MCS-51 instruction set, i.e., it is "upward-compatible".

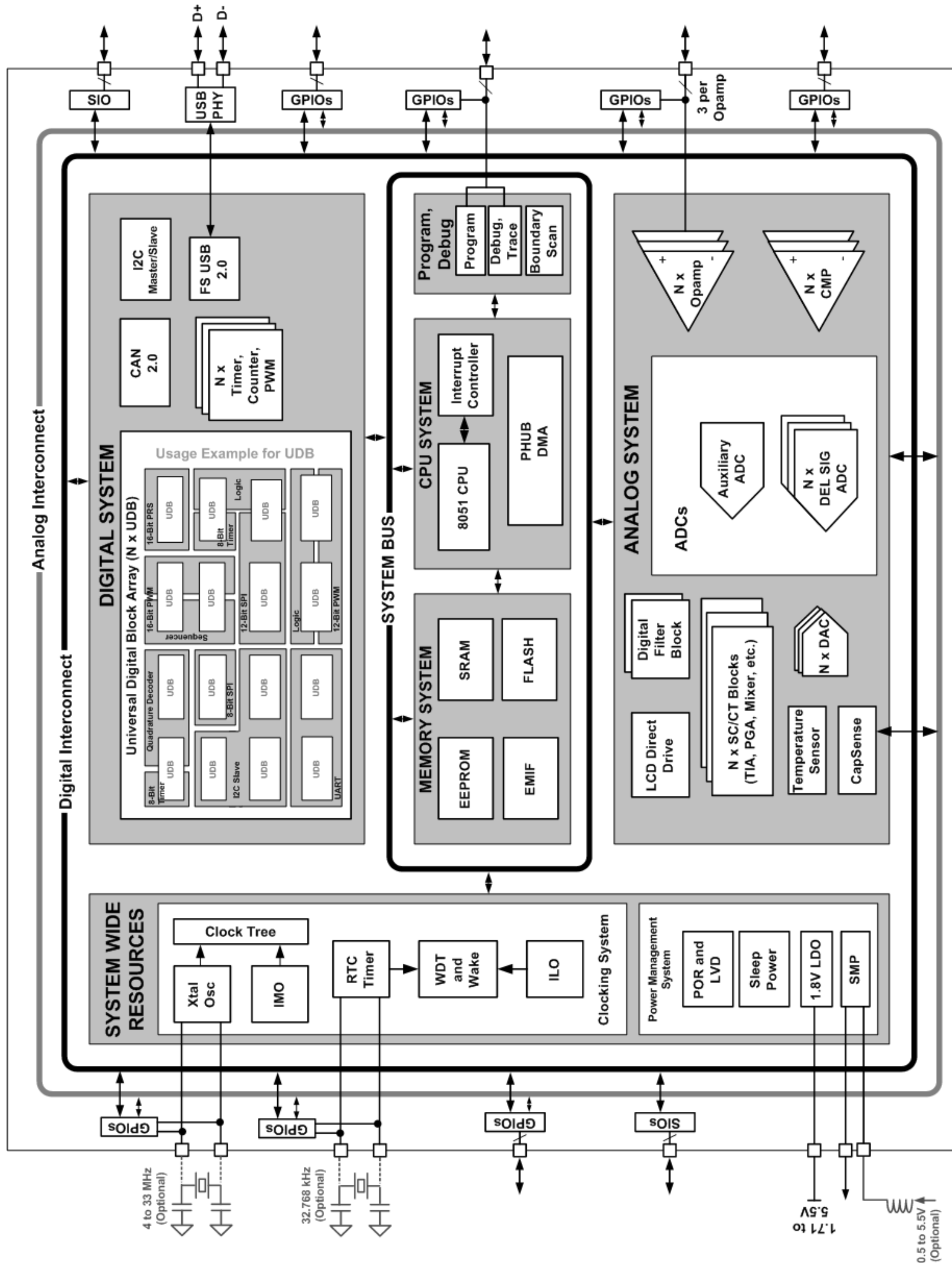


Figure 2.1: Top Level Architecture for PSoC3

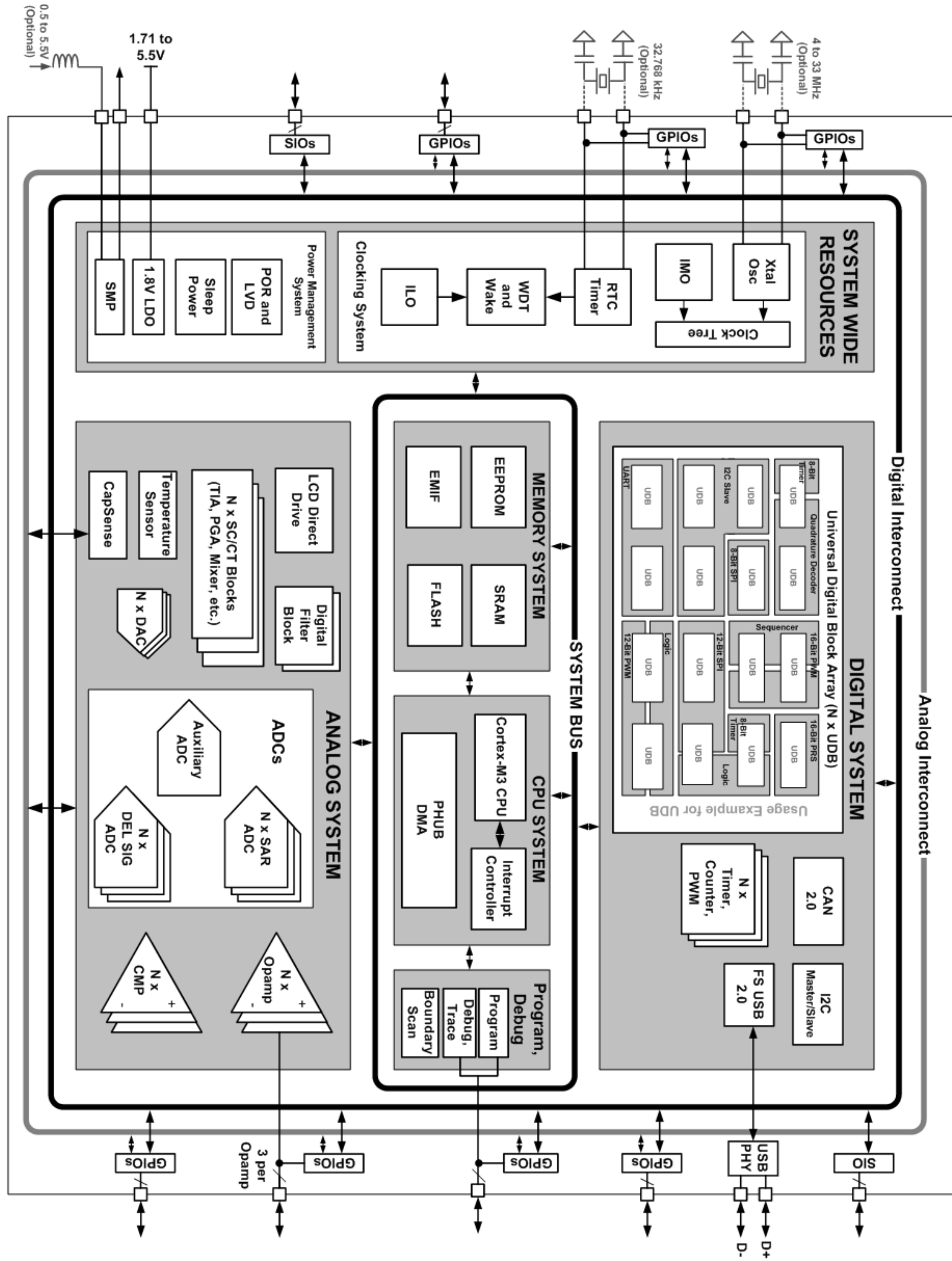


Figure 2.2: Top Level Architecture for PSoC5.

- A serial interface
- Two 16 bit timers
- Internal memory consisting of GP/bit-addressable storage and register banks and special function registers
- Support for 64K of external memory (code)[5]
- Support for 64K external memory (data)
- Four 8-bit I/O ports[6]
- 210 bit-addressable locations[7]

In addition to supporting memory-mapped I/O, the registers are also memory-mapped and the stack resides in RAM that is internal to the 8051. The ability to access individual bits makes it possible to set, clear AND, OR, etc., utilizing a single 8051 instruction. Register banks are contained in the lowest 32 bytes of the 8051's internal memory. Eight registers are supported, viz, R0-R7, inclusive, and their default locations are at addresses 0x00-0x07. Register banks can be used to provide efficient context switching and the active register bank is selected by bits in the Program Status Word (PSW). At the top of the internal RAM there are 21 special registers located at addresses $0x80 - 0xFF$. Some of these registers are bit- and byte-addressable, depending on the instruction addressing the register.

The 8051 implementation in PSoC3 has the following features:

- The architecture is RISC-based and pipelined.
- It is 100% binary-compatible with the industry standard 8051 instruction set, i.e., it is upward compatible in terms of executables.
- Most instructions operate in one or two machine cycles.
- It supports a 24-bit external data space that enables access to on-chip memory and registers, and to off-chip memory.
- A new interrupt interface has been provided that enables direct interrupt vectoring.
- 256 bytes of internal data RAM are available.
- The Dual Data Pointer (DPTR) has been extended from 16-bits in the "standard 8051" architecture to 24-bits to facilitate data block copying.
- Special Function Registers (SFRs) provide fast access to PSoC3 I/O ports and control of the CPU clock frequency.

2.2.1.1 8051 Wrapper

In order to most efficiently, and effectively, incorporate the 8051 core into PSoC3, a"wrapper" is provided as shown in Figure 2.3. This so-called wrapper is in fact simply logic that surrounds the core and provides an interface between the core and the rest of the PSoC3 system. The 8051 is one of two bus masters, the other being the DMA controller. Two bus slaves are available in the form of the PHUB, discussed in Section 2.2.6, and the on-chip SRAM, and are accessible via the 8051's external memory space. This configuration provides access to all of PSoC3's registers , as well as, externa memory. The wrapper also provides an SFR-I/O interface and gives direct access to the I/O port registers via the SFRs. A CPU clock divider is also included in the wrapper.

[5]The 8051 utilizes a separate 64K for data and code, respectively.

[6]I/O ports in the 8051 are "memory-mapped", i.e. to readgions for SFRwrite from/to an I/O port the program being executed must read/write to the corresponding memory location in the \same manner that a program would normally read/write to any memory location.

[7]128 of these are at addresses 0x20-0x2F with the remaining 73 being located in special function registers

Each port has two interfaces, one of the interfaces is to the PHUB to allow boot configuration and access to all of the I/O port registers, and the second interface is to the SFRs in the 8051 which gives faster access to a limited set of I/O port registers. The clock divider makes it possible to operate the CPU at frequencies that are divisors of the bus clock speed, cf Section 2.2.10.

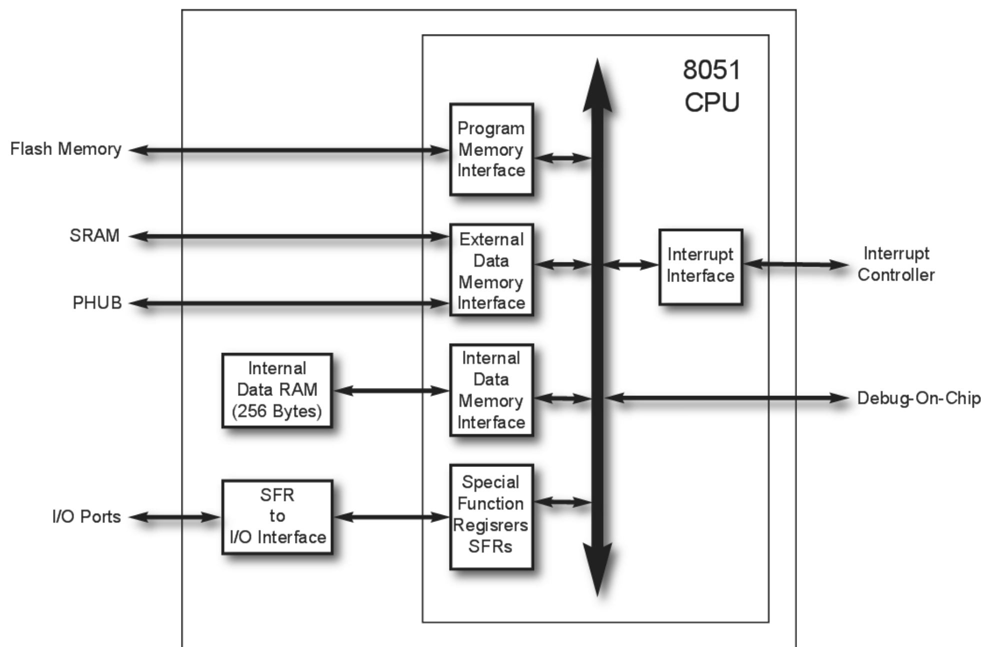


Figure 2.3: PSoC3's 8051 wrapper.

2.2.1.2 8051 Instruction Set

The 8051 instruction set consists of 44 basic instructions as shown in Table 2.7.[8] These basic instructions result in 256 possible instructions, of which 255 (24 3-bytes, instructions, 92 2-byte instructions and 139 1-byte instructions) are documented. The full set of opcodes is shown in Table 2.1.

2.2.1.3 Internal and External Data Space Maps

A diagram of the 8051's internal data space is shown in Figure 2.4. This space is divided into five regions as shown. While the Internal Data Memory addresses are in fact only one byte wide, implying that the address space is limited to 256 bytes, direct addresses higher than 7FH access one memory space and indirect addresses higher than 7FH access a different memory space. Thus the upper 128 bytes can be used as SFR space for ports, status bits, etc., if direct addressing is employed. Sixteen of the addresses in the SFR memory space are both bit and byte addressable

The lower 128 bytes consists of the lowest 32 bytes grouped as 4 banks of 8 registers referred to as R0-R7. Bank selection is determined by two bits in the PSW. The next 16 bytes, i.e., above the register banks, is a bit-addressable memory space with bit addresses ranging from 00H-7FH,

[8]The mnemonics used here for the 8051(8052) are copyrighted by Intel Corporation, 1980.

Table 2.1: The complete set of 8051 opcodes.

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
0x00	AJMP	LJMP	RR	INC	INC	INC	INC	INC	INC	INC	INC	INC	INC	INC	INC	INC
0x10	ACALL	LCALL	RRC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC
0x20	AJMP	RET	RL	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD
0x30	ACALL	RETI	RLC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC	ADDC
0x40	AJMP	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL	ORL
0x50	ACALL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL	ANL
0x60	AJMP	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL	XRL
0x70	ACALL	ORL	JMP	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0x80	AJMP	ANL	MOVC	DIV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0x90	ACALL	MOV	MOVC	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB	SUBB
0xa0	AJMP	MOV	INC	MUL	?	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0xb0	ACALL	CPL	CPL	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE	CJNE
0xc0	AJMP	CLR	SWAP	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH	XCH
0xd0	ACALL	SETB	SETB	DA	DJNZ	XCHD	XCHD	XCHD	XCHD	XCHD	XCHD	XCHD	XCHD	XCHD	XCHD	XCHD
0xe0	AJMP	MOVX	MOVX	CLR	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
0xf0	ACALL	MOVX	MOVX	MOVX	CPL	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV

inclusive. The 8051's instruction set includes a number of instructions for manipulating single bits in this area, using direct addressing.

Table 2.2 is a tabulation of the external memory addressable by PSoC3.

2.2.1.4 Instruction Types

The 8051 has five types of instructions:

1. **Arithmetic -** addition, subtraction, division, incrementing and decrementing.
2. **Boolean** - clearing a bit, complementing a bit, setting a bit, toggle a bit, move a bit to carry, etc.
3. **Data Transfer -** internal data, external data and lookup table data.
4. **Logical** - Boolean operations such as AND, OR, XOR and rotating/swapping of nibbles.

and,

5. **Program branching -** Conditional and unconditional jumps (branches) to modify program execution flow

2.2.1.5 Data Transfer Instructions

The 8051 is capable of three types of data transfer:

1. **External Data Transfer** - MOVX instructions are used to transfer data between the accumulator and an external memory address.
2. **Internal Data Transfers -** Direct, indirect, register and immediate addressing instructions allow data to be transferred between any two internal RAM locations of SFRs.
3. **Lookup Table Transfers** - MOVC instructions are used to transfer data between the accumulator and program memory addresses.

2.2.1.6 Data Pointer

The Data Pointer (DPTR) is located in a 16-bit register at 0x83 (high byte) and 0x82 (low byte), respectively which is used to access up to 64K, inclusive, of external memory.

Table 2.2: XDATA Address Map.

Address Range	Purpose
0x00 0000 - 0x00 1FFF	SRAM
0x00 2000 - 0x00 2FFF	Trace SRAM
0x00 4000 - 0x00 42FF	Clocking, PLLs and Oscillators
0x00 4300 - 0x00 430F	Power Management
0x00 4400 - 0x00 44FF	Interrupt Controller
0x00 4500 - 0x00 45FF	Ports Interrupt Control
0x00 4600 - 0x00 46FF	Reserved
0x00 4700 - 0x00 47FF	Flash Programming Interface
0x00 4900 - x00 49FF	I2C Controller
0x00 4E00 - 0x00 4EFF	Decimator
0x00 4F00 - 0x00 4FFF	Fixed Timer/Counter/PWMs
0x00 5000 - 0x00 51FF	General Purpose IOs
0x00 5300 - 0x00 530F	Output Port Select Register
0x00 5400 - 0x00 54FF	External Memory Interface (EMIF) Control Registers
0x00 5800 - 0x00 5FFF	Analog Subsystem Interface
0x00 6000 - 0x00 60FF	USB Controller
0x00 6400 - 0x00 6FFF	UDB Configuration
0x00 7000 - x00 7FFF	PHUB Configuration
0x00 8000 - 0x00 8FFF	EEPROM
0x00 A000 - 0x00 A400	CAN
0x00 C000 - 0x00 C800	Digital Filter Block
0x00 0000 - 0x00 FFFF	Digital Interconnect Configuration
0x00 0000 - 0x00 FFFF	Direct Access To Cache Memory
0x00 0000 - 0x00 FFFF	Flash Memory
0x00 0000 - 0x00 FFFF	Direct Access To 8051 IDATA
0x00 0200 - 0x00 021F	Test Controller (Internal)

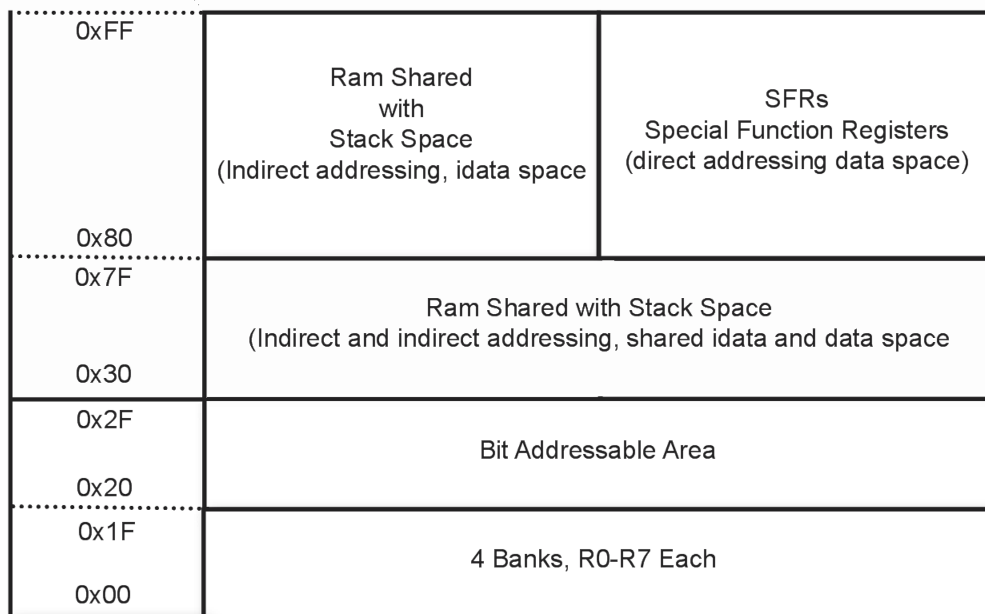


Figure 2.4: 8051 Internal Data Space Map

2.2.1.7 Dual Data Pointer SFRs

In order to facilitate the copying of blocks of data, four Special Function Registers (SFRs) are employed to hold two 16-bit pointers, DPTR0 and DPTR1 and INC DPTR can be used to switch between them. The active DPTR register is selected by the SEL bit (0x86) in the SFRs space, e.g., if the SEL bit is equal to zero, then DPTR0 (SFRs 0x83:0x82) is selected, otherwise DPTR1 is selected.

The data pointer select register is used in conjunction with the following instructions:

- INC DPTR
- JMP @A+DPTR
- MOVX @DPTR,A
- MOVX A,@DPTR
- MOVC A,A+DPTR
- MOV DPTR,#data16

2.2.1.8 Boolean Operations

Boolean instructions allow single bit operations to be performed on the individual bits of registers, memory locations and the CY Flag (the AC, OV and P flags cannot be altered by these instructions). The operations that can be conducted on individual bits are clear, complement, move, set, AND, OR, and tests for conditional jumps.

JC/JNC - jump to a relative address if the CY Flag is set or cleared
JB/JNB - jump to a relative address if the CY FLAG is set or cleared.
JBC - Jump to a relative address if a bit is set or cleared.
Short jump.

Table 2.3: Special Function Registers

	0/8 (Bit Addressable)	1/9	2/A	3/B	4/C	5/D	6/E	7/F
0xF8	SFRPRT15DR	SFRPRT15PS	SFRPRT15SEL					
0xF0	B		SFRPRT12SEL					
0xE8	SFRPRT12DR	SFRPRT12PS	MXAX					
0xE0	ACC							
0xD8	SFRPR6DR	SFRPRT6PS	SFRPRT15S6					
0xD0	PSW							
0xC8	SFRPRT5DR	SFRPRT5PS	SFRPRT5SEL					
0xC0	SFRPRT4DR	SFRPRT4PS	SFRPRT4SEL					
0xB8								
0xB0	SFRPRT3DR	SFRPRT3PS	SFRPRT3SEL					
0xA8	IE							
0xA0	P2AX	CPUCLK_DIV	SFRPRT1SEL					
0x98	SFRPRT2DR	SFRPRT2PS	SFRPRT2SEL					
0x90	SFRPRT1DR	SFRPRT1PS		DPX0		DPX1		
0x88		SFRPRT0PS	SFRPRT0SEL					
0x80	SFRPRT0DR	SP	DPL0	DPH0	DPL1	DPH1	DPS	

2.2.1.9 Program Status Word (PSW)

The Program Status Word is located at 0xD0 and contains information about the 8051's flags as shown in Figure 3.5. The Carry Flag (CF) can also be used as a 1-bit "Boolean accumulator", i.e., a 1-bit register for Boolean instructions. Flag 0 is a general purpose flag and RS0/RS1 are used to determine the active register. The Overflow Flag is set after a subtraction or addition if an arithmetic overflow has occurred. The parity bit is used in each machine cycle to maintain even parity of the accumulator byte and B is a bit-addressable register (accumulator) located at 0xF0 for multiply and divide operations.[9]

2.2.1.10 Stack Pointer

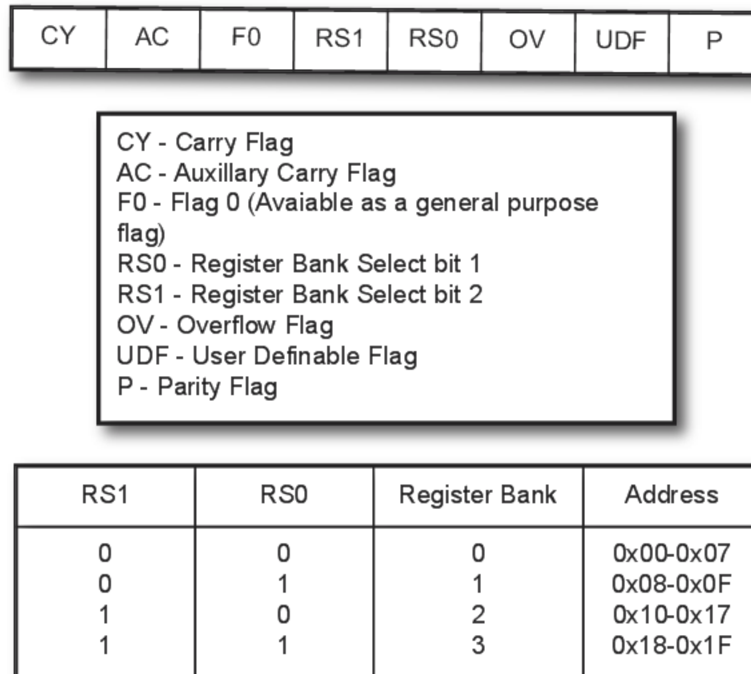
The Stack Pointer (SP) is an 8-bit register located at 0x81 that contains the default value of 0x07, when the system is reset. This causes the first "Push" to the stack to be stored in location 0x80 and therefore Register Bank 1, and potentially Register Banks 2 and 3 may not be accessible. However initializing the SP pointer will allow all of the Register Banks to be used.

2.2.1.11 Addressing Modes

The 8051 architecture supports seven addressing modes:

1. **Direct** - the operand is specified by a "direct" 8-bit address, but only internal RAM and special function registers can be addressed by this mode.

[9] Following an 8-bit by 8-bit multiplication, the resulting 16-bit (two byte) value is stored in A (Low byte) and B (High Byte), respectively.



RS1	RS0	Register Bank	Address
0	0	0	0x00-0x07
0	1	1	0x08-0x0F
1	0	2	0x10-0x17
1	1	3	0x18-0x1F

Figure 2.5: Program Status Word (PSW) [0xD0]

2. **Indirect** - a register, either R0 or R1, containing the 8-bit address of the operand is specified by the instruction. In this mode, the Data Pointer (DPTR) is used to specify 16-bit addresses.
3. **Immediate Constants** - Except for the Data Pointer, all 8051 instructions involving immediate addressing utilize 8-bit data values. In the case of the Data Pointer, a 16-bit constant must be used.
4. **Bit Addressing** - In this mode, the operand ia specified as one of 256 bits
5. **Indexed Addressing** - Indexed addressing uses the Data Pointer as a base register with an offset stored in the accumulator to point to an address in program memory that is to be read.This addressing mode is intended for reading data from look-up tables. In such cases, a 16-bit "base" register such as the DPTR or PC, is used to point to the base of a table and the accumulator holds a value that points to a particular entry in the table. Thus the actual address in program memory is the sum of the values held in the accumulator and 16-bit base register. [10]
6. **Register addressing** - eight registers (R0-R7) are used for register addressing. Instructions using these registers utilize the three least significant bits of the instruction opcode to specify a particular register. Since an address byte is not required use of ti mode, where possible, results in improved code efficiency. The bank select bits stored in the PSW determine which bank holds the register.
7. **Register Specific** - some instructions are used only in conjunction with specific registers such as the accumulator or DPTR and therefore an address byte is obviated, e.g., any

[10] Another form of indexed addressing is employed by "case jump" instructions, i.e., a jump instruction's destination address is determined by the sum of the base pointer and the value in the accumulator.

instruction referencing the Accumulator (A) are both accumulator and therefore register specific.

The following are illustrative examples of some of the most common addressing modes:

- SBB A,2FH (Direct addressing)
- SBB AA,@R0 (Indirect Addressing)
- SBB A,R4 (Register Addressing)
- SBB A,#31 (Immediate Addressing)

Absolute - ACALL and AJMP require the use of absolute addresses and store the 11 least significant bits of the address and the remaining 5 bits are derived form the five most significant bits of the Program Counter.

Relative - relative addressing is used with some of the jump instructions. The relative address serves as an 8-bit (-128-127) offset that is added to the program counter to provide the address of the next instruction to be executed. Use of relative addressing can result in the program code that is relocatable, i.e., does not have any memory location dependence.

Long - LCALL and LJMP require long addressing and consist of a 3-byte instruction that includes the 16-bit destination address as bytes 2 and 3. These opcode allow the full 64K code space to be used.

2.2.2 The 8051 Instruction Set

The following are brief descriptions of the 8051 instruction set. Additional information is available in Appendix F and references [17] and [29] .

ACALL LABEL - unconditionally calls a subroutine located at an address LABEL. When this instruction is invoked the program counter, and stack pointer, are both advanced two bytes so that the next instruction address to be executed, upon return from the subroutine, is stored on the stack.

ADD A, <src-byte> - performs an 8-bit addition of two operands, one of which is stored in the accumulator. The result of the addition is stored in the accumulator and the CY flag is set/reset as required by the results of the addition. <src-byte> can be R_n(a register), Direct (a direct byte), @R_i(indirect RAM), or $\#data$(immediate data)

ADDC A, <src-byte> - invokes an 8-bit addition of two operands based on the previous value of the CY flag. e.g. when carrying out 16-bit addition operations. <src-byte> can be R_n(a register), Direct (a direct byte), @R_i(indirect RAM), or $\#data$(immediate data).

AJMP addr11- Absolute jump using an 11-bit address. This is a two-byte instruction that uses the upper 3-bits of the address, combined with a 5-bit opcode, to form the first byte and the lower 8-bits of the address from the second byte. The 11-bit address replaces the 11-bits of the PC to produce the 16-bit address of the target. Therefore, the resulting locations are within the 2K byte memory page containing the **AJMP** instruction.

ANL <dest-byte>,<src-byte> - performs a bitwise logical AND between the dest and src byte and stores the result in dest.

ANL, bit - Performs a logical AND operation between the Carry bit and and a bit, placing the result in the Carry bit.

ANL, /bit - Performs a logical AND operation between the Carry bit and the inversion of a bit. placing the result in the Carry bit.

Table 2.4: Arithmetic Instructions

Mnemonic	Description	Cycles
ADD A,Rn	Add register to accumulator	1
ADD A,Direct	Add direct byte to accumulator	2
ADD A,@Ri	Add indirect RAM to accumulator	2
ADD A,#data	Add immediate data to accumulator	2
ADDC A,Rn	Add register to accumulator with carry	1
ADDC Direct	Add direct byte to accumulator with carry	2
ADDC A,@Ri	Add indirect RAM to accumulator with carry	2
ADDC A,#data	Add immediate data to accumulator with carry	2
SUBB A,Rn	Subtract register from accumulator with borrow	1
SUBB Direct	Subtract direct byte from accumulator with a borrow	2
SUBB A,@Ri	Subtract indirect RAM from accumulator with borrow	2
SUBB A,#data	Subtract immediate data from accumulator with borrow	2
INC A	Increment Accumulator	1
INC Rn	Increment register	2
INC Direct	Increment direct byte	3
INC @Ri	Increment indirect RAM	3
DEC A	Decrement accumulator	1
DEC Rn	Decrement register	2
DEC Direct	Decrment direct byte	3
DEC @Ri	Decrement indirect RAM	3
INC DPTR	Inrement data pointer	1
MUL AB	Multiply accumulator by B	2
DIV AB	Divide accumulator by B	6
DA A	Decimal adjust accumulator	3

CJNE <dest-byte>,<src-byte>, rel - Compares the magnitude of the first two operands and branches if they are not equal to the address given by adding **rel** (signed relative displacement) to he PC, after it has been incremented to the start of the next instruction. The carry flag is set if the unsigned, integer value of **<dest-byte>** is less than the unsigned integer value of **<src-byte>**; otherwise the carry is cleared. The first two operands allow four addressing mode combinations: the accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

CLR A - Clear the accumulator, i.e., reset all A bits to zero. Flags are not affected.

CLR bit - Clear the indicated bit, i.e., reset to zero. Flags are not affected. CLR can operate on the carry flag, or any directly addressable bit.

CPL A - Complement the accumulator.

CPL bit - Complement a bit. No other flags are affected.

DA A - Decimal adjust the accumulator. This instruction "adjusts" the eight-bit value in the accumulator resulting from the prior addition of two variables, each of which is in packed BCD format, resulting in the production of two four-bit values. Any ADD, or DDC, instruction may have been used for the addition.

DEC <src-byte> - Decrement the operand by one. <src> can be a direct address, an indirect address, the accumulator, or a register. ($00H \Rightarrow 0FFH$).

DIV AB - Divide the unsigned contents of the accumulator (A) by the unsigned contents of the B, placing the resulting integer value of the quotient in A and the integer remainder in B. If B originally contained 00H then both of the returned values will be undefined and the overflow flag will be set. The carry flag is cleared in all cases.

DJNZ <byte>,<rel-addr> - Decrement the byte value and jump if not zero to the relative address given by adding **rel** (signed relative displacement) to the PC, after it has been incremented to the first byte of the next instruction. No flags are affected and $00H \Rightarrow 0FFH$.

INC <src-byte> - Increment the operand by one. The operand can be a direct address, indirect address, register, accumulator or the data pointer (DPTR)[11]. No flags are affected.<src-byte> can be R_n(a register), Direct (a direct byte), $@R_i$(indirect RAM), or $\#data$(immediate data).

JB bit,rel - jump if the bit is set to one, otherwise proceed to the next instruction. The branch destination is computed by adding the signed relative displacement to the PC after incrementing the PC to the first bye of the next instruction. The tested bit is not modified and no flags are affected.

JBC bit,rel- Jump if the bit is set to one and clear the bit. The destination is computed by adding the signed relative displacement to the PC after incrementing the PC to the first bye of the next instruction.

JC rel - If carry is set, then branch to the destination computed by adding the signed relative displacement to the PC, after incrementing the PC to the first bye of the next instruction. No flags are affected.

JMP @A+DPTR - Jump indirect. Add the eight-bit unsigned contents of the accumulator to the sixteen-bit data pointer, and load the resulting sum to the program counter. The resulting sum is the address for the instruction. Sixteen-bit addition is performed (modulo 2^{16}). A carry from the low-order eight bits propagates through the higher-order bits. Neither the Accumulator nor the Data Pointer is altered. No flags are affected.

JNB bit - If the bit is not set, then branch to destination computed by adding the signed relative displacement to the PC after incrementing the PC to the first bye of the next instruction. No flags are affected.

JNC rel - If the carry flag is a zero, branch to the address indicated, otherwise proceed with the next instruction.The branch destination is computed by adding the relative-displacement to the PC, after incrementing the PC twice to point to the next instruction.The carry flag is not modified.

JNZ - If the accumulator contains a value other than zero, branch to the indicated address, otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative displacement after incrementing the PC twice. The accumulator is not modified and no flags are affected.

JZ - If the value in the accumulator is zero, branch to the address indicated, otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The accumulator is not modified. No flags are affected.

LCALL addr16 - calls a subroutine located at the indicated address. The instruction adds

[11]Incrementing the DPTR by 1 causes this 16-bit pointer to be increased by 1, An overflow of the lower byte (DPL), i.e., $0xFF \Rightarrow 0x00$, causes the upper byte (DPH) to be incremented. DPTR is the only PSoC3, 16 bit register that can be incremented in this manner.

Table 2.5: Jump Instructions.

Mnemonic	Description	Cycles
ACALL addr11	Absolute subroutine call	4
LCALL addr16	Long subroutine call	4
RET	Return from a subroutine	4
RETI	Return from an interrupt	4
AJMP addr11	Absolute jump	3
LJMP addr16	Long jump	4
SJMP rel	Short jump (relative address)	3
JMP @A + DPTR	Jump indirect relative to DPTR	5
JZ rel	Jump if accumulator is zero	4
JNZ rel	Jump if accumulator is non zero	4
CJNE A,Direct,rel	Compare immediate data to accumulator	5
CJNE A,#data,rel	Compare immediate data to accumulator	4
CJNE Rn,#data,rel	Compare immediate data to regsiter and jump if not equal	4
CJNE @Ri,#data,re;	Compare immediate data to indirect RAM an jump if not equal	5
DJNZ Rn,rel	Decrement register and jump if non zero	4
DJNZ Direct,rel	Decrement direct byte and jump if non zero	5
NOP	No operation	1

three to the program counter to generate the address of the next instruction and then pushes the result onto the stack (low-byte first), incrementing the Stack Pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively,with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64K-byte program memory address space. No flags are affected.

LJMP addr16 - Long Jump using a 16-bit address. This is a three byte unconditional jump to any location in the 64K program space. address,by loading the high-order and low-order bytes of the PC (respectively)with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.

MOV <dest-byte><src-byte> - The byte variable indicated by the src-byte is copied into the location specified by the first dest-byte. The source byte is not affected. No other register or flag is affected. There are fifteen combinations of source and destination addressing modes for this instruction.

MOVC A,@A+<base-reg> -loads the accumulator with a code byte, or constant, from program memory. The address of the byte fetched is the sum of the original unsigned eight-bit accumulator contents and the contents of a sixteen-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added to the accumulator, otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

MOVX A,@Ri - These instructions, tabulated in Table 2.6, transfer data between the accumu-

lator and a byte of external data memory, and is denoted by appending an X to MOV. There are two types of instructions, differing in whether they provide an eight-bit or sixteen-bit indirect address to the external data RAM. In the first type, the contents of R0 or R1, in the current register bank, provide an eight-bit address multiplexed with data on P0[12]. Eight bits are sufficient for external 1/0 expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX. In the second type of MOVX instruction,the Data Pointer generates a sixteen-bit address. P2 outputs the high-order eight address bits (the contents of DPH) while P0 multiplexes the low order eight bits (DPL) with data. The P2 Special Function Register retains its previous contents while the P2 output buffers are emitting the contents of DPH. This form is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports. It is possible in some situations to mix the two MOVX types. A large R4M array with its high order address lines driven by P2 can be addressed via the Data Pointer,or with code.

MUL AB - This instruction multiplies the unsigned eight-bit integers in the accumulator and in register B. The low-order byte of the sixteen-bit product is left in the accumulator,and the high-order byte in B. If the product is greater than 255 (OPPH)the overflow flag is set; otherwise it is cleared. The carry liag is always cleared.

NOP - Execution continues at the following instruction.Other than the PC, no registers or flags are affected.

ORL<dest-byte><src-byte> - performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect,or immediate addressing; when the destination is a direct address, the source can be the accumulator, or immediate data. When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

POP direct - causes the contents of the internal RAM location addressed by the Stack Pointer to be read ("POPped"), and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

PUSH direct - increments the Stack Pointer by one.The contents of the indicated variable is then copied ("PUSHed") into the internal RAM location addressed by the Stack Pointer. The flags are not affected.

RET - return from a subroutine by "POP"ing the return address from the stack and continue execution from that location. RET pops the high-and low-order bytes of the PC successively from the stack decrementing the Stack Pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL.No flags are affected.

RETI - return from an interrupt service routine by "POP"ing the return address from the stack, restoring the interrupt logic to accept interrupts at the same level of interrupt as the one just processed and continue execution from the address retrieved from the stack. (Note that the PSW is not automatically restored). RETI pops the high-and low-order bytes of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer is left decremented by two. No other registers are affected. *Special Note: PSW is not automatically restored to its pre-interrupt status.* Program execution continues at the resulting address,which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower- or same-level interrupt had been pending when the RETI instruction is executed, that instruction will be executed before the pending interrupt is processed.

[12]P0,P1, P2 and P3 are the SFR latches on ports 0,1,2 and 3 respectively.

Table 2.6: Data Transfer Instructions.

Mnemonic	Description	Cycles
MOV A,Rn	Move register to accumulator	1
MOV A,Direct	Move direct byte to accumulator	2
MOV A,@Ri	Move indirect RAM to accumulator	2
MOV A,#data	Move Immediate data to accumulator	2
MOV Rn,A	Move accumulator to register	1
MOV Rn,Direct	Move direct byte to register	3
MOV Rn,#data	Move immediate data to register	2
MOV Direct,A	Move accumulator to direct byte	2
MOV Direct,Rn	Move register to direct byte	2
MOV Direct,Direct	Move direct byte to direct byte	3
MOV Direct@Ri	Move indirect RAM to direct byte	3
MOV Direct,#data	Move immediate data to direct byte	3
MOV @Ri,A	Move accumulator to indirect RAM	2
MOV @Ri,Direct	Move direct byte to indirect RAM	3
MOV @Ri,#data	Move immediate data to indirect RAM	2
MOV DPTR,#data16	Load data pointer with 16-bit constant	3
MOVC A,@A+DPTR	Move code byte relative to DPTR to accumulator	5
MOVC A,@A+PC	Move code byte relative to PC to accumulator	4
MOVX A,@Ri	Move external RAM (8-bit) to accumulator	3
MOVX A,DPTR	Move external RAM (16-bit) to accumulator	2
MOVX @Ri,A	Move accumulator to external RAM (8-bit)	4
MOVX DPTR,A	Move accumulator to external RAM (16-bit)	3
PUSH Direct	Push direct byte onto stack	3
POP Direct	Pop direct byte from stack	2
XCH A,Rn	Exchange direct byte with accumulator	2
XCH A,Direct	Exchange direct with accumulator	3
XCH A<@Rn	Exchange indirect RAM with accumulator	3
XCH A,@Ri	Exchange low order indirect digit RAM with accumulator	3

Table 2.7: Logical Instructions.

Mnemonics	Description	Cycles
ANL A,Rn	AND register to accumulator	1
ANL A,Direct	AND direct byte to accumulator	2
ANL A@Ri	AND indirect RAM to accumulator	2
ANL A,#data	AND immediate data to accumulator	2
ANL Direct,A	AND accumulator to direct byte	3
ANL Direct,#data	AND immediate data to direct byte	3
ORL A,Rn	OR regsiter to accumulator	1
ORL A,Direct	OR direct byte to accumulator	2
ORL A,@Ri	OR indirect RAM to accumulator	2
ORL A,#data	OR immediate data to accumulator	2
ORL Direct,A	OR accumulator to direct byte	3
ORL Direct,#data	OR immediate data to direct byte	3
XRL A,Rn	XOR regsiter to accumulator	1
XRL A,Direct	XOR direct byte to accumulator	2
XRL A@Ri	XOR indirect RAM to accumulator	2
XRL A,#data	XOR immediate data to accumulator	2
XRL Direct,A	XOR accumulator to direct byte	3
XRL Direct,#data	XOR immediate data to direct byte	3
CLR A	Clear the accumulatorComplet	1
CPL A	Complement the accumulator	1
RL A	Rotate accumulator left	1
RLC A	Rotate accumulator left through carry	1
RR A	Rotate accumulator right	1
RRC A	Rotate accumulator right through carry	1
SWAP A	Swap nibbles within accumulator	1

RL A - rotates the contents of the accumulator A, one bit position to the left. The eight bits in the accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position.No flags are affected.
RLC A - rotates the contents of the accumulator one bit position to the left through the Carry flag. The eight bits in the accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position.No flags are affected.
RR A - rotate the contents of the accumulator one bit position to the right. The eight bits in the Accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.
RRC A - rotate the contents of the accumulator one bit position to the right through the Carry flag. The eight bits in the accumulator and the carry flag are rotated together, one bit to the right. Bit 0 moves into the carry flag; the original value of the Carry flag moves into the bit 7 position. No other flags are affected.
SETB - sets the indicated bit to one. SETB can operate on the Carry flag or any directly addressable bit. No other flags are affected.

Table 2.8: Boolean Instructions.

Mnemonic	Description
CLR C	Clear carry
CLR bit	Clear direct bit
SETB C	Set Carry
SETB bit	Set direct bit
CPL C	Complemenmt carry
CPL bit	Complement direct bit
ANL c,bit	AND direct bit to carry
ANL C,/bit	AND indirect bit to carry
ORL C,bit	OR direct but to carry
ORL C,/bit	OR complement of direct
MOV C,bit	Move direct bit to carry
MOV bit,C	Move carry to direct bit
JC rel	Jump if carry is set
JNC rel	Jump if no carry is set
JB bit,rel	Jump if direct bit is set
JNB bit,rel	Jump if direct bit is not set
JBC bit,rel	Jump if direct bit is set and clear bit

SJMP rel - causes a Short Jump using an 8-bit signed offset relative to the first byte of the next instruction. This instruction causes the program to make an unconditional control branche to the address indicated.The branch destination is computed by by adding the signed displacement to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to the 127 bytes following it.
SUBB A,<src-byte> - Subtract with borrow results in a subtraction of an operand and the previous value of the CY flag. (A <= A - <operand>- CY). This instruction subtracts the indicated variable and the Carry flag from the accumulator, leaving the result in the accumulator. SUBB sets the Carry (borrow) flag if a borrow is needed for bit 7, and clears C, otherwise. (If

C was set before executing a **SUBB** instruction, this indicates that a borrow was needed for the previous step in a multiple precision subtraction, so the carry is subtracted from the accumulator along with the source operand.) A C is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6. When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number. The source operand allows four addressing modes: register,direct, register-indirect or immediate. <src-byte>can be R_n(a register), Direct (a direct byte), $@R_i$(indirect RAM), or $\#data$(immediate data).
SWAP A- interchanges (SWAPs) the low- and high-order nibbles (four-bit fields) of the accumulator (bits 3-0 and bits 7-4). The operation is equivalent to a four-bit rotate instruction. No flags are affected.
XCH A,<byte> - loads the accumulator with value of the byte variable and laods the accumulator contents to the byte variable.The src/dest operand can use register, direct, or register-indirect addressing.
XCHD A,@Ri - exchanges the low-order nibble of the accumulator (bits 3-0), generally representing a hexadecimal or BCD digit,with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected.
XRL <dest-byte><src-byte> - This instruction performs a bitwise, logical, Exclusive-OR operation between <dest-byte>and <src-byte>, storing the results in <dest>. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect,or immediate addressing; when the <dest>is a direct address, <src> can be the accumulator or immediate data. (Note When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.)
XCHD - exchanges low-order undirect digit RAM with the accumulator.
Undefined - OpCode 0xA5 is an undocumented function.

2.2.3 ARM Cortext M3 (PSoC5)

The ARM CORTEX M3 utilizes a three stage, pipelined, Harvard-based bus architecture to provide a single cycle, hardware-based multiply/divide capability. It also supports the Thumb-2[13] instruction set [26]. Thumb instructions include arithmetic operations, logical operators, conditional/unconditional branches and store/load data operations. I/O and exception handling typically require the use of 32-bit ARM instructions.[14] It should be noted, however, that there are inherent limitations of the Thumb instruction set. While the use of Thumb instructions, guarantee efficient code execution and power consumption when employing Thumb operators is not guaranteed.[25] The ARM CORTEX M3 provides hardware support that greatly facilitates debugging by providing trace, profiling, breakpoints, watch points and code patching. The Advanced RISC Machines (ARMs), have a 32-bit architecture with sixteen registers, one of which is the program counter (PC). Most the instructions have a 4-bit condition code to facilitate branching.

2.2.3.1 Thumb Instructions

Some modern microprocessor architecture can be described as a Complex Instruction Set Computer (CISCs) and are capable of carrying out arbitrarily complex instructions. Another class

[13]The Thumb instruction set is a subset of the 32-bit instruction set that has been compressed from 32 bits to 16 bits resulting in a reduction in code density of approximately 30%. Because of this reduction it is possible to maintain more instructions in the on-chip memory which further reduces power consumption since of-chip fetches tend to consume more power than on-chip fetches.

of microprocessors are referred to as RISC machines, or Reduced Instruction Set Computer(s). RISC instructions are usually executed in a single clock instruction and can result in substantially improved execution times. Such improvement is not without cost however, since more RISC instructions may be required than CISC instructions to execute a given program, which in turn means increased memory requirements.

Advanced RISC Machines (ARMs) utilize a set of instructions referred to as Thumb Instructions which consists of 16-bit instructions that are "extensions" of the 32-bit ARM instructions. Thumb instructions are fetched as 16-bit instructions[14] and then expanded utilizing dedicated hardware within the microprocessor to 32-bits. Thus Thumb instructions and their 32-bit counterparts are functionally equivalent.[15]ARM instructions are aligned on a four-byte boundary and Thumb instructions are aligned on a two-byte boundary. Any Thumb instruction that that involves data processing operates on 32-bit values. Instruction fetches and data access instructions create 32-bit addresses. In the Thumb state, eight general purpose, integer registers (R0-R7) are employed.

Thumb instructions fall into the following categories:

- Arithmetic
- Branch
- Extend
- Load
- Logical
- Move
- Process.or State Change
- Push and Pop
- Reverse
- Shift and Rotate
- Store

Thumb instructions are a 16-bit subset of ARM 32-bit instructions which can be conditionally executed, while Thumb instructions are always executed. The Thumb-2 instruction set consists of a mixture of 16- and 32-bit instructions.

2.2.4 Interrupts and Interrupt Handling

As discussed in Chapter 1, interrupts and interrupt handling are extremely important aspects of many embedded system applications. Proper treatment of interrupts allows the most efficient response of such systems and ensures that requests are handled in the appropriate order with minimal latency. An interrupt controller provides a hardware resources that allow the system to suspend tasks prior to their completion.

The interrupt controller employed in PSoC3 has a number of enhanced features not available in the original 8051, e.g.,

- eight levels of "nestable" interrupts,
- multiple I/O vectors,

[14]Thus conserving memory space.

[15]It should be noted when handling exceptions, the processor must be in the "ARM state", i.e., exceptions cannot be handled in the Thumb state and therefore cannot be handled by Thumb instructions.

- programmatic interrupts,
- programmatic clearing of interrupts,
- 32 interrupt vectors and levels of interrupt priority,
- dynamic assignment of one of eight priorities,

As shown in Figure 2.6, PSOC3's integral interrupt controller supports up to 32 interrupt signals, inclusive, which when active are processed by the interrupt controller. Each of these inputs can be enable/disabled programmatically and a dedicated interrupt vector table[16], stores the addresses of the respective interrupt service routines (ISRs). Under program control the priority assigned to an input signal can be changed as well as the vector address.

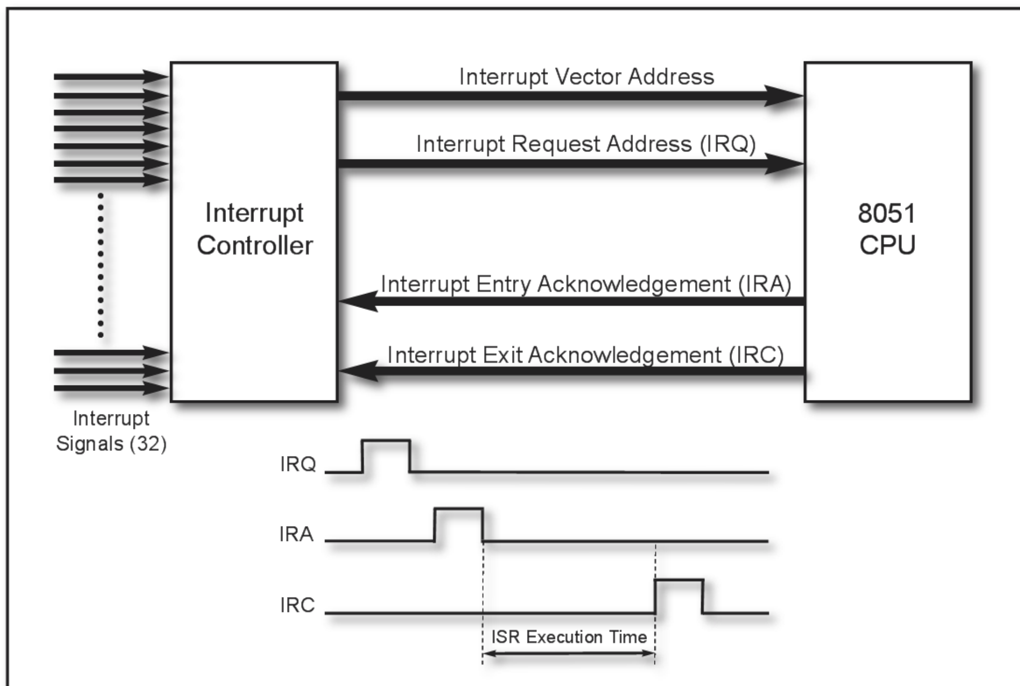


Figure 2.6: PSoC3 Interrupt Controller.

When an interrupts occur, the interrupt controller processes them and assigns them a priority based on the preassigned interrupt priority for each interrupt signal.[28] When an interrupt occurs all information required to reinstate the interrupted task must be stored on the stack as discussed in Chapter 1. If the program has been written in C, then the C compiler will automatically introduce the necessary code to store the required information on the stack, However, if the program has been written in assembly language the necessary push and pop instructions must be manually included in the assembly source file so that prior to entry into the ISR and following a return and prior to attempting to resume the interrupted task the required information is pushed/popped to/from the stack.

Two hardware stacks are maintained by the interrupt controller, one for storage of the interrupt priorities and the other for the related vector addresses.When an interrupt acknowledgement

[16]Such tables are sometimes referred to as a "jump" tables.

Table 2.9: Interrupt Vector Table (PSoC3)

#	Fixed Function	DMA	UDVB
0	LVD	phub_termout0[0]	udb_intr[0]
1	ECC	phub_termout0[1]	udb_intr[1]
2	Reserved	phub_termout0[2]	udb_intr[2]
3	Sleep (Pwr Mgr)	phub_termout0[3]	udb_intr[3]
4	PICU[0]	phub_termout0[4]	udb_intr[4]
5	PICU[1]	phub_termout0[5]	udb_intr[5]
6	PICU[2]	phub_termout0[6]	udb_intr[6]
7	PICU[3]	phub_termout0[7]	udb_intr[7]
8	PICU[4]	phub_termout0[8]	udb_intr[8]
9	PICU[5]	phub_termout0[9]	udb_intr[9]
10	PICU[6]	phub_termout0[10]	udb_intr[10]
11	PICU[12]	phub_termout[11]	udb_intr[11]
12	PICU[15]	phub_termout0[12]	udb_intr[12]
13	Comparator Int	phub_termout0[13]	udb_intr[13]
14	Switched Cap Int	phub_termout0[14]	udb_intr[14]
15	I^2C	phub_termout0[15]	udb_intr[15]
16	CAN	phub_termout0[0]	udb_intr[16]
17	Timer/Counter0	phub_termout0[1]	udb_intr[17]
18	Timer/Counter1	phub_termout0[2]	udb_intr[18]
19	Timer/Counter2	phub_termout0[3]	udb_intr[19]
20	Timer/Counter3	phub_termout0[4]	udb_intr[20]
21	USB SOF Int	phub_termout0[5]	udb_intr[21]
22	USB ARB Int	phub_termout0[6]	udb_intr[22]
23	USB Bus Int	phub_termout0[7]	udb_intr[23]
24	USB Endpoint [0]	phub_termout0[8]	udb_intr[24]
25	USB Endpoint Data	phub_termout0[9]	udb_intr[25]
26	Reserved	phub_termout0[10]	udb_intr[26]
27	Reserved	phub_termout0[11]	udb_intr[27]
28	DFB Int	phub_termout012]	udb_intr[28]
29	Decimator Int	phub_termout0[13]	udb_intr[29]
30	PHUB Error Int	phub_termout0[14]	udb_intr[30]
31	EEPROM Fault Int	phub_termout0[15]	udb_intr[31]

entry (IRA) is received from the CPU, the interrupt controller pushes the current interrupt vector address and priority level to their respective stacks. When an acknowledgement for an interrupt exit (IRC) is received, the interrupt controller pops the previous state information from the stack.

Interrupts can be nested so that a higher priority interrupt can "interrupt" a lower priority interrupt. Interrupts that occur while PSoC3's microcontroller is shutdown, e.g., while asleep, should be of the type referred to as "sticky" interrupts[17], i.e., interrupts asserted while PSoC3 is inactive must be held until PSoC3 "wakes up".

If an interrupt requests occurs which is of higher priority than that assigned to the currently executing task, the current task is suspended, and the higher priority task is invoked. Once completed the lower priority task resumes. Priorities are assigned numbers in the range of 0-31, with zero being the highest priority and 31 the lowest. If two tasks have been assigned the same priority and their respective interrupt requests occur simultaneously, then the task with the lower vector number has priority.

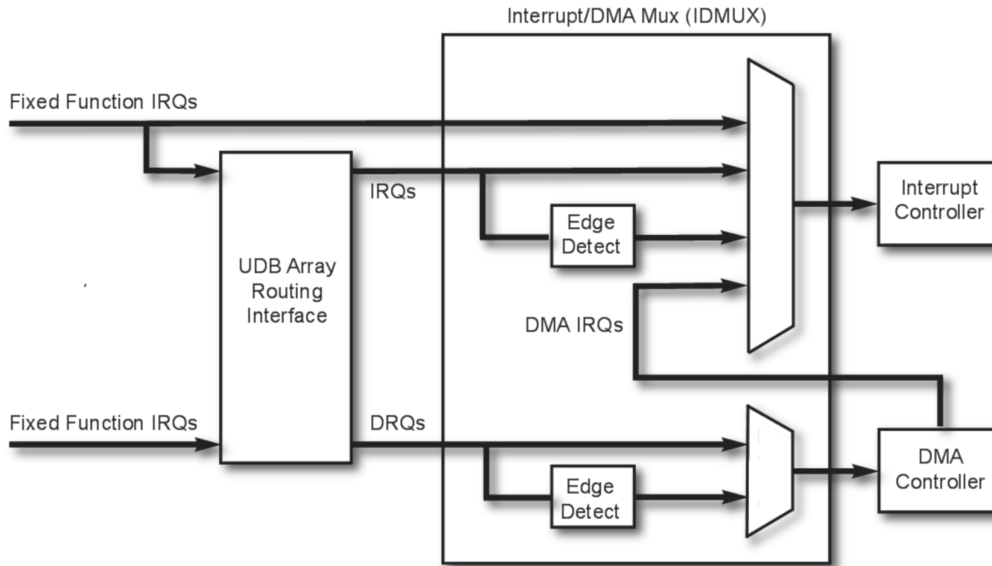


Figure 2.7: Interrupt processing in the IDMUX

2.2.4.1 Interrupt Lines

As discussed previously, the interrupt controller has 32 interrupt input lines, numbered as shown in Figure 2.8, and possible input sources asserted on these lines are defined as:

1. **Fixed Function -** these are asserted by peripherals such as I2C, Sleep, CAN, Port Interrupt Controller Unit (PICU) and the Low Voltage Detector (LVD).
2. **DMA Controller Interrupts -** these signal the completion of a DMA transfer.
3. **UDB -** interrupts initiated by various Universal Device Blocks implemented as timers, counters, etc.

[17]This is to avoid the possibility of an interrupt being asserted and then cleared while the microprocessor is "sleeping" thereby resulting in a missed interrupt request.

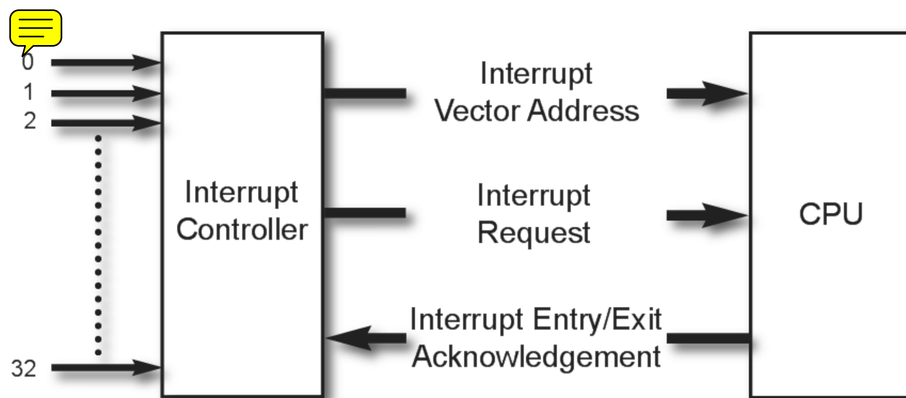


Figure 2.8: Interrupt controller block diagram.

However, each interrupt line is assigned one of these three types of interrupt sources as shown in Figure 2.9, and the designation for each line is determined by the IDMUX control register, IDMUX.IRQ_CTL,

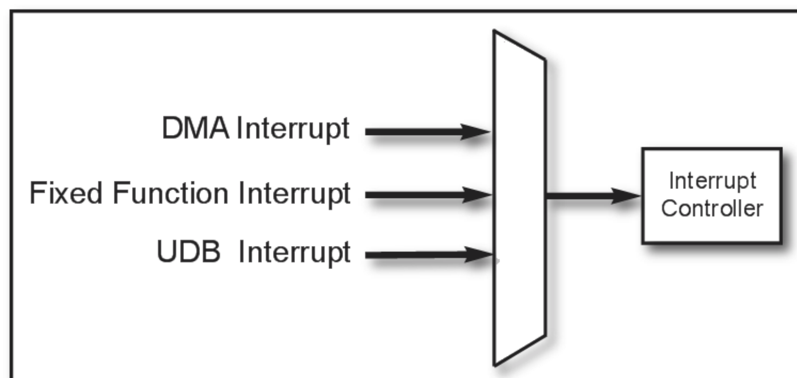


Figure 2.9: Interrupt Signal Sources.

Interrupt lines pass through a multiplexer on their way to the interrupt controller, as shown in Figure 2.7, which selects one of the following: a Fixed Function interrupt request (IRQ), a Universal Digital Block (UDB) IRQ with level, on a UDB IRQ with Edge or a DMA IRQ. The IDMUX,IRQ_CTL register determines the mux path with respect to IRQ selection.

The interrupt controller supports two types of interrupt assertions on the lines:

1. **Level Shift** - an interrupt request is initiated by a shift of the level of the interrupt line.
2. **Pulse** - a pulse on a pulse-designated interrupt line creates an IRQ when the low-to-high edge transition occurs, which cause the pending bit for that interrupt line to be set. In the event that a second pulse occurs while the first is still pending, it has no effect.When the CPU acknowledges receipt of the IRQ by transmitting an IRA, the pending bit for that line is reset. If another pulse now occurs, the pending bit is set again, even if the first ISR is still active.

2.2.4.2 Enabling/Disabling Interrupts

The interrupt controller's Enable register (SETEN) and Clear ENABLE registers allow interrupt lines to be enabled and disabled, respectively. Writing a 1 to the SETEN register enables an interrupt, while a zero has not effect. Reading a one from the the SETEN register implies interrupt is enable and a zero implies the interrupt is disabled. Similarly, writing a 1 to CLREN register disables an interrupt and writing a zero has not effect. Reading a one from the the CLREN register implies that an interrupt is enabled and reading a zero implies that it is disabled.

2.2.4.3 Pending Interrupts

The "Pending" bit is set when the interrupt controller receives an interrupt signal. This can also be set/cleared programmatically by using the "Set Pending" register (SETPEND) and the "Clear Pending" register (CLRPEND), respectively. Each of the bits in these registers represents the status of one interrupt line. Interrupt requests can be made by either asserting a level shift or a pulse on an interrupt line. In either case, following an IRQ, the pending bit is cleared immediately once an interrupt acknowledgement has been received from the CPU. Should a new pulse be received on the same line after receipt of the CPU's acknowledgement, the pending bit is set. However, when a line level shift occurs, the interrupt controller checks the status of the line after it receives an acknowledgment that the CPU has exited the interrupt service routine (ISR).

2.2.4.4 Interrupt Priority

The proper handling of priorities is obviously very important and there are two possibilities to be considered in such cases:

1. **An interrupt occurs while an interrupt service routine (ISR) for the previous interrupt is being executed.** If the interrupt is of higher priority, then the ISR is suspended, the information required to reinstate that ISR is placed on the stack and the ISR associated with the higher priory interrupt is invoked. If the priority of the interrupt is lower than that of the prior interrupt, then the ISR continues execution until completed, at which point, if no new interrupt requests have been received, the ISR for the most recent requested is invoked. If the most recent interrupt is of the same priority as that of the currently executing ISR then the ISR continues execution and, upon completion, the new interrupt ISR is invoked.
2. **Two interrupts occur contemporaneously.** If they are of the same priority, then the interrupt with the lower index number[18] is serviced first. Otherwise the interrupt with the higher priority is serviced first.

When an interrupt signal occurs, i.e., an IRQ, the pending bit for that line is set, in the pending register , indicating that an IRQ has occurred, and is waiting. The priority for this IRQ is read and a determination is made as to when this request should be serviced. The request, and the associated vector address, are then sent to the CPU. At this point the CPU acknowledges the request and returns an Interrupt Entry Acknowledgement signal (IRA). When the ISR is completed the CPU sends and Interrupt Exit Acknowledgment (IRC),

2.2.4.5 Interrupt Vector Addresses

PSoC3 allows the ISR starting addresses to be explicitly specified, i.e., the addresses are programmable. Thus calling an ISR does not involve a branch instruction and therefore the ability

[18]The interrupt controller's 32 input lines are numbered from 0-31 and referred as "index numbers".

Table 2.10: Bit Status During Read and Write

Register	Operation	Bit Value	Comment
SETEN	Write	1	To enable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled
CLREN	Write	1	To disable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled

Table 2.11: Pending Bit Status Table

Register	Operation	Bit Value	Comment
SETPEND	Write	1	To put an interrupt to pending
		0	No effect
	Read	1	Interupt is pending
		0	Interrupt is not pending
CLRPEND	Write	1	To clear a pending interrupt
		0	No effect
	Read	1	Interrupt is pending
		0	Interrupt is not pending

to make direct calls to an ISR reduces latency. The programmable ISR addresses are stored in the 16 bit Vector Address registers, VECT[0...31][19]. When writing to these registers, the LSB must be written first and followed by the MSB. When an IRQ occurs, the respective address is passed to the CPU for execution of the appropriate ISR.

2.2.4.6 Sleep Mode Behavior

It should be noted that in Sleep Mode, all of the status and configuration registers associated with interrupts retain their values. However, the Pending and Interrupt Controller stack registers are set with the “power-on” value at wakeup.

2.2.4.7 Port Interrupt Control Unit

PSoC3 has a Port Interrupt Control Unit (PICU) which interfaces to the GPIO pins and provides a way to process externally generated interrupts that:

- support 8 pins,
- handle rising/falling/both-edge interrupts,
- interfaces to the PHUB over AHB for reading/writing to its registers,
- does not support Level sensitive interrupts,
- allows pin interrupts to be individually disabled,
- transmits single interrupt request (PIRQ) to the interrupt controller,

and,

- has pin status bits to allow easy determination of the source of the interrupt, at the pin level.

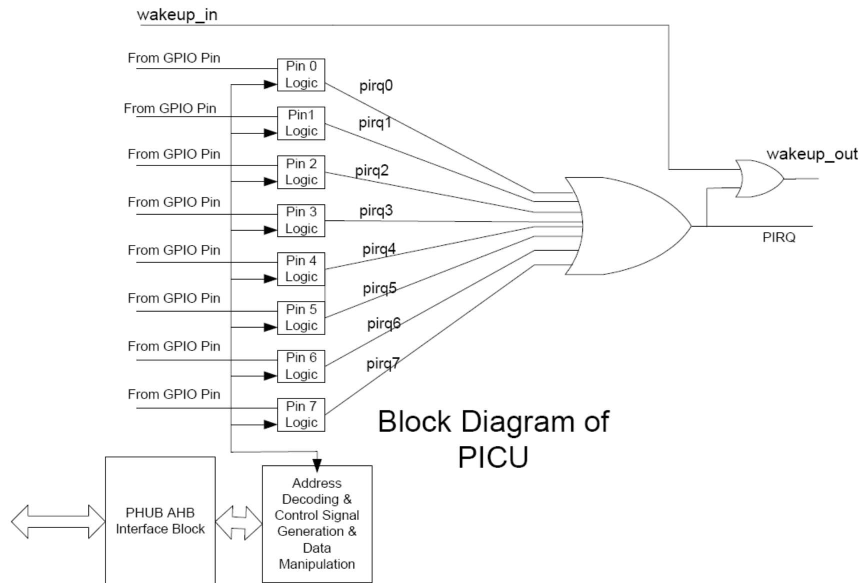
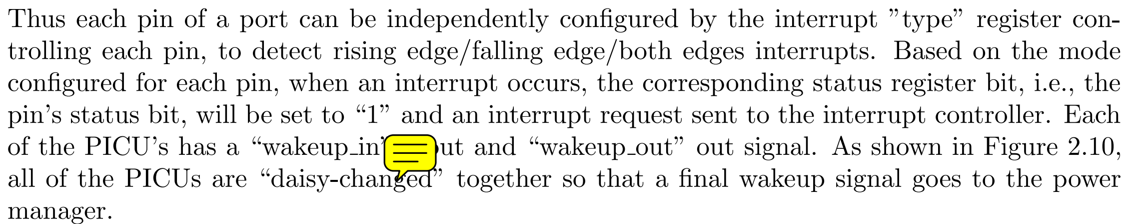


Figure 2.10: PICU Block Diagram

[19]There are 32 of these registers , i.e., one for each of the input lines.

Thus each pin of a port can be independently configured by the interrupt ”type” register controlling each pin, to detect rising edge/falling edge/both edges interrupts. Based on the mode configured for each pin, when an interrupt occurs, the corresponding status register bit, i.e., the pin’s status bit, will be set to “1” and an interrupt request sent to the interrupt controller. Each of the PICU’s has a “wakeup_in” input and “wakeup_out” out signal. As shown in Figure 2.10, all of the PICUs are “daisy-changed” together so that a final wakeup signal goes to the power manager.



2.2.4.8 Interrupt Nesting

PSoC3 supports up to eight levels of “nested” interrupts. Nesting occurs whenever a lower level interrupt service routine is suspended as a result of the receipt of a higher level interrupt. Interrupt nesting involves both the CPU stack and the interrupt controller stack(s) which store the interrupt number and priority. Two upward growing stacks with a depth of eight levels are maintained by the interrupt controller, viz., STK which stores the interrupt priority and STK_INT_NUM which stores the interrupt number.

The CPU stack is used to store the contents of various registers, e.g., the ACC, B, GPR, PC, PSW and SFR. While the CPU automatically handles pushing and popping of the PC register to/from the stack, the ISR must store any other required register contents.

2.2.4.9 Interrupts Masking and Exception Handling

Exceptions are predefined interrupts designed to handle various, typically serious, fault conditions that can occur such as bus fault, memory management fault, program error, etc. PSoC5 provides support for 15 different types of exceptions and Non-Maskable Interrupts (NMIs). NMIs are not programmable in the general sense, but rather pre-defined ISRs designed to handle serious system faults.

Masking is a technique for blocking an interrupt, or group of interrupts, and includes:

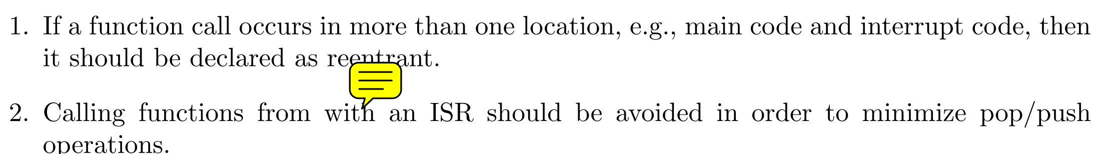
- BASEPRI - Specifying a specific priority level in the BASEPRI register are masked, i.e., blocked.
- FAULTMASK - setting a bit in the FALUTMASK register blocks all interrupts except for NMI.
- PRIMASK - setting a bit in the PRIMASK register blocks all interrupts except Hard Fault (3)and NMI (2).

Exception handling and NMIs are not explicitly supported in PSOC3.

2.2.4.10 Interrupt “Best Practices’

It’s important to exercise care when dealing with interrupts to avoid among other things introducing unnecessary latencies that degrade the embedded system’s responsiveness. The following represent some suggested guidelines that are often overlooked in developing program code that involves interrupts[28]:

1. If a function call occurs in more than one location, e.g., main code and interrupt code, then it should be declared as reentrant.



2. Calling functions from with an ISR should be avoided in order to minimize pop/push operations.

3. The status register should be read from within an ISR if the interrupt signal is a level shift.
4. ISR's should involve as little code as possible. (Setting a flag bit in the ISR and then checking its status from main code can significantly reduce latency)
5. In order for the 8051 (PSoC3) to service interrupts, the global interrupts enable bit (EA) in the Interrupt-Enable (bit 7 of IE) special function register (SFR 0xA8) must be set.
6. Before enabling an interrupt, the pending bit should be set to avoid unexpected ISR calls. The interrupt should also be disabled before dynamically changing the vector address and priority in software. After the configuration has been completed, the interrupt should then be enabled.

2.2.5 Memory

Up to 8K of Static RAM (SRAM) is employed in PSoC3 for temporary (volatile) data storage that can be accessed by both the DMA controller and by the 8051. Simultaneous access is also supported provided that there is no attempt to access the same 4K block. Flash is also used as nonvolatile memory for firmware, user configuration data, bulk data storage and optional Error Collecting Codes (ECC) data. Flash memory can be up to 64 Kbyte for storage of user program space.An additional 8K of Flash memory space is available for ECCs, but if ECCs are not used then this space is available to store device configuration and bulk user data. However, user code cannot be run from within the ECC memory space.

Current ECC technology is, in general, quite effective at correcting single bit errors which are the most common form of error . The ECCs used in PSoC3 are capable of detecting 2-bit errors in every 8 bytes of firmware memory. If an error is detected an interrupt can be generated to allow appropriate action to be taken. Flash output is 9 bytes wide with one byte reserved for ECC data.

2.2.5.1 Memory Security

Maintaining security of proprietary code is often a major concern and embedded controller such as PSoC3 and PSoC5 have mechanisms for preventing access to, and visibility of, such code and to prevent reverse-engineering, or duplication, of the intellectual property. Flash memory is organized as blocks of 256 bytes of program code, or data, and 32 bytes of ECC, or configuration data. Therefore up to 256 blocks are provided for 64 Kbyte of Flash.There are four levels of protection that can be assigned to each row of Flash as shown in Table **??**. Changing these levels of protection can only be accomplished by first imposing a complete Flash erase. The Full Protection mode allows internal reads to occur, but precludes external reads/writes and internal writes, which among other things prevents loading of code to download the internal code. In addition a fifth option is available called "Device Security" that permanently disables all test, programming and debug ports as a further security measure. While there may be no completely effective method for protecting code, the methods described here do represent the current state-of-the-art.

2.2.5.2 EEPROM

Byte addressable nonvolatile memory, consisting of 128 rows of 16 bytes each, is provided in the form of EEPROM that can be erased and written to at the row level and up to 2 KB of user data can be stored in EEPROM. At the byte level, random access reads can be carried out directly and writes are accomplished by sending write commands to an EEPROM program interface. It is not not necessary to suspend CPU activity while EEPROM writes are occurring. However, the

CPU cannot execute EEPROM code directly and there is no ECC hardware to secure EEPROM code integrity. In applications requiring ECC protection for EEPROM code, it is necessary to do so at the firmware level.

2.2.5.3 Interfacing External Memory

Many embedded system applications employing microcontrollers rely on external memory to meet a variety of data storage requirements. This usually involves the use of some form of External Memory Interface (EMIF). This interface makes it possible for the CPU to read from, and write to, external memories. PSoC3's EMIF functions in conjunction with I/O ports, UDBs and other hardware to provide the necessary external memory control and address signals. Eight or sixteen bit memory can be accessed in a memory space addressable by as much as 24 bits, i.e., 16 megabytes of memory.

The EMIF is compatible with four types of external memory:

1. Asynchronous SRAM
2. Synchronous SRAM
3. cellular RAM/PSRAM

and,

4. NOR Flash

The EMIF provides external memory control signals for synchronous memory, but not for the other forms of memory. Both 8- and 16-bit external memory can be accessed via either the XDATA memory space (PSoC3), or the ARM Cortex-M3 external RAM space (PSoC5). EMIF addresses can be one byte, two bytes or three bytes utilizing one, two or three of the ports shown in Figure 2.11. These ports are selected by configuring the 3-bit portEmifCfg field in the PRT*_CTL register which allows the least significant and middle byte or most significant byte of a three byte address to be assigned to a given port. However, the data transferred via the EMIF is restricted to either two either a single port or two ports. A particular data port can be selected as the path for either the most or least significant byte of the data. The fourth port is used to provide control utilizing 3-6 pins on the fourth I/O port. While unused pins on this port are available for other use, depending on the application, any unused address pins are not to be used for any other purpose. When the system is in Sleep Mode all of the EMIF registers retain their respective configurations.

EMIF clocking is derived from the bus clock which also servs as the clock for the PHUB and the CPU. This signal can be provided, as EM_CLOCK, to eternal memory at frequencies either equal to, one half or one quarter of the bus clock, i.e., the bus clock divided by a factor of 1, 2 or 4. However, the maximum allowable I/O rate for PSoC3 and PSoC5 GPIO pins is 33 MHz. In addition, the maximum bus clock frequencies are 67 MHz and 80 MHz for PSoC3 and PSoC5, respectively Therefore in most cases, EM-CLOCK will only be available for external memory at frequencies lower than the bus clock frequency.

2.2.6 Direct Memory Access (DMA)

As discussed in Chapter 1, minimizing latency is often a primary concern in the design of an embedded system because of the need for it to respond within certain time constraints to anticipated events and/or conditions. Furthermore, when dealing with data from, for example, a number of

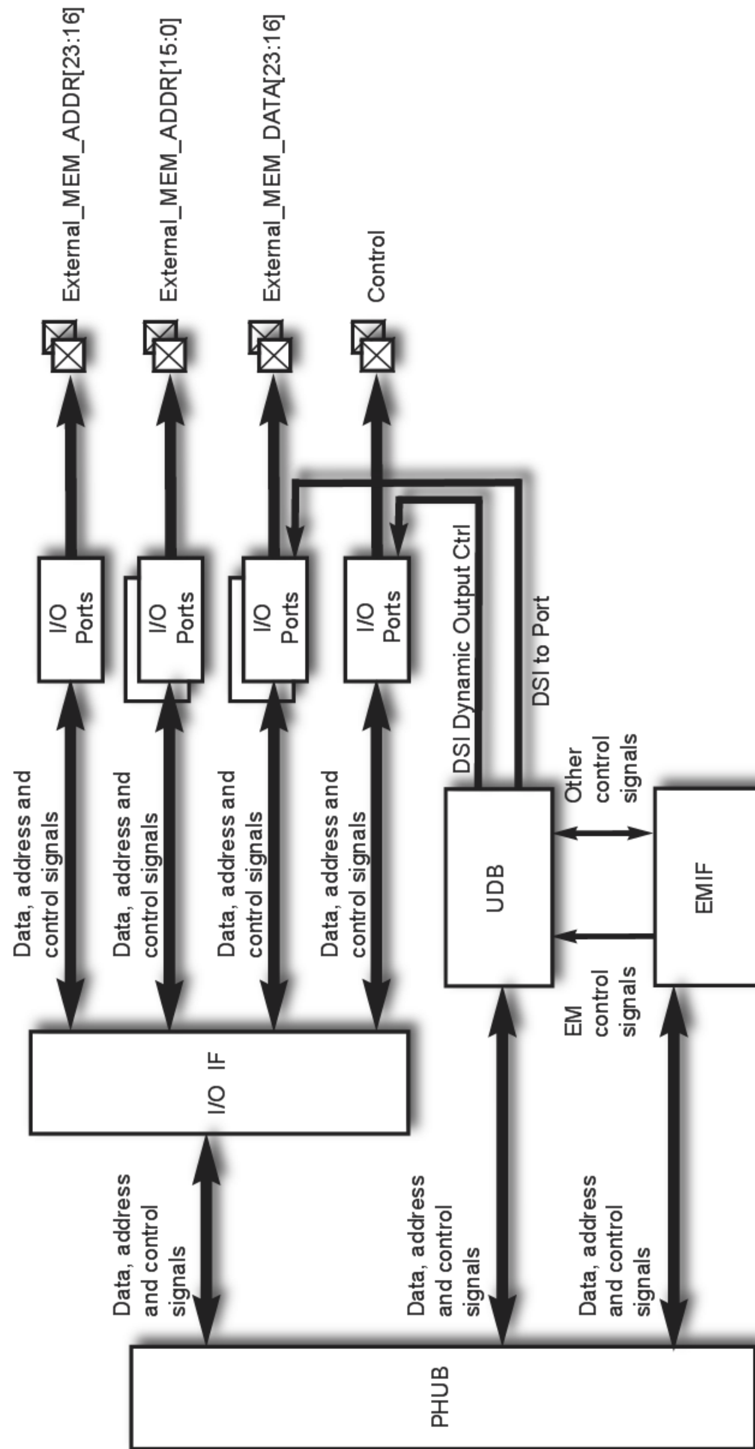


Figure 2.11: EMIF Block Diagram.emi

sensors, the rate of data collection may be much faster than the CPU's ability to process it, under some circumstances. Therefore, the ability to move data to/from an embedded system without incurring significant CPU overhead can an important concern when trying to minimize latency.

PSoC3 and PSoC5 have integral Direct Memory Access controllers that are capable of:

- memory-to-memory transfers
- memory-to-peripheral transfers
- peripheral-to-memory transfers
- peripheral-to-peripheral transfers
- supporting up to 24 independent DMA channels,
- handling data transfers that can be initiated, stalled or terminated,
- allowing multiple DMA channels, or transaction descriptors, to be chained, or nested, to perform complex tasks,
- assigning one, or more Transcription Descriptors[20] to each DMA channel for complex operations,
- allowing large data transfers to be split into multiple packets, varying in size from 1 to 127 bytes, which can be transferred in "bursts",
- supporting the triggering of DMA transfers by externally routed, digital signals via GPIO, by another DMA channel, or by the CPU,
- assigning one if eight priority levels (0 to 7) to each DMA channel.
- supporting up to 128 Transcription Descriptors, inclusive,

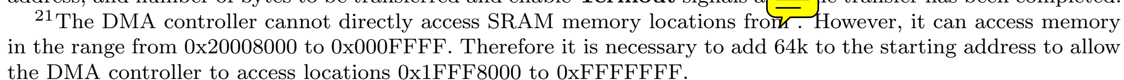
and,

- generating an interrupt (nrq) when a data transaction (DMA transfer) has been completed.

One technique for handling such requirements is to employ a combination of a DMA controller and a high performance bus over which peripheral access occurs and bulk data transfers, referred to as DMA transfers, take place and an associated controller. The PHUB is a combination of high speed bus, arbiter, router, and DMA controller together with radiating "spokes" each of which connects to a peripheral, as shown in Figure 2.12. The bus ports support 16, 24 and 32 bit addressing modes. Since both the CPU and DMA controller (DMAC) can initiate block transfers, the arbiter determines how such transfers are to be handled[21]. Spoke numbers, the number of peripherals supported by each, the spoke address widths and spoke data width are tabulated in Table 2.12.

Both the CPU and DMAC can act as masters and can initiate transactions on the bus. In the event that multiple requests occur, the arbiter in the central hub determines which DMA channel has the highest priority. If a higher priority transaction request occurs while a lower priority

[20] Transaction descriptors contain information regarding the transfer of data, e.g., source address, destination address, and number of bytes to be transferred and enable **Termout** signals a[illegible]he transfer has been completed.



[21] The DMA controller cannot directly access SRAM memory locations from . However, it can access memory in the range from 0x20008000 to 0x000FFFF. Therefore it is necessary to add 64k to the starting address to allow the DMA controller to access locations 0x1FFF8000 to 0xFFFFFFF.

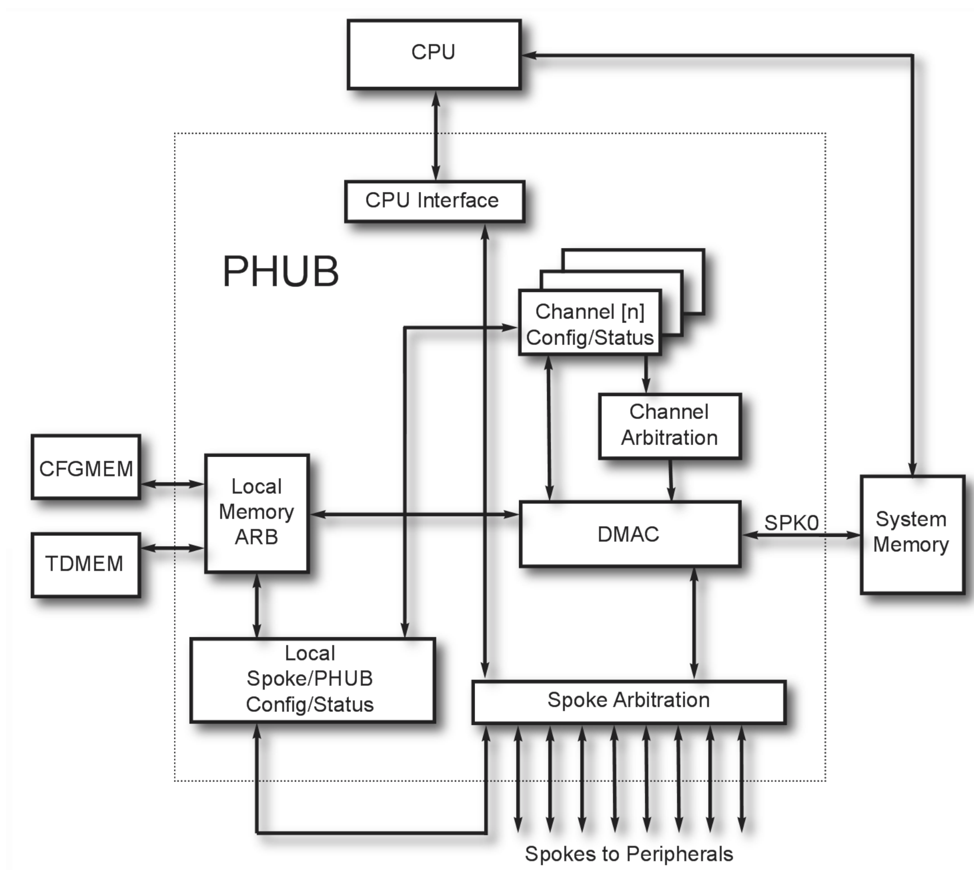


Figure 2.12: Block diagram of PHUB.

transfer is occurring, the lower priority transfer can be interrupted.The primary configurations of the PHUB are the number of DMA channels and spokes. The PHUB's architecture allows the CPU and DMAC to simultaneously access peripherals located on different spokes.

The PHUB employs two local memories referred to as CFGMEM and TDMEM, respectively. CFGMEM serves as the channel configuration memory which stores information for each record defined as CH[n]_CONF0/1 with one 8 byte set of each per channel with the result that CFGMEM is sized as 8 bytes x the number of DMA channels. CFGMEM is configured as an x64 memory to allow all 8 bytes of a CHn_CONFIG0/1 set to accessed on a single cycle to maximize DMA processing efficiency. TDMEM stores the transcription descriptor chains for a given channel and contains the DMAC instructions required for a DMA transfer via the channel. spokesSuch chains are considered as TDs, 8 bytes wide with a maximum of 128 TDs in TDMEM. The allocation of of TD chains is based on a given sequence that the DMA channel requires and in a maximal configuration. TDMEM can be a maximum of 8 bytes x 128 TDs, or 1 KB. If multiple bursts are required, the DMAC must keep track of where it was when it completed the last burst, while interleaving other channels' bus access. The intermediate TD states can stored either on top of CH(n)_ORIG_TD0/0 of the TD chain, or in CH[N]_SEP_TD0/1 to allow the chain to be preserved.

Table 2.12: Spoke Parameters

Spoke Number	Number of Peripherals	Spoke Address Width (Bits)	Spoke Data Width (Bits)
00	1	14	32
01	4	9	16
02	15	19	32
03	3	11	16
04	4	10	16
05	1	11	32
06	6	17	16
07	5	17	16
08-15	0	NA	NA

The DMA controller has five semi-independent functions that operate, in parallel, in a pipelined manner:

1. **ARB** - arbitrates between the various DMA requests regarding the DMA channels,
2. **DST** - handles data bursting via the Destination Spoke (DST)
3. **Fetch** - causes the transaction desciption (TD) and configuration (CONFIG) for a channel to be fetched when a channel wins arbitration,
4. **SRC** - causes data bursting on the Source (SRC) spoke for the channel,
5. **WRBAK** - the updated TD and CONFIG information is written back to their respective locations when the burst for a channel has ben completed.

2.2.7 Spoke Arbitration

The CPU and DMAC can access all of the spokes except for SPK0, which is the SYSMEM spoke, provided that they do not attempt to access the same spoke, contemporaneously. When the CPU and DMA are using different spokes there is no conflict. However, should the CPU and either the DMAC or the DST engines attempt to access the same spoke, the DMAC is required to allow the CPU to access the spoke first. The results of arbitration are a function of 1) the spoke's priority, 2) which attempted access first, 3) did both attempt access at the same time, and 3) whether the spoke in question is a CPU or DMA priority spoke which is determined by SPKxx_CPU_PRI(CFG_15:1). If the DMA engine is waiting for the CPU to finish its use of a spoke, it can still be subject to interruption by a higher priority DMA channel initiating the following sequence of events:

- The interrupted DMA channel completes any data in transit as may be required when it gains access to the spoke(s).
- The state of the channel is then saved by the DMA WRBACK function and the AUTO_RE_REQ bit for than channel is set. The setting of this bit results in the channel being returned to the DMA request pool subject to the normal arbitration rules.
- When that channel again has access to the DMAC, it simply resumes the "burst" where it left off.

While single requests are given immediate access, multiple requests must be subjected to arbitration subject to the following guidelines:

- PRIO is the highest priority and therefore not subject to arbitration.

2.2.8 Priority Levels and Latency Considerations

As mentioned previously DMA channels of higher priority can interrupt lower priority DMA transfers, i.e., those with a lower priority number, subject to the constraint that the lower priority transfer is allowed to complete its then current transaction. In cases for which multiple DMA access requests have occurred, a "fairness" algorithm is employed to minimize latency. This algorithm requires that the priority levels 2-7, inclusive, have at least some minimum percentage of the bus's bandwidth. If two requests are tied, then a simple round-robin method is employed to allow each to share half of the allocated bandwidth. However, this technique can be disabled for any of the DMA channels to allow that channel to always have priority. Table 2.14 shows the minimum bus bandwidth allocated for each priority level, once the CPU and DMA transactions of priority 0 and 1 have been completed. If the fairness algorithm has been disabled, then DMA access is based solely on their respective priority levels and without minimum bandwidth constraints.

2.2.9 Supported DMA Transaction Modes

The ability to chain transactions and the flexibility available in configuring each DMA channel; makes it possible to support simple, relatively complex and highly complex transaction modes, e.g.:

- **Auto Repeat DMA** - The same memory contents are repeatedly transferred.
- **Circular DMA** - Multiple buffers and TDs are employed with the last TD chained back to the first TD.

Table 2.13: Peripheral Interfaces to PHUB

Spoke Number	Peripheral Number	Short Name
00	00	SYSMEM
01	00	IOIF
01	01	PICU
01	02	EMIF CSR
01	03	EMIF DATA
02	00	PHUB LOCSPK
02	01	PM
02	03	CLKDIST
02	04	IC
02	05	SWV
02	06	EE
02	07	SPC
02	13	PM TRIM
02	14	BIST_ASSIST
03	00	ANALOG I/F
03	01	DECIMATOR
03	02	ANALOG I/F TRIM
04	00	FS USB
04	01	CAN
04	02	I2C
04	03	TIMERS
05	00	DFB
06	00	UDB SET0 8-BIT
06	01	UDB SET0 16-BIT
06	02	UDB SET0 CONFIG
06	03	UDB SET0 DSI
06	04	UDB SET0 CTRL
06	05	UDBIF
07	00	UDB SET1 8-BIT
07	01	UDB SET1 16-BIT
07	02	UDB SET1 CONFIGI
07	03	UDB SET1 DS
07	04	UDB SET1 CTRL

Table 2.14: Priority Level vs. Bus Bandwidth

Priority Level	% Bus Bandwidth
0	100
1	100
2	50
3	25
4	12.5
5	6.3
6	3.1
7	1.5

- **Indexed DMA** - This technique allows an external master to access locations on the system bus as if they were in shared memory.
- **Nested DMA** - Since the TD configuration space is memory mapped, one TD can modify another, e.g., a TD loads another TD's configuration and then calls that TD, when the second TD's transaction is complete, it then calls the first, which updates the second TD's configuration, with this cycle repeating as often as required.
- **Packet Queuing DMA** - packets are employed with specific protocols employing separate configuration, data and status phases for the transmission and receipt of data.
- **Ping-Pong DMA** - Double buffering is used to allow one buffer to be filled while the contents of the other are being transferred.
- **Scatter Gather DMA** - a transaction involving multiple, noncontiguous sources and/or locations for a given DMA transaction.

2.2.10 PSoC3's Clocking System

PSoC3's clocking generator provides the main/master times bases and allows the designer to make tradeoffs between accuracy, power and frequency. A broad range of clock frequencies are available due to the ability to accommodate multiple clock inputs and employ PSoC3's highly configurable internal clock distribution system. Table 2.15 provides a summary of the clock naming conventions.

PSoC3's internal clock generator can use internal/external clock sources[22], as shown in Figure 2.13, in the kHz and MHz range and input from Digital System Interconnects (DSI)[23] and an internal Phased-Locked Loop (PLL) can also be used for frequency synthesis.

In addition to support for multiple clock sources, there are eight individually sourced, 16-bit, clock dividers for the digital system peripherals, four individually sourced 16-bit dividers for the analog system peripherals and a dedicated 16-bit divider for the the bus clock.

[22]External clocks sources such as crystal oscillators are often used.

[23]DSI can provide clock signals created in UDBs, of-chip clocks routed through I/O pins and clock signals from the systems clock distribution resources.

Table 2.15: Clock Naming Conventions.

Clock Signal	Description
clk_sync	Synchronizing clock from the Master clock mux used to synchronize the dividers in the distribution.
dsi_clkin	Clocks that are taken as input into the clock distribution from DSI.
clk_bus	Bus clock for all peripherals.
clk_d[0:7]	Output clock from the seven digital dividers.
clk_ad[0:3]	Output clock from the four analog dividers synchronized to the digital domain clock.
clk_a[0:3]	Output clock from the four analog dividers synchronized to the analog synchronization clock.
clk_usb	Clock for USB block.
clk_imo2x	Output of the doubler in the IMO block.
clk_imo	IMO output clock.
clk_ilo1k	1 kHz output from the ILO.
clk_ilo100k	100 kHz from the ILO.
clk_ilo33k	33 kHz output from the ILO.
clk_eco_kHz	32.768 kHz output from the MHz ECO.
clk_eco_kHz	4-33 MHz output from the MHz ECO.
clk_pll	PLL output.
dsi_glb_div	DSI global clock source to USB block.

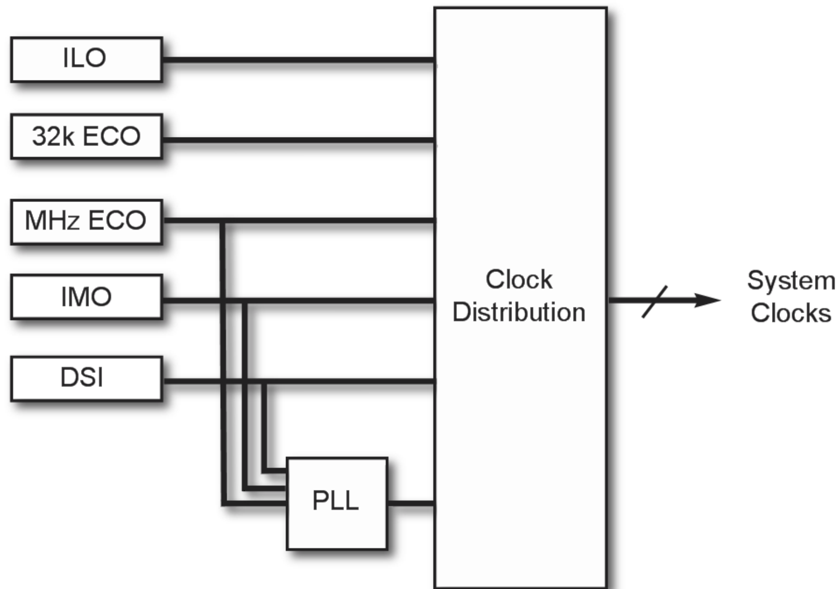


Figure 2.13: PSoC3 Clocking System

The primary clock sources consist of the following:

1. A fixed 36 MHz clock that routes to SPC,

2. A 4-33 MHz crystal oscillator,

3. A 3-67 MHz Internal main oscillator (IMO),

4. 12-67 MHz Doubler output source form from the IMO, MHz external crystal oscillator (MHzECO) or Digital System Interconnect (DSI),

5. 1 kHz, 33 kHz and 100 kHz Internal Low speed Oscillator (ILO),

and,

6. Digital System Interconnect from an I/O pin or other logic. 12-67 MHZ fractional Phase-Locked-Loop (PLL) driven by the IMO, MHzECO or DSI

If required, the internal PLL[24] can be used to synthesize frequencies in the range from 12-100 MHz. The PLL's input can be from IMO, a MHz crystal oscillator or from a DSI signal. As shown in Figure 2.14, the Master Clock Mux selects the IMO, DSI, PLL or MHz crystal oscillator as the primary clock source. Note that it is also possible to independently control the phase of the primary clock source for both digital and analog clocks, respectively. This arrangement also make it possible to change the clock source for the primary clock in multiple systems.

[24] An integral PLL prescaler(Q) and PLL divider (P) can be used to create clocks that are P/Q times the PLL's input frequency.

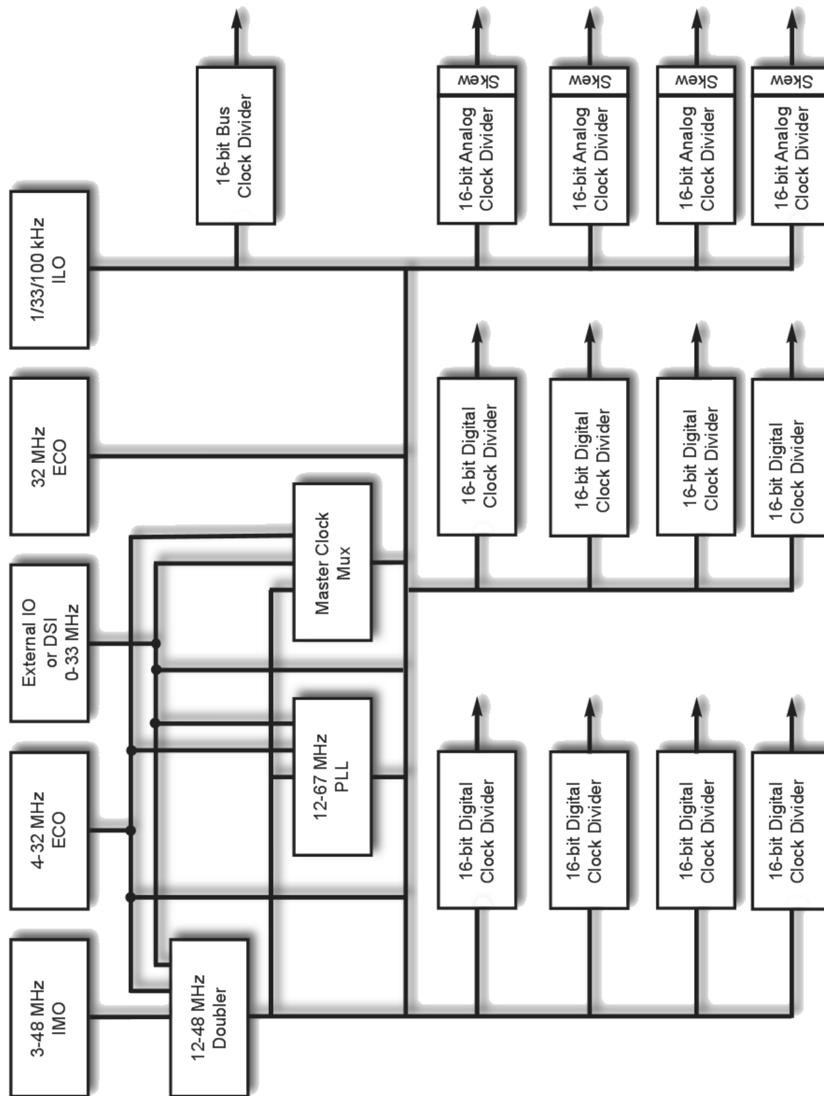


Figure 2.14: Clock distribution network for PSoC 3 and PSoC 5.

2.2.10.1 The Internal Master Oscillator (IMO)

The IMO produces a stable clock frequency without the use of any external components and contains a doubler circuit that provides an output that is twice the frequency of the IMO frequency, i.e., 6 to 24 MHz. However, the IMO output can be either the IMO's frequency or double that frequency, but not both.

Alternatively, other clock sources can be routed through the IMO, as shown in Figure 2.15, e.g., a DSI or MHz crystal oscillator output and hence through the IMO doubler[25]. As shown, IMO2X_SRC(PSoC3) selects either DSI or XTAL\CLK as a source for the clock signal. Thus DSI, XTAL or OSC (3,6,12,24,48 or 92 MHz) can then be selected by the clock mux (as determined by the (CLKDIST.CR) IMO_OUT register) following which the resulting clock signal (clk_imo) may be IMOCLK, IMOCLKX2 or 36MHz[26].

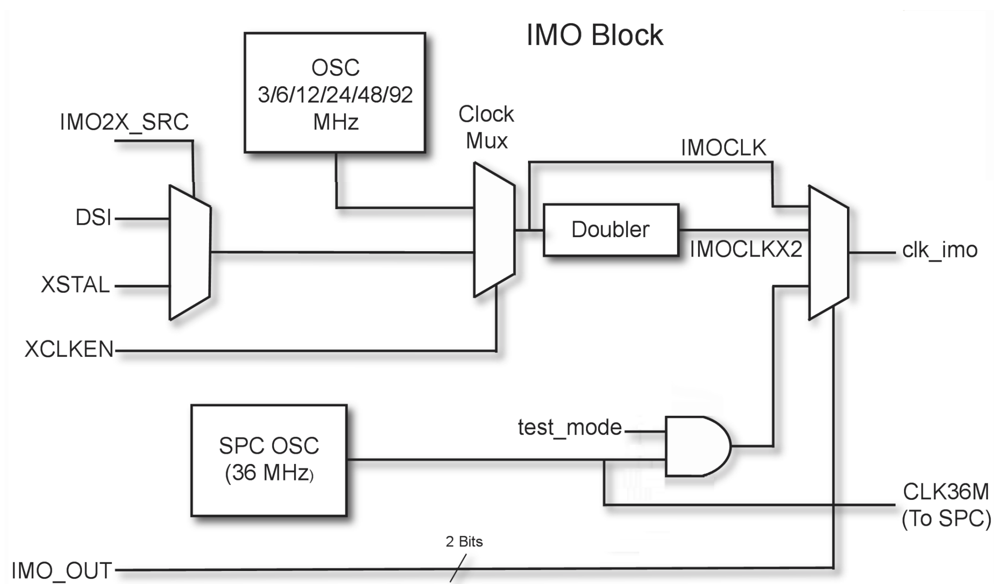


Figure 2.15: PSoC3's Clock Distribution System

The IMO block employs a precision input voltage and current to charge a capacitor from ground to a reference voltage. An integral comparator senses when a predetermined threshold voltage is reached and causes the charging cycle to repeat, between two capacitors, resulting in a pulse on each edge of the input clock and producing a clock frequency which is two times the input clock. (The AHB interface and registers for the IMO are implemented in the FAST Clock interface, i.e., logic for PLL, IMO and external oscillator).

2.2.10.2 Trimming the IMO

The IMO has provisions for "trimming" the clock frequency in terms of both gain and offset. The offset trim step size is determined by the gain setting. Factory trim settings are provided for the proper 24 MHz setting, since this frequency is used for USB operation. For non-USB

[25] To reduce power consumption the doubler can be disabled.

[26] This clock signal (36MHz) is routed to SPC and available to clock distribution only in test mode, that is it is not available in user mode. Its accuracy is approximately 10%.

operation the gain should be fixed to reduce power requirements. It is recommended that the gain setting employed be the same as the setting for 24 MHz. If it is necessary to change ranges, the offset trim should be loaded first at the lower frequency range, i.e., when moving to a higher frequency range, apply the new offset value and then change the range. Conversely when moving to a lower frequency range, change the range and then apply the new offset. Note that range and trim values take effect immediately.[27]

The ability to trim the frequency allows automatic "Clock-Frequency Locking" for USB operation to be employed so that small frequency variations of incoming USB signals can be corrected by comparing the incoming USB timing (frame markers) to the IMO clock rate[28]. Alternatively, a crystal controlled clock operating at 24 MHz "doubled" to 48 MHz could be used for Full Speed USB operation, or other crystal controlled frequencies could be employed in conjunction with the PLL to synthesize 48 MHz.

2.2.10.3 Fast-Start IMO

The IMO can also be operated in the Fast-Start IMO (FIMO) mode which is activated when "waking up" and provides a clock output within 1 microsecond after exiting the power down mode. In this mode the clock frequency is 48 MHz with an accuracy of about 10% of the primary IMO mode. The FIMO mode is selecting by setting the FASCLK_IMO_CR[3] bit in the IMO.CR register which cause the IMO clock to be replaced at the next wake-up. The FIMO mode is deselected by clearing the fimo bit resulting in the IMO clock replacing the FIMO in the next wake-up

2.2.10.4 Internal Low Speed Oscillator

The Internal Low Speed Oscillator (ILO) generates two independent clock frequencies, one at 1 KHz and the other at 100 KHz, respectively, neither of which require external components, as shown in Figure 2.16. In addition to operating independently of each other, they are not synchronized to each other and can be enabled/disabled independently or simultaneously. The 1 KHz clock is typically deployed as a "heart beat" timer and for the watchdog timer. The 100 KHz clock serves as a low power system clock and can be used to time sleep mode entry/exit intervals. Finally, a third clock frequency which is derived by applying "divide-by-three" to the 100 KHz clock. The power required, in terms of current, is in the 100nA to 1μA range, with an accuracy of .20% and 300μs start-up time.

2.2.10.5 Phase-Locked Loop

The PLL is capable of producing synthesized frequencies in the range of 12-67 MHz. The PLL uses a 4-bit input divider Q (FASTCLK_PLL_Q) to divide the reference clock, selected by the Mux as the IMO, an external crystal oscillator, or the DSI (an external clock signal) and an 8-bit feedback divider P (FASTCLOCK_PLL_P) to divide the output as shown in Figure 2.17. The outputs of the two dividers are compared by the phase frequency detector (PFD). The PFD compares the phase and frequency difference between the two signals to determine whether the signal fed back from the output is leading or lagging the reference signal F_{ref} defined by Equation 2.1. The PFD drives the output frequency, via "Up" or "Down" signals, either higher, or lower, as required and then it is "locked" producing an output frequency that is P/Q times the input reference clock. This PLL is capable of locking frequency within 10μ seconds, and once lock

[27]The clock may exhibit one cycle of slight variation.

[28]This will require, however, that the IMO frequency be 24 MHz and that the doubler be used to provided 48 MHz

has been achieved a bit (FASTCLK_PLL_SR[0]) is set and at that point the output frequency is available for distribution to the clock trees.

Thus:

$$F_{ref} = \frac{F_{in}}{Q} \tag{2.1}$$

$$clk_pll = F_{VCO} = F_{ref}(P) = \left[\frac{F_{in}}{Q}\right]P \tag{2.2}$$

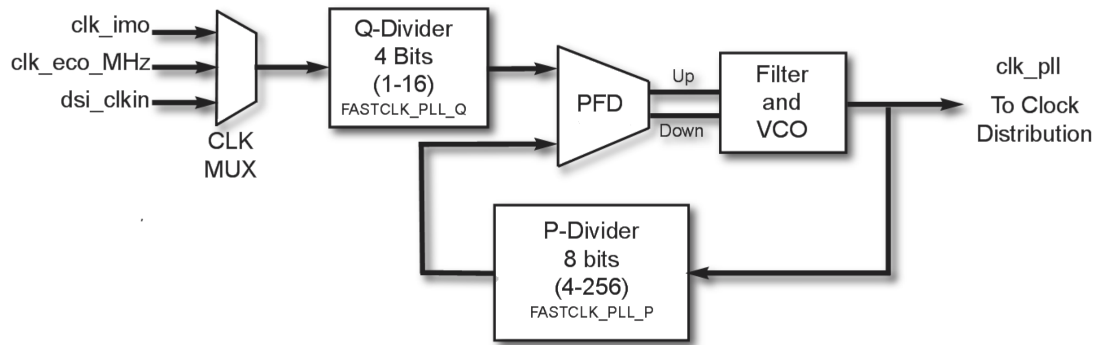


Figure 2.16: Internal design of the PLL.

In low power operation, during sleep and hibernate modes, the PLL must be disabled to allow "clean entry" into these modes of operation. Following wakeup and lock, the PLL can be enabled so that it can serve as the system clock. PSoC3/5 will not enter sleep or hibernate mode as long as the PLL remains enabled.

2.2.10.6 External 4-33 MHz Oscillator

Precision clock signals in the range from 4-33 MHz can be employed by adding an external fundamental mode, parallel resonance crystal and two capacitors as shown in Figure 2.18. The pins used for this purpose can also be used with standard I/O functions, e.g., GPIO, LCD and analog global, thus they must be tri-stated when used with an external crystal. The resulting signal is routed to the clock distribution network and may be routed to the IMO doubler, if the crystal frequency is in valid range for the doubler, i.e., less than, or equal to, 24 MHz. While this configuration is compatible with a wide range of crystals, crystal starup times are a function of crystal resonant frequency and quality. Oscillator settings can be matched to a given crystal by setting the xcfg bits of the FASTCLK_XMHZ_CFG0[4:0] register. The oscillator is enabled by FASTCLK_XMHZ_CSR[0].

Should the crystal oscillator fail, e.g., due to the adverse effects of moisture, or for some other reason, it is is possible to detect this condition by checking the clock error status bit, FASTCLK_XMHZ_CSR[7]. If the FASTCLK_XMHZ_CSR[6] bit is set and the crystal oscillator fails, then the crystal oscillator output is driven low and the IMO is enabled, assuming that it is not already running, and the output of the IMO is routed through the crystal oscillator output mux. Thus the system can continue to operate in the event of a crystal fault. When the system is in SLEEP/HIBERNATE mode, it is not necessary to allow the crystal oscillator to continue running and therefore consume power. The 32 kHz oscillator can be kept active if

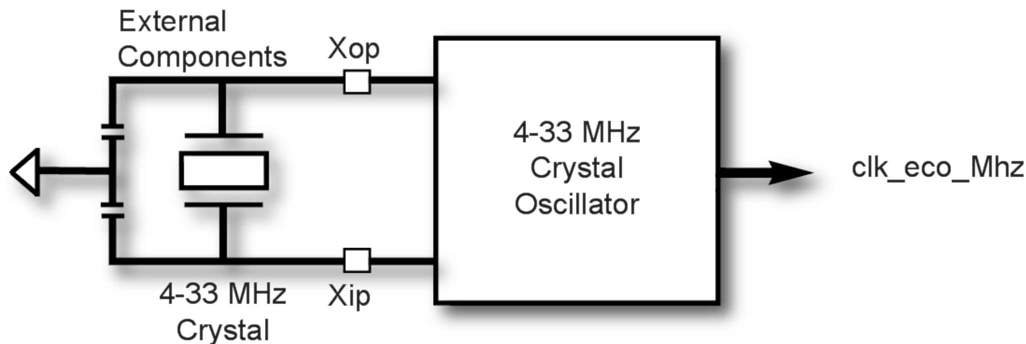


Figure 2.17: 4-33 MHz crystal oscillator.

precise timing is required, e.g., for the Real Time Clock (RTC). However, it is not possible to enter the SLEEP/HIBERNATE mode when the MHz crystal oscillator is running. One approach is to switch clock trees to the IMO source, and then disable the MHz crystal oscillator and the PLL, if it is active. It is then possible to enter a sleep mode. When the system wakes up from a sleep mode, the MHz crystal oscillator, and if necessary the PLL, can be enabled and employed, once stability has been achieved.

2.2.10.7 External 32 MHz Crystal Oscillator

The 32 MHz oscillator, kHzECO, utilizes a low cost, external crystal (32.768 kHz) and external capacitors to produces a precision timing signal, and serve as the basis for a real time clock operating at very low power, i.e., current levels less than 1μA. The resulting timing signal, clk_eco_Khz, is routed to the clock distribution network within PSoC3 and serves as a clock source for the clock distribution logic and the Real Time Clock (RTC) timer. Enabling/disabling of the kHzECO is accomplished by setting/clearing SLOWCLK_X32_CR[0]. This oscillator can operate at one of two power levels, depending on the state of the LPM bit, SLOWCLK_X32_CR[1], and the sleep mode status of the system.

The default mode is “Active” for the kHzECO and a hardware interlock forces the oscillator into its high power mode. which consumes 1-2 μA and minimizes noise sensitivity. Assuming that the LPM bit is set for low power mode, the oscillator only operates at low poer when the system when the system is in SLEEP/HIBERNATE mode. However, if LP_ALLOW (SLOW-CLK_X32_CFG[7] is set, the oscillator enters low power mode immediately when the LPM bit is set.

It should be noted that this oscillator is not stable when activated, and therefore some time is required for it to achieve stability. The DIG_STAT status bit, SLOWCLK_X2_CR[4], indicates that oscillation is stable by comparing it to the 33 kHz ILO signal. The ANA_STAT bit, SLOWCLK_X32_CR[5], use an internal analog monitor to measure the oscillator's amplitude.[29]

2.2.10.8 Implementing A Real Time Clock

Many embedded systems require the availability of a real time clock to time events, record time, data logging, etc. The kHzECO oscillator can be used to provide real time clock functionality by dividing the kHzECO signal by 32,768 to produce one pulse per second which in turn can be

[29]To avoid excessively long startup times before stability is achieved it is a good practice to start the oscillator in high power mode.

used to generate interrupts at one second intervals, update counters, etc., unless the system is in HIBERNATE mode.

2.2.10.9 Clock Distribution

The clock sources discussed previously produce signals that can be made available to other PSoC resources through the clock distribution logic within PSoC3 by routing them through analog and digital clock dividers. Some peripherals require specific clocks for their operation, e.g., the Watchdog Timer (WDT) requires the ILO. Clock distribution is facilitated by the use of clock trees. PSoC3 has four such trees for clock distribution, viz.,

1. Analog Clock tree
2. Digital Clock tree
3. System Clock tree

and,

4. USB clock tree

Eight dividers for the digital clock tree and four analog dividers for the analog clock tree are provided as part of the clock distribution system, as shown previously in Figure 2.14. Clock sources in each case are selected by an eight input mux for connection to the dividers and the outputs of the dividers are synchronized with their respective domain clocks. Distribution of sync clocks is facilitated by the Master Clock Mux and there are options that provided delay for the digital synch clock. All of the digital dividers are synchronized to the same digital clock, but the analog dividers can each be synchronized to their respective analog clock with different, or the same, delays.

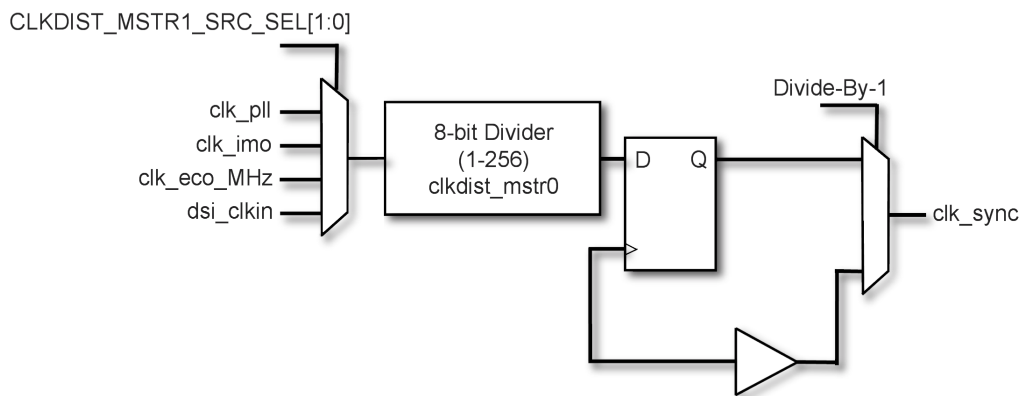


Figure 2.18: Master Clock mux.

The Master Clock Mux (MCM), shown in Figure 2.20, selects a clock from the available inputs, viz., PLL, IMO, ECO_MHz or DSI. The output of this mux becomes the source that is supplied to the phase mod circuit to produce skewed clocks selected by the digital and analog phase mux blocks. The MCM provides two re-sync clocks for the system: clk_sync_dig, for the digital clocks, and clk_sync_a for the analog system clocks. The Master clock, which is always the fastest clock in the system, is also the basis for switching the clock source for multiple clock trees simultaneously. Clock trees select the clk_sync_dig or clk_sync_a clock as their input for

systems that must maintain known relationships. An 8 bit divider is provided to make it possible to generate lower frequencies clocks, CLKDIST_MSTR0[7:0].

2.2.10.10 USB Clock Support

The advent of the now ubiquitous Universal Serial Bus (USB) has resulted in increasing support for it at the microcontroller level. PSoC3 provides the USB logic a synchronous bus interface while allowing that logic to operate asynchronously to process USB data.

The USB clock mux, shown in Figure 2.19, can be used to select the USB clock source as:

- imo1x
  - The 48 MHz DSI clock is subject to the accuracy of the clock.
  - Since the oscillator cannot operate at 48 MHz and therefore imo1x must be multiplied by the PLL to get 48 MHz.
- imo2x
  - 24 MHz crystal with doubler.
  - 24 MHz IMO and doubler with USB lock.
  - 24 MHz DSI with doubler.
- clk_pll
  - Crystal and PLL to generate 48 MHz.
  - IMO and PLL to generate 48 MHz.
  - DSI input and PLL to generate 48 MHz.
- DSI input
  - 48 MHz.

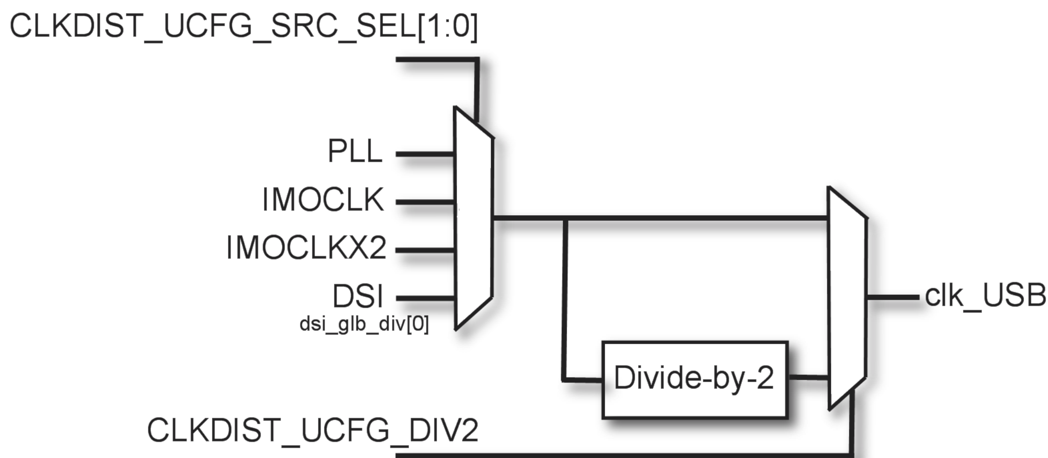


Figure 2.19: The USB clock mux.

If the internal main oscillator is selected, the oscillator locking function must be used to allow it to develop the required USB accuracy for USB traffic.This automatic clock frequency locking facility allows small frequency adjustments based on the incoming frame marker timing with respect to the IMO frequency. This type of clock frequency locking allows the clock frequency to stay within $\pm 0.25\%$ with respect to accuracy for the USB full speed mode.The locking mode is enabled by setting the FASTCLK_IMO_CR[6]. It is also possible to use a 24 MHz crystal controlled clock which is subsequently doubled to 48 MHz for full speed USB operation. Another option is to use other frequencies, e.g., 4 MHz, with the PLL to synthesize 48 MHz. In addition to the clk_imo option, the DSI signal, dsi_glb_div[0], can also be employed.

2.2.11 Clock Dividers

Clock dividers are an integral and important aspect of the clock distribution system. In addition, they also provide some control over the duty cycles. It is possible to to generate a single cycle clock pulse. A 50% duty cycle mode produces a clock with approximately a 50% duty cycle. A divider reloads its divide count after it reaches a terminal count of zero. The divider count is set in the CLKDIST_DCFG[0..7]_CFG0/1 register for digital dividers and the CLKDIST_ACFG[0..3]_CFG[0..3]_CFG0/1 register for analog dividers. The counter is driven by a clock source selected by an 8-bit mux controlled by CLKDIST_DCFG[0..7]_CFG2[2:0] for digital dividers and CLKDIST_ACFG[0..7]_CFG2[2:0] for analog dividers, in either a single-cycle pulse mode, or a nominally 50% duty cycle mode.

Regardless of the mode selected, a divide by zero causes the divider to be bypassed, resulting in a divide by one, and the input clock is applied to the output after a resynch, assuming that the sync option has been previously selected. If the loaded value is M, then the total period for the output clock is given by:

$$N = M + 1 \tag{2.3}$$

The CLKDIST_DCFG[x]_CFG2[4] or CLKDIST_ACFG[x]_CFG2[4] bit in the configuration register for each clock output is set high to enable the 50% duty cycle mode. However, it may not be possible to provide a 50% in all cases, because of dependencies on phase and frequency differences between the sync clock and the output clock.

2.2.11.1 Clock Phase

Another important clock parameter is phase. In addition to two duty cycle choices the outputs can be phase-shifted to go high after the terminal count, or at the half-period cycle. The default mode is known as "Standard Phase" and refers to the rising edge of the output, after the terminal count. Alternatively, "Early Phase' refers to the output being effectively shifted to an earlier point in time to an approximate count that is one half of the divide value. Setting the CLKDIST_DCFG_CFG2[5] or CLKDIST_ACFG_CFG2[5] bit in the configuration register for each clock will enable the Early Phase mode and the rising edge will occur near the half count point. While analog clock dividers are architecturally similar to digital dividers, they have an additional resync circuit to synchronize the analog and digital clocks. Synchronizing the digital and analog clocks facilitates communications between the digital and analog domains.

2.2.11.2 Early Phase

The clock outputs can also be phase-shifted by requiring them to "go high" after the terminal count or at the half-period cycle. The phrase "Standard Phrase" refers to the the rising edge of the output to occur after the terminal count. The Early Phase option allows the output to be viewed

as having been shifted to a point earlier in time for an approximate count that is one-half of the divide value. The Early Phase Mode can be invoked by setting the CLKDIST_DCFG_CFG2[5] bit, so that the rising edge occurs near the half-count point. While analog dividers are similar to digital dividers, they have an additional resynch circuit to synchronize the analog clock to the digital domain clock, thereby synchronizing the output of the analog dividers, called clk_ad, with the digital domain.

2.2.11.3 Clock Synchronization

Each of the clock trees can be set for one of the following options, with respect to output clocks:

- **Bypassed clock source** - If the divider value is set to zero and the synch bit is reset, The clock tree's selected source is routed to the output without division and results in an asynchronous clock.
- **Phase-Delayed clk_sync**, e.g., as clk_sync_dig - The tree operates at the same frequency as clk_sync but with the appropriate phase. In this case, the input clock source is ignored.
- **Resynchronized clock** - Activating the sync bit causes a clock at clk_sync/2 maximum frequency to be resynchronized by the phase-delayed clk_sync.
- **Unsynchronized divided clock** - This clock is asynchronous and occurs when the synch bit s reset, and the divider has a non-zero value.

2.2.12 GPIO

Modern microcontrollers make extensive use of buses which are analog or digital transmission paths, typically consisting of multiple conducting paths grouped together to facilitate digital and analog signal transmission, e.g., memory access depends critically upon the availability of high speed data and address buses to allow program code an data to move quickly and efficiently between , e.g, the CPU and RAM. Similarly buses are also needed for internal peripherals to allow communication to take place peripheral-to-peripheral, peripheral-to-CPU, CPU-to-peripheral, CPU-to-I/O, etc. Bus design varies, but they are typically a minimum of 8 parallel paths in width. Care must be taken in laying out such paths to assure that the electrical path length is the same for each path in a given bus, particularly as the speed of allowable bus traffic is increased. Since a large number of devices may have access to the same bus, tri-stating techniques, as described in Chapter 1, are often used to make sure that bus performance isn't degraded, to avoid collisions and to simplify bus use.

PSoC3 and PSoC5 make extensive use of buses and particularly of the analog interconnect, digital interconnect and system bus. The system bus allows traffic to be moved between the CPU, Memory and debug facilities and the digital/analog systems. The system bus is also used by the system wide resources. Routing of data along bus paths is also a common requirement and analog/digital multiplexers, and switches, are used to determine how bus traffic is to be routed.

Switches are functionally quite similar to multiplexer, since they are both based on analog switches, except for the fact that in the case of a multiplexers, while there may be "n" inputs there is only one connected to the output at any give time. However in the case of a switch, it is possible to have zero to "n" inputs connected to output at any given time. This is an important distinction and it should also be noted that in the case of a multiplexer fewer bits are required to connect an input to an output than is the case for a switch when both have the same number of inputs[30]. PSoC3 and PSoC5 have several analog routing resources, e.g., local analog

[30] For example, selecting one of 8 inputs requires only three bits for a multiplexer and eight bits for a switch

buses (abus), global analog buses (AGs), analog mux buses (AMUXBUS) and an LCD bias bus (LCDBUS). The analog globals and AMUXBUS connect to the GPIO's and provide a method of interconnecting GPIOs and the analog resource blocks (ARBs) such as DACs, comparators, switched capacitors, CapSense, Delta Sigma ADC and OpAmps. A voltage reference bus (V_{ref}) that provides precision reference voltages for the ARBs that are created by the precision reference block which is capable of generating precisions voltages and currents that are not a function of temperature.

Each GPIO pin can be connected to analog global path by use of a switch and it is possible to connect two pins on each port to the same global path. The analog global bus provides interconnection options via muxes and switches to the inputs/outputs of the following ARBs for I/O: CapSense (a virtual block), comparator, DAC, Delta Sigma ADC and Output buffer. Each GPIO pin has two analog switches, one to connect the pin to analog global and the other to connect to the AMXBUS. The control signals required to open, or close, these switches are invoked either by using the PRT[x]_AMUX and PRT[x]_AG registers which is the default option, or dynamically by using the DSI control that is connected to the input of the port pin logic block. However, before using the latter option, it is necessary to set a bit in the Port Bidirection Enable register, i.e., PRT[x]_BIE.

There are nine input/output ports consisting of seven General Purpose I/O (GPIO) ports, one SIO and one mixed-function port. This allows digital input sensing, output drive, pin interrupts, connectivity for analog input/output, LCD and access to internal peripherals to be supported directly via defined ports, or the UDB Individual I/O channels are arranged in groups of eight bits, or pins, and defined as the respective “ports”.

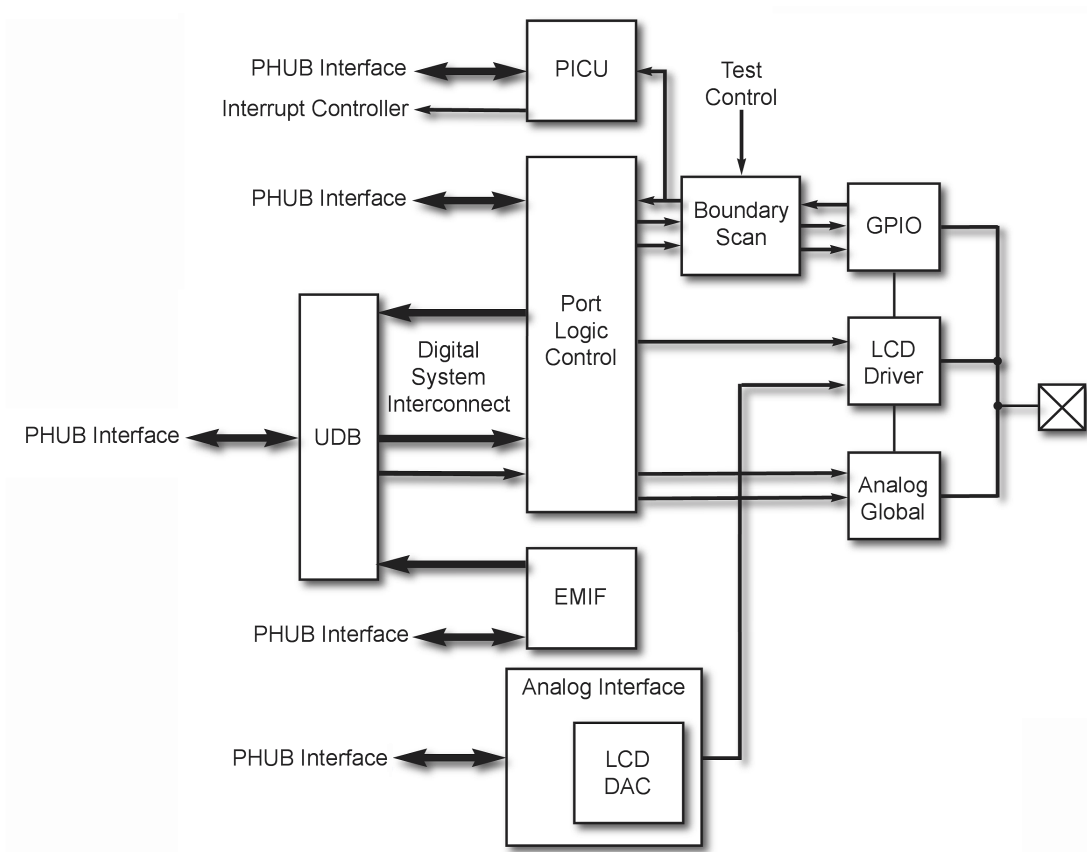


Figure 2.20: GPIO Block Diagram

2.3 Power Management

Power management in any embedded system is an important consideration in terms of maintaining the proper power levels, minimizing power consumption, proper distribution of power, minimizing noise and its effects in the supply lines, etc. PSoC3 and PSoC 5 maintain separate external analog and digital supply pins for the internal core logic. Two internal 1.8 voltage regulators are used to provide V_{ccd} for digital and V_{cca} for the analog circuitry. A sleep regulator is also maintained for operation in the sleep domain, an I^2C regulator for powering I^2C logic and a hibernate regulator for supplying"keep-alive" power to assure state retention when the system is in a hibernate mode. External connection the internal power distribution systems are made via pins labelled V_{dda} , V_{ddd} and V_{ddiox} for the analog, digital and and I/O power systems, respectively. Capacitors are required, as shown in Figure 2.22, preferably placed as physically close to their respective pins as possible. The capacitors are provided to minimize external power supply transients and to minimize adverse load effects. The digital and analog regulators are referred to as "active domain" regulators since they enter low power modes of operation in sleep mode. The sleep and hibernate regulators provide the necessary power when the system enters its lowest power consumption modes.

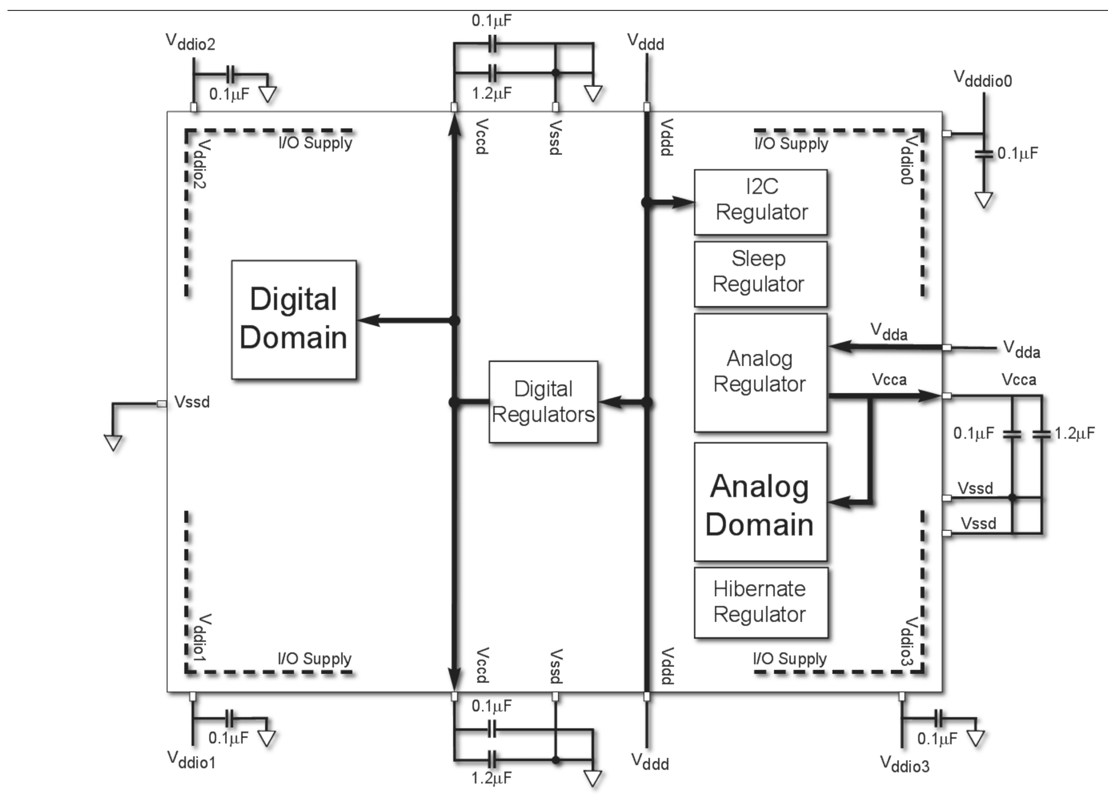
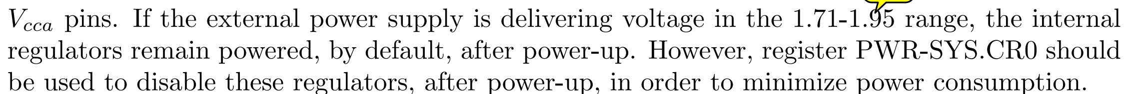


Figure 2.21: Power Domain Block Diagram

2.3.1 Internal Regulators

When operating in regions for which external power supplies provide voltages that range from 1.95-5.55 volts, the internal regulars draw power from these external supplies via the V_{ddd} and



V_{cca} pins. If the external power supply is delivering voltage in the 1.71-1.95 range, the internal regulators remain powered, by default, after power-up. However, register PWR-SYS.CR0 should be used to disable these regulators, after power-up, in order to minimize power consumption.

2.3.1.1 Sleep Regulator

When the system is in sleep mode, a sleep regulator provides a regulated voltage, V_{sleep}, for the 32kHz ECO, Central Timewheel (CTW), Fast Timewheel (FTW), ILO, , RTC Timer and watchdog time (WDT). The Hibernate regulator supplies keep-alive power, V_{pwrKA}, to those domains responsible for state retention during hibernation.

2.3.1.2 Boost Converter

PSoC3 and PSoC5 are capable of operating from voltage supplies over the range from 1.7 to 5.5 volts. However, external supplies may not be able to maintain a constant voltage to the system under some circumstances, e.g., systems using external supplies in the form of batteries can experience a wide variance in supply voltage as the battery system discharges, or as in the case of solar cells, the ambient illumination varies. Therefore PSoC3/5 have an integral boost converter that is capable of accepting input voltages over a wide range, e.g., as low as 0.5 volts and producing a constant output voltage at the required power levels. The internal converter requires an input voltage, an external inductor and external capacitors, , unless the external voltage is greater than 3.6 volts, in which case an external Schotty diode[31] is also required. In addition to being able to provide voltage for internal use, an external pin, V_{Boost}, is provided for driving voltages for external devices, e.g., an LCD.

PSoC3/5's boost converter can be disabled, or enabled, by setting, or resetting, the BOOST_CR1[3] and the output voltage can be changed by writing to the BOOST_CR0 [4:0] register. At startup the boost converter is enabled and by default the output voltage setting is 1.8V. If the boost converter is not to be used, then V_{bat} should be "tied to ground" and the IND pin should be left floating.

The following C language, code fragment illustrates how to start the boost converter, set its operating frequency at 100 kHz, and then stop it.[32]:

```
#include <device.h>
    void main()
        {
            Boostconv_1_Start();
            BoostConv_1_SelFrequency(BoostConv_1_SWITCH_FREQ_100KHZ);
            BoostConv_1_Stop();
        }
```

As shown in 2.24, when the boost converter's MOSFET is conducting, the voltage across the inductor is:

$$V_{input} = V_L = L\frac{di}{dt} \tag{2.4}$$

[31] Schotty diodes are named for Walter H. Schotty a German physicist (1886-1976) whose work led to the development of the hot carrier diode, also known as the Schotty diode. It has the important property that when conducting, the voltage drop across the diode is significantly lower than most diodes, viz., 0.15-0.45 volts versus 0.7-1.7 volts.

[32] Note thst the Boost Converter is by default "Active" when the system powers up

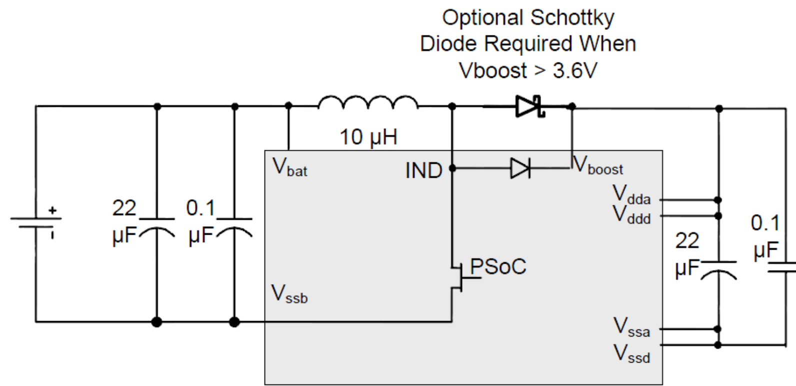


Figure 2.22: The Boost Converter.

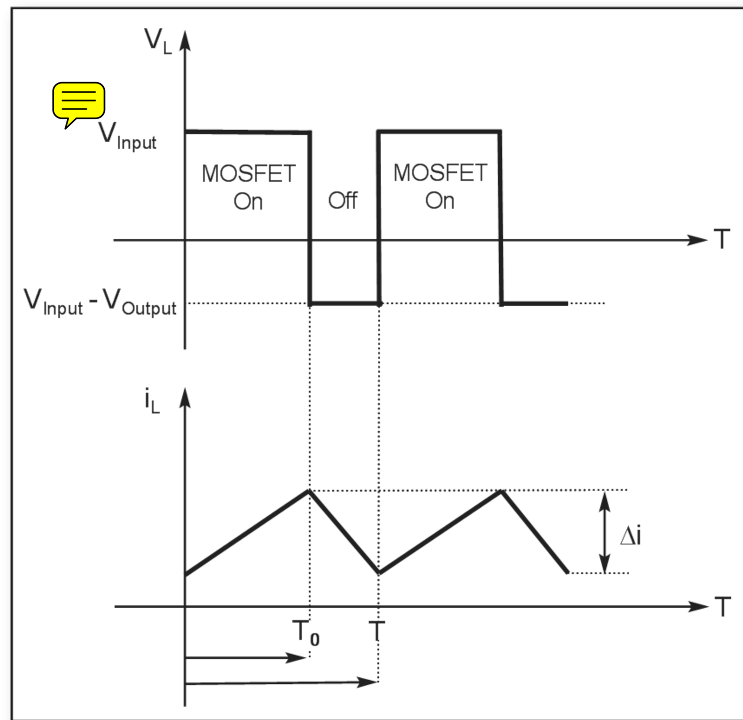


Figure 2.23: Boost Converter current flow characteristics.

and therefore,

$$\frac{di_L}{dt} = \frac{V_{input}}{L} = constant \tag{2.5}$$

The duty cycle is given by:

$$Duty\,Cycle = \frac{T_0}{T} = D \tag{2.6}$$

and therefore,

$$T_0 = DT \tag{2.7}$$

Equation 2.5 implies that[33],

$$\frac{di_L}{dt} = \frac{\Delta i_L}{\Delta t} = \frac{\Delta i_L}{DT_2} \tag{2.8}$$

and thus:

$$[\Delta i_L]_{on} = \frac{V_{input} DT}{L} \tag{2.9}$$

When the MOSFET is not conducting (MOSFET-Off state),

$$V_L = V_{input} - V_{output} = L\frac{di_L}{dt} \tag{2.10}$$

and therefore,

$$\frac{di_L}{dt} = \frac{V_{input} - V_{output}}{L} = \frac{\Delta i_L}{\Delta t} = \frac{\Delta i_L}{(1-D)T} \tag{2.11}$$

so that,

$$[\Delta i_L]_{off} = \frac{(1-DT)(V_{input} - V_{output})}{L} \tag{2.12}$$

But,

$$[\Delta i]_{off} + [\Delta i]_{on} = 0 \tag{2.13}$$

and thefore,

$$\frac{V_{input} DT}{L} + \frac{(1-D)T(V_{input} - V_{output})}{L} = 0 \tag{2.14}$$

so that,

$$V_{output} = \frac{V_{input}}{(1-D)} \tag{2.15}$$

Therefore when the MOSFET is conducting, energy is being stored in the inductor's magnetic field and the capacitor is supplying power to the load. Conversely, when the MOSFET is not-conducting, the energy supplied to the inductor, plus the additional input energy, is being supplied to the load and capacitor. It should be noted that in deriving Equation 2.15, that the inductors resistance, the diode's resistance and forward conducting voltage were assumed to be negligible and that the MOSFET functioned purely as a switch with insignificant resistance when conducting.

[33]Some will undoubtedly find the use of Δt and dt in the same expression disturbing, as well they should. However, this transgression does not adversely affect the calculation, or its conclusion.

2.3.1.3 Boost Converter Operating Modes

The boost converter can operate in one of three modes that are determined by the BOOST_CR0[6:5] register:

1. **Active -** In this mode the Boost regulator produces a regulated output voltage from a battery. The switching frequency of the Boost Converter is selected by the BOOST_CR[1:0] register. The available switching frequencies are 100kHz, 400kHz and 2MHz but are not synchronous with any other clock, i.e. these frequencies are "free running".
2. **Standby-** In the standby mode, only the band gap and comparators are active and other systems are disabled to reduce the power consumed by the Boost Converter.The Boost Converter's output voltage monitored continuously and supervisory data is available in BOOST_SR[4:0]. The supervisory data is referenced to the selected voltage.
3. **Sleep -** In this mode except for the band gap, the comparators and other circuits are turned oof. In this mode the output of the boot converter is a very high impedance, and the active circuits are powered by the energy stored in the 22μ capacitor. Over a prolonged period, the voltage across the capacitor will decay. This can be handled in some cases by awakening the system and recharging the capacitor.

Register	Function
PWRSYS_CR0	Regulator control
PWRSYS_CR1	Analog regulator control
Boost_CR0	Boost thump, voltage selection and mode select
Boost_CR1	Boost enable and control
Boost_CR2	Boost control
Boost_CR3	Boost PWM duty cycle
Boost_SR	Boost Status
RESET_CR0	LVI trip value setting
RESET_CR1	Voltage monitoring control
RESET_SR0	Voltage monitoring status
Reset_SR2	Real-time voltage monitoring status

Figure 2.24: Boost Converter register functions.

2.3.1.4 Monitoring Booster Converter Output

The status register BOOST_SR contains information regarding the input and output voltages of the boost converter referenced to the nominal voltage setting. The BOOST_SR[4:0] register provides the following status information:

1. Bit4: ov - above over-voltage threshold (nominal + 50 mv)
2. Bit3: vhi - above the high regulation threshold (nominal +25 mv)
3. Bit 2: vnom - above nominal threshold (nominal)
4. Bit 1: vlo - below low regulation threshold (nominal to 25 mv)
5. Bit 0: uv - below under-voltage limit (nominal to 50 mv)

2.3.1.5 Monitoring Voltages

Two circuits, shown in Figure 2.26, are provided for monitoring voltages to detect any deviation from the selected thresholds for external analog and digital supplies:

1. **Low Voltage Interrupt** (LVI) - this circuit generates an interrupt when it detects a voltage below the set value. The low voltage monitors defaults to the off mode. However, the trip level for the LVI is set in the RESET_CRO register over a range from 1.7-5.45V, in steps of 250mv.

2. **High Voltage Interrupt** (HVI) - this circuit generates an interrupt when it detects a voltage above the set value.

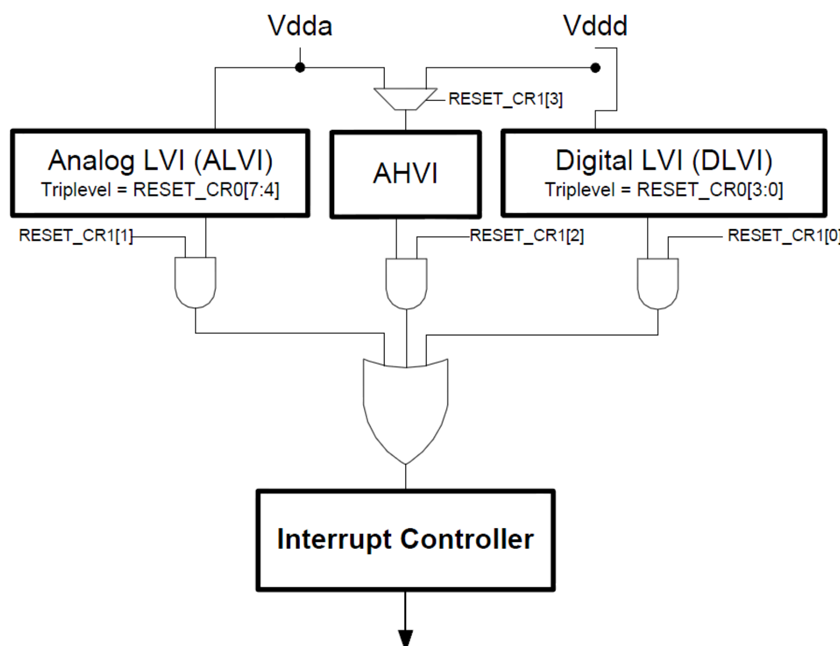


Figure 2.25: Voltage Monitoring Block Diagram

2.4 PSoC3 Debugging

The architectures for PSoC3 and PSoC5 include a Test Controller (TC) that provides access to pins for boundary scanning and to memory/registers via either PSoC3's Debug On-Chip module, or PSoC5's Debug Access Port (DAP) which supports functional testing, programming and programm debugging. Connection to the PSoC3 debugging is facilitated by the availability of Debug-On_Chip (DOC) and the Single Wire Viewer (SWV). The DOC serves as the interface between the CPU and the Test Controller (TC) and is used to debug, trace code execution and for trouble shooting device configuration.[34]

[34]DOC is used for PSoC3 and Cypress' Semiconductor's CY8C38 family of devices. Debugging for PSoC5 is accomplished by using ARM's Coresight components for debug and traces. SWV targets resident code to provide diagnostic info through a single pin.

The test controller serves as a physical interface between a debugging host, and PSoC3 and PSoC5 debug modules and connects to the host via either JTAG or SWD. JTAG support for PSoC3 and PSoC5 exceed the IEEE 149 standard in terms of the access provided to instructions and registers. In the case of PSoC3, the test controller translates JTAG instructions/registers or SWD accesses to register accesses in the DOC module as indicated schematically in Figure 2.27

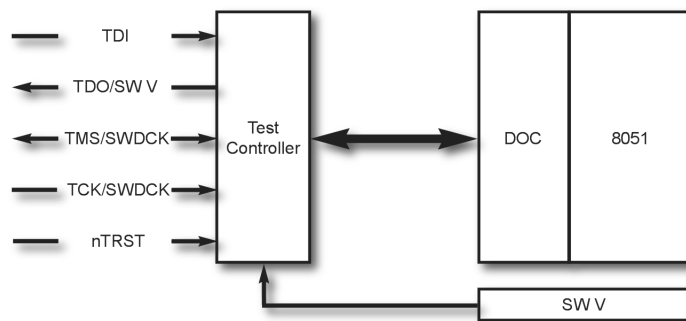


Figure 2.26: Test Controller for PSoC3 (8051) block diagram.

The DOC has a number of important features:

- It can take control of PSoC3's CPU (8051) and access any address accessible by the CPU via the PHUB interface. This capability includes the CPU's internal memory, SFRs and PC.
- The DOC can HALT the CPU and single step through instructions.
- Breakpoint capabilities of the DOC include setting as many as 8 program address breakpoints, setting one memory access breakpoint and setting a Watchdog trigger breakpoint.
- Trace capability includes: tracing the PC, ACC and a single byte from the CPU's internal memory or SFRs; 2048 instruction trace buffer for the PC; 1024 instruction trace buffer for PC, ACC and a single SFR/memory byte; operating in a triggered, continuous or windowed mode; CPU halt or overwrite of the oldest trace when the trace buffer is full and when not tracing the trace buffer is available for other use.

The SWV provides:

- either Manchester or UART for output,
- a simple, efficient packing and serializing protocol,

and,

- Thirty two stimulus port registers

PSoC3 supports three debugging/testing protocols for communicating with PSoC3:

1. JTAG[35]
2. Parallel test mode (PTM) and,
3. Serial Wire Debug (SWD) - this protocol allows a designer to debug using only two pins of the PSoC3 device.[36]

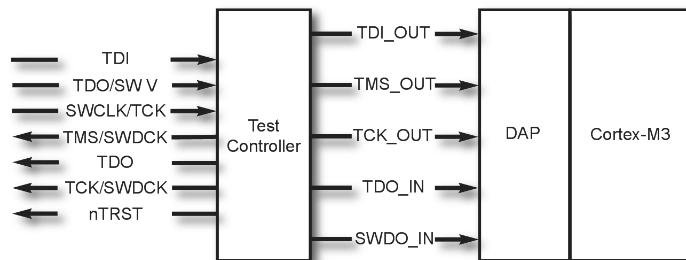


Figure 2.27: Test controller configuration for PSoC5.

DOC functionality is controlled by accessing registers within DOC. However, these registers are only accessible trough the TC interface and not through PHUB. A debugging session utilizing DOC, requires that the CPU enable debugging. Debugging commands are sent to the TC by JTAG or SWD and from there to DOC as shown in Figure 2.29. Addresses transmitted in this manner are used to access TC registers, DOC registers or alternatively, these addresses are sent on to the DOC memory interface. Within the DOC are a number of memory interfaces and an incoming address is decoded and forwarded to the correct memory interface address output.The DOC waits until the memory access has been completed the DOC transmits a signal to the TC that either the write is complete, or that data from a read command is available.

The DOC is able to take over control of the CPU memory interfaces and carry out reads and writes to memory as if the actions were CPU based. Flash, CPU internal memory, CPU SFRs and the CPU's external memory and registers and the PC can all be accessed by the DOC. Reading an writing to these resources is based on the addresses shown in Table 2.17. In the case of reading or writing to the PC it is necessary to first halt the CPU.

Table 2.16: PSoC Memory and Registers

Address Range	Description
0x050000 - 0x0500FF	CPU Internal Memory
0x050000 - 0x0500FF	CPU PC (16-bit register)
0x050000 - 0x0500FF	CPU SFR Space
0x050000 - 0x0500FF	TC and DOC Registers
All Other Addresses	CPU External Memory/Registers

2.4.1 Breakpoints

Breakpoints are a useful tool in analyzing and diagnosing program execution issues, particularly in light of the fact that it is possible to allow the program to operate at normal execution speed before being halted at a breakpoint. PSoC3 has support for eight, program address breakpoints, a memory access breakpoint and a watchdog trigger breakpoint. Program address breakpoints

[35]PSoC3 complies with IEEE 1149.1 (JTAG Specification)

[36]These two pins, once designated for SWD debugging, must be reserved for debug use only and may not be used for any other purposes.

employ eight registers, DOC_PA_BKPT0 - DOC_PA_BKPT7. Setting an address breakpoint requires that the address for the breakpoint must be stored in bits[15:0]

2.4.2 The JTAG Interface

One of the most popular interfaces for testing integrated circuits (ICs)such as PSoC3 and PSoC5 was developed by the Joint Test and Action Group (JTAG) as a method for controlling and reading an IC's pin values. . The JTAG interface includes the following signals: Test Data In (TDI), Test Data Out (TDO), Test Mode Select (TMS) and a clock signal(TCK). This configuration makes it possible to test multiple ICs on a given board in a daisy-chain manner.)

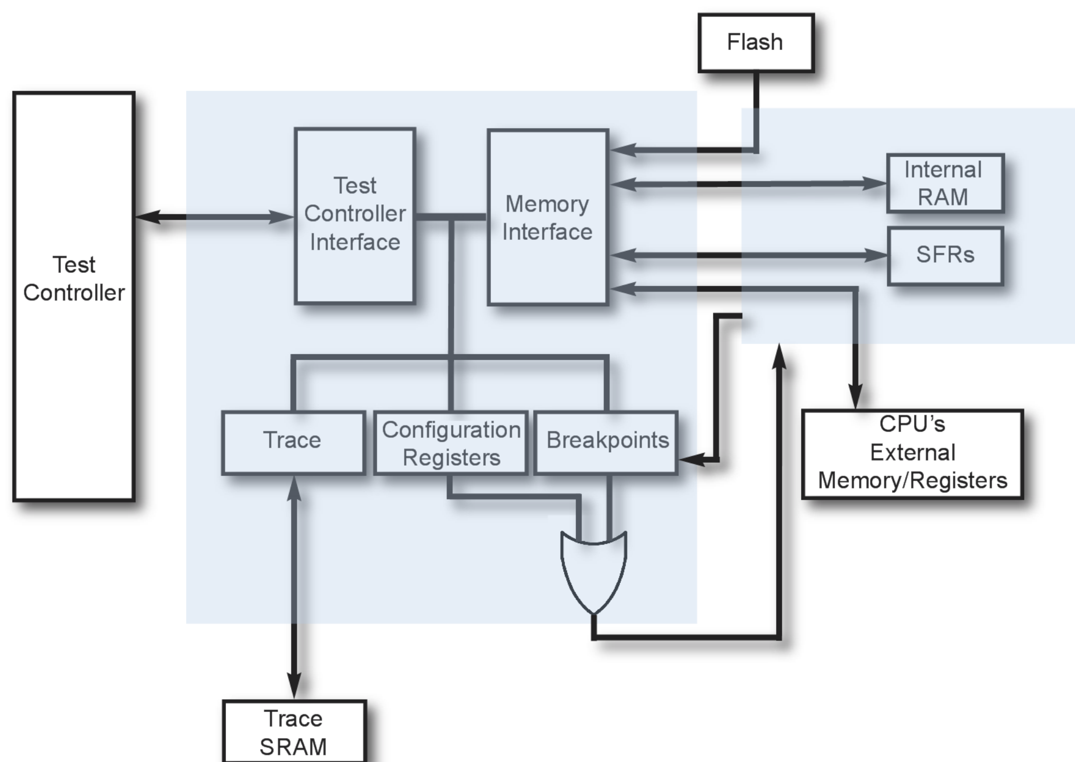


Figure 2.28: DOC,CPU and TC Block Diagram

Summary: In this Chapter, discussion has focused on subsystems using PSoC3 and PSoC5 as illustrative examples of some of fundamental aspects of current microcontroller architectures. Included were detailed discussion of the 8051 instruction set, the wrapper concept as employed in PSoC3 to integrate an 8051 core into the PSoC environment, basic concepts of interrupts and interrupt handling, DMA transfer concepts including using various DMA functions in conjunction with a peripheral hub for transferring data and to/from peripherals, clock sources and clock distribution, internal and external memory use, power management, sleep/hibernate considerations, implementation of a RTC, hardware testing and debugging, etc. In the following chapters, discussion will focus on a microcontrollers digital and analog peripherals, the development environment and modules such as delta-sigma converters, PWMs, OpAmps, etc. and finally conclude with a detailed implementation of a digital voltmeter,

Chapter 3

System and Software Development

3.1 Realizing the Embedded System

The development of embedded systems based on hardware platforms such as PSoC3 and PSoC5 requires a design process that begins with a concept and ends with the completed application. While there is not any one "best way" to carry out such a process there are a number of widely adopted models for this type of activity, e.g.,

- Waterfall - a series of sequential steps, as shown in Figure 3.1, viz., development of a

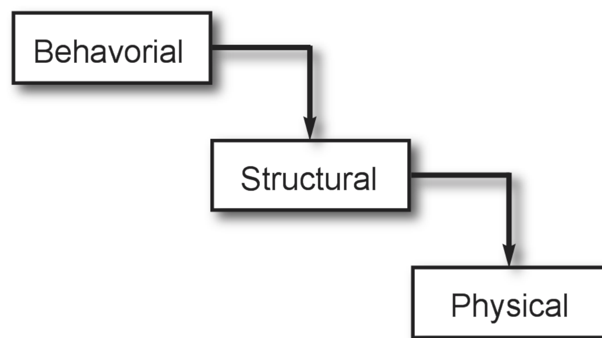


Figure 3.1: Waterfall Design Model

specification, creation of a preliminary design (behavorial), development into a detailed design (structural) and full implementation (physical).[1]

- Top-down - the design progresses from an abstract description to a specific design.
- Bottom-up - begins at the lowest level with individual modules or components and evolves as an aggregate to form a larger system.
- Spiral[2] - can be described as a combination of both top-down and bottom-up methodologies. Design begins with a minimal configuration which is then iterated through incorporating an additional feature or features, then tested, evaluated, followed by the addition of more features and the iteration continued, tested and evaluated until the completed design emerges, as shown diagrammatically in Figure 3.2.[11]

[1]The transition to the next step requires prior completion of the preceding step.
[2]This model is particularly useful when requirements are changing during the design process.

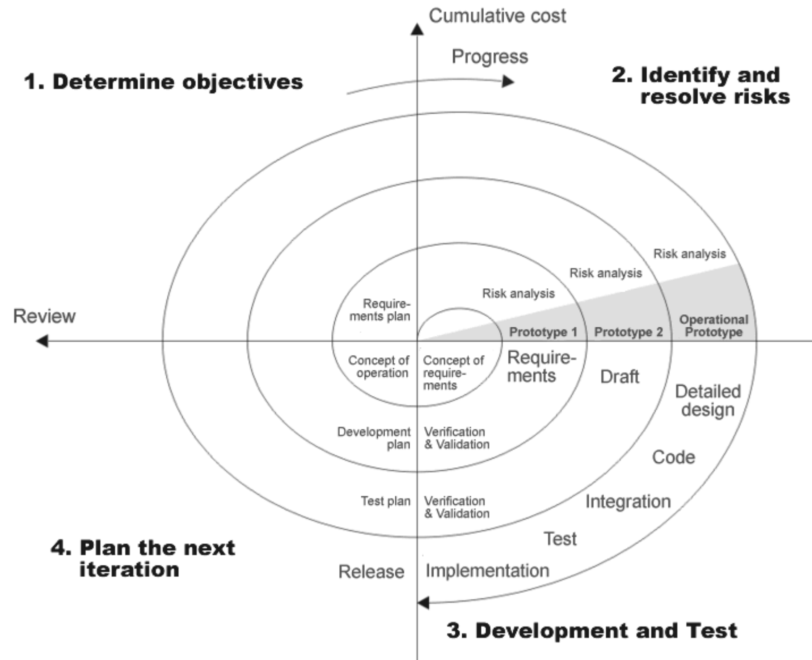


Figure 3.2: The Spiral model.

- V-Cycle[3] - allows testing to occur early in the life of the project and affords an opportunity to discover faults in the design earlier in the design process. The lefthand side of the V represents the definition and decomposition process and the righthand side represents the verification and integration processes, as shown in Figure 3.3.

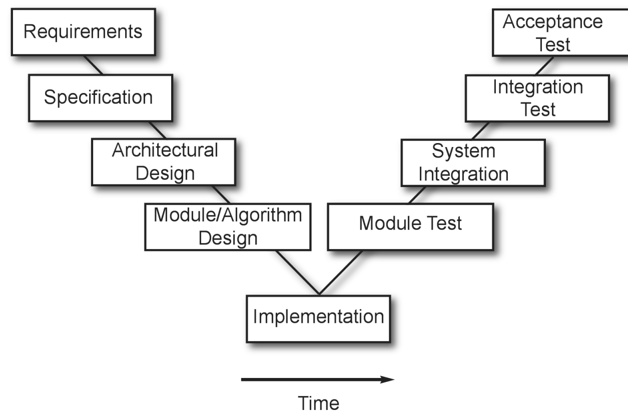


Figure 3.3: The V-Model.

[3] Also referred to as the validation and verification model.

Each of these models is described generically as a *Life-Cycle Model*[4] and each offers certain advantages and disadvantages. Regardless of which a designer may choose, there are an underlying set of principles and steps that serve as the foundation for each and are discussed in the rest of this chapter, in some detail. The models discussed originated largely as software development models, but because the line drawn between hardware and software can often, in some respects at least, be regarded as arbitrary, they are quite applicable to the design process for embedded systems.

3.2 Design Stages

In setting out to design an embedded system, the first step is to define the physical system that is to be controlled and determine from the associated requirements, a set of specifications[5], for the system. It is then possible to draw a functional block diagram of the system and from that point to derive a schematic. The schematic can then be used to create a signal flow diagram, an associated block diagram, or a state-space representation of the system. This results in either an open- or closed-loop system that can be implemented, and tested, to determine whether or not it conforms to the requirements and specifications. It is common for embedded systems that are intended to control a system, or process, to employ negative feedback loops to assure that the control system remains stable and/or to allow the system to be "self-correcting". Open-loop systems can result in drift away, or in some cases rapid departure from, the desired operating (set) points[6] and conditions for the controlled system, or process. Also, as discussed in a previous chapter, system disturbances must be taken into account, as well, even for well designed embedded systems.

Parameters to be considered, when designing embedded systems, include:

- variables that are to be controlled,
- variables that are to be manipulated,
- variables that are associated with disturbances,
- controller output variables,
- error signals variables,
- internal set points employed by the controller,
- external set points associated with the controlled system or process,

and,

- process or system variables that are to be controlled.

Note that in addition to these variables, their respective rates of change may also be important variables, e.g., derivatives of first and higher order. One of the figures of merit of an embedded system is robustness, which is defined in terms of sensitivity[7], and the system's ability to maintain set points, in spite of external disturbances.

[4]There is a fourth design methodology referred to as the "big bang" model, development proceeds for some period of time in relative isolation and is then released in the feverent hope that, with any luck at all, it will provide acceptable.

[5]In the present context. the term *requirements* refers a description of properties required in order to meet a set of needs whereas the term *specification* refers to a description of a system capable of implementing those properties.[54]

[6]Set points are the desired, or target, values for a controlled system, or process, that a controller is to maintain, well within acceptable limits, e.g., a process temperature, flow rate, angular velocity, etc.

[7]Sensitivity, in the present context, is a measure of an embedded systems ability to perform as required in the presence of external disturbances and it is defined as the ratio of the relative change in steady-state output to the relative change of a system parameter.

There are various approaches/techniques for modeling systems, e.g.,

- Deterministic versus stochastic
- Linear versus non-linear
- Continuous-time versus discrete-time
- Time-invariant versus time-variant

Systems that employ feedback, and therefore have outputs that depend on previous variable values, are often modeled as a set of differential equations for which the independent variable is time. Such system can then be mapped into the frequency domain and represented analytically in the form of transfer functions making it relatively easy to study the systems stability. Non-linear systems can be considerably more challenging than linear systems in that, as noted by Poincaré , "... it may happen that small differences in the initial conditions produce very great ones in the final phenomena. A small error in the former will produce an enormous error in the latter. Prediction becomes impossible...".[57]

3.3 Signal flow and the schematic view of the system

A signal flow graph is simply a graphical representation of nodes that are interconnected by several directed branches and represent variables such as inputs, outputs, etc. A directed branch illustrates the dependence of one variable on another, the gain associated with each branch and the signal flow direction. The default value for gain is unity, and the allowed direction of signal flow is defined by the direction of the arrow on each branch.

A *path* is defined as any branch, or continuous sequence of branches, that can be traversed in moving from one given node to a second given node. Two loops are said to be *non-touching,* if they do not share a common node. Branches that share one or more common nodes are said to be *touching*. Any path that begins and ends on the same node is referred to as a *loop*. A *forward path* is defined as a path from a *source* to a *sink*. The *gain of a path* is defined as the multiplicative product[8] of the gains of each of the branches that are part of the path. Figure 3.4 illustrates examples of some commonly encountered block diagrams, and the associated signal graphs.

Thus a signal graph, in the simplest terms, is just a graphical representation of a set of linear relationships and, as such, is only applicable to linear systems[9] [57]. It is also referred to as a *directed graph* because the direction of signal flow is indicated by the arrows in each branch. The nodes in the signal graph represent the variables of a set of linear equations representing the system, as shown for example in Figure 3.5. Note that a_{11} and a_{22} are non-touching, *self-loops* and the loop formed by a_{12} and a_{21} is also a self-loop.

The pair of linear equations shown in this figure can be rewritten as

$$(1 - a_{11})x_1 - a_{12}x_2 = b_1 \tag{3.1}$$

$$-a_{21}x_1 + (1 - a_{22})x_2 = b_2 \tag{3.2}$$

[8]If the gain of each branch is expressed in terms of dB, then the overall gain, as expressed in terms of dB, is the summation of the dB gain of each branch.

[9]However, if the parameters in a non-linear system can be considered sufficiently small, the system can be *approximated* by a set of linear equations. If a non-linear system is constrained to operate in a linear region only, it may be possible to employ the techniques described in this chapter that would otherwise be reserved only for truly linear systems. The reader is cautioned, nonetheless, that ignoring non-linear aspects of any system can lead to unintended consequences, e.g., chaotic behavior, as shown by Poincaré in 1908.

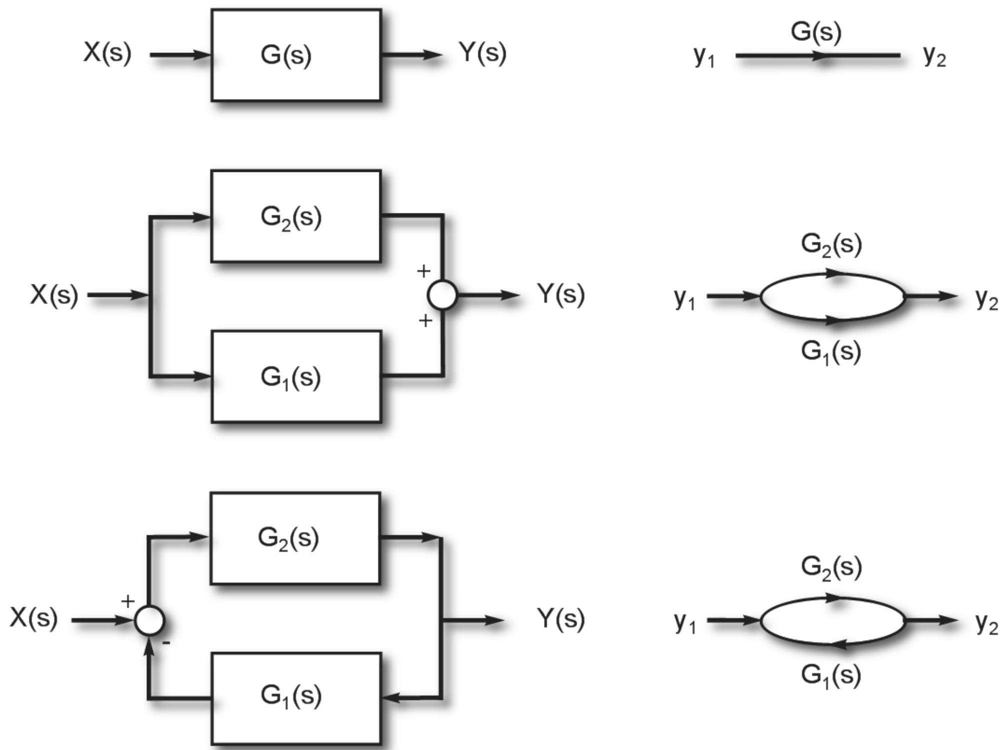


Figure 3.4: Block diagrams and their respective SFGs.

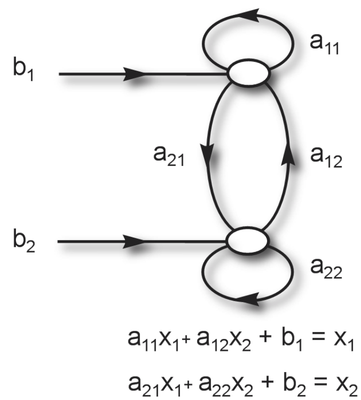


Figure 3.5: A simple signal path example and the associated linear equations.

and solving for x_1 and x_2 yields

$$x_1 = \frac{(1 - a_{22})}{\Delta b_1} + \frac{a_{12}}{\Delta b_2} \tag{3.3}$$

$$x_2 = \frac{(1 - a_{11})}{\Delta b_2} + \frac{a_{21}}{\Delta b_1} \tag{3.4}$$

where

$$\Delta \stackrel{def}{=} \begin{vmatrix} (1 - a_{11}) & -a_{12} \\ -a_{21} & a_{22} \end{vmatrix} = determinant \tag{3.5}$$

$$= 1 - a_{11} - a_{22} + a_{22}a_{11} - a_{12}a_{21} \tag{3.6}$$

Using so-called "block rules", it is sometimes possible to substantially simplify a block diagram of a system before attempting to create the signal path graph. Several of these rules are shown in Figures 3.6, 3.7 and 3.8. Thus, as illustrated by this simple example, it is possible to begin with a graphical representation of signal flow for a particular system and develop therefrom an analytic representation of the system, in a straightforward manner.

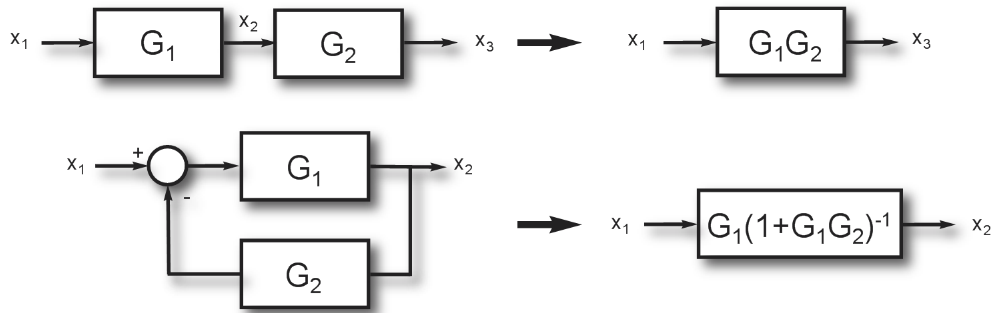


Figure 3.6: Reduction of two blocks to one.

3.3.1 Mason's Rule[10]

An important parameter of a system, such as those discussed in the previous section, is its overall gain [27] which can be expressed, for linear systems, by

$$H = \frac{y_{out}}{y_{in}} = \sum_{k=1}^{N} \frac{G_k \Delta_k}{\Delta} \tag{3.7}$$

where y_{in} and y_{out} represent the input and output node parameters, respectively, and H is the *transfer function* that represents the total gain of the system, G_k is the forward gain of the kth forward path and Δ_k is the loop gain of the kth loop. Δ, the determinant is formally defined, in the present context, as

$$\Delta = 1 - \sum L_i + \sum LiL_j - \sum L_i L_j L_k + \cdots + (-1)^n \sum \cdots + \cdots \tag{3.8}$$

[10]Mason's rule is also referred to as "Mason's gain formula".

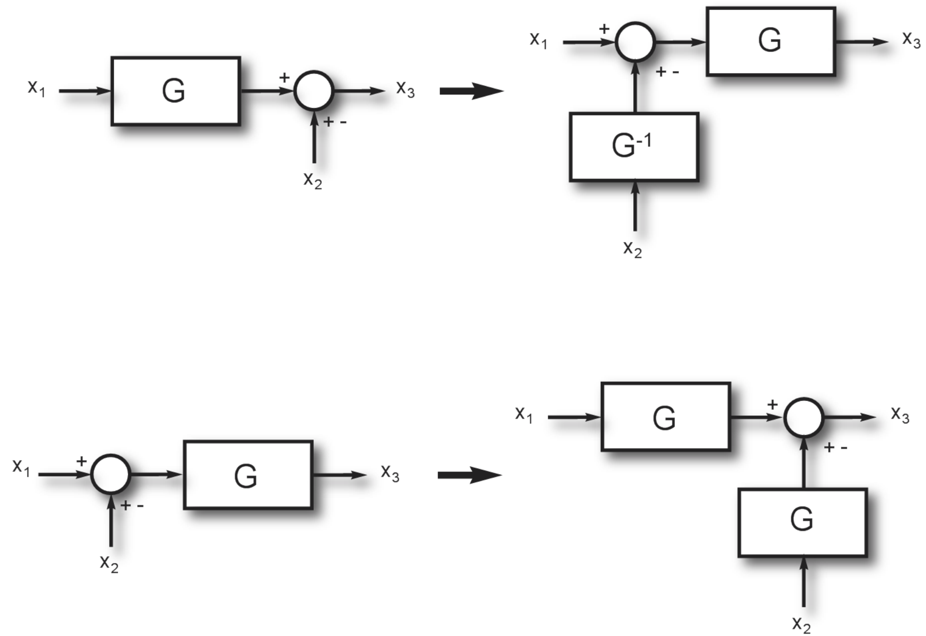


Figure 3.7: Expansion of single blocks involving summing junctions.

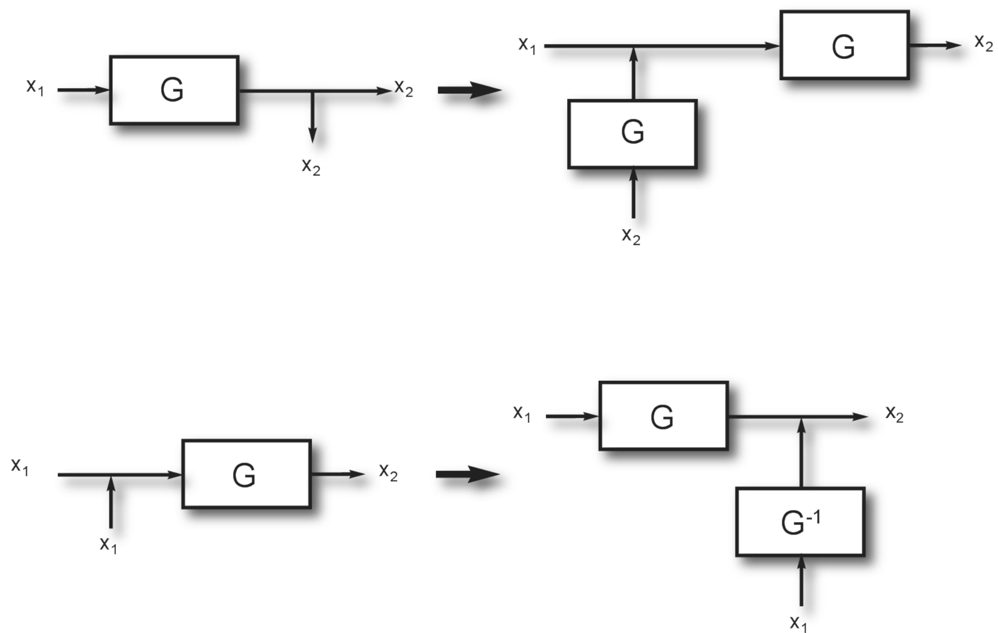


Figure 3.8: Expansion of single blocks to two equivalent blocks.

Equation (3.9) can be expressed in words as

$$\begin{aligned}\Delta &= 1 - \textit{(sum all the different loop gains)} \\ &+ \textit{(sum of the products of all pairs of loop gains for non-touching loops)} \\ &- \textit{(sum of products of all of the triples of loop gains, for non-touching loops)} \\ &+ \cdots \end{aligned} \tag{3.9}$$

The gain of the circuit shown in Figure 3.9 can be determined by applying Mason's rule as follows:

$$M_1 = A_2 \tag{3.10}$$

$$\Delta_1 = 1 \tag{3.11}$$

and therefore,

$$\Delta = 1 - L_1 = 1 - A_2 A_1 \tag{3.12}$$

so that

$$H = \frac{\sum M_j \Delta_j}{\Delta} = \frac{A_2}{1 - A_1 A_2} \tag{3.13}$$

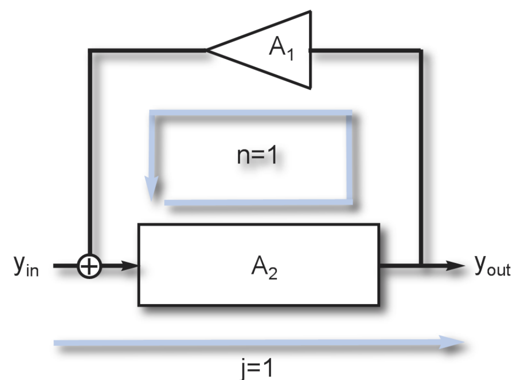


Figure 3.9: A simple application of Mason's rule.

3.3.2 Finite State Machines

Systems that consist solely of combinational logic, do not have any explicit time dependence and therefore the outputs are not time, or prior history, dependent. Simply stated, the outputs of such systems do not depend on any previous values of input, or output[11]. However, the outputs, at any point in time, of a finite state machine, are dependent on the states that the system passed through in order to reach the current state, the current input values and therefore the time required to produce the current outputs.[12]

[11]This assumes of course that the outputs of such systems are not subject to delays within the system and therefore are an immediate consequence of the inputs to the system.

[12]Chapter 6 treats FSMs in some detail.

Finite state machines (FSMs) are commonly used to implement decision making algorithms which are a key element of most embedded systems. They are particularly attractive for systems that are highly event-driven and are often employed as an alternative to a system based on a real time operating system. State machines are used in applications where distinguishable, discrete states exist. Finite state machines are based on the idea, that for a given system that has a finite number of states, there are two types of FSMs (Mealy and Moore) and they are distinguished by their output generation, viz., a Mealy machine has outputs that depend on the state and the input, and a Moore machine has outputs that depend on the state only. FSMs can also be represented by graphical representations in the form of state charts and hierarchically nested states, as illustrated by the example shown in Figure 3.10.[13] The subject of finite state machines

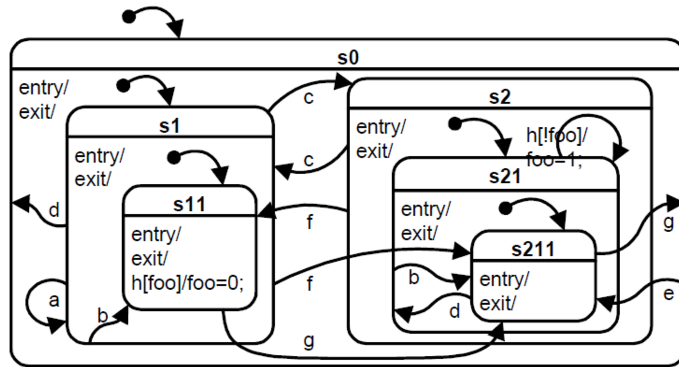


Figure 3.10: An example of a six state, nested state chart.

is treated in some detail in Section 5.12.

3.3.3 Coupling and Cohesion

There are a number of important characteristics of an embedded system, e.g., fault tolerance/ prevention, identification of exceptions, exception handling and module independence in terms of coupling and cohesion. The term *coupling* refers to the relative interdependence of modules and can be broadly characterized in terms of either *tight-* or *loose-coupling.* Examples of loose- and tight-coupling are shown Figures 3.11 and 3.12, respectively.

> The terms *tight-* and *loose-coupling* express the degree to which all the elements of a module are directed towards a single task/procedure and all elements directed towards that task/procedure are contained within a single component.

A design consisting of a two, or more, loosely-coupled modules, can provide some immediate benefits because the complexity of a system is often directly proportional to the degree of coupling between modules[14], i.e., the tighter the coupling the greater the interdependence between modules, and therefore the greater the complexity of the interaction between them. Thus, the interactions between modules should, as a general rule, be kept to the minimum level required

[13]State nesting allows new states to be defined in terms of previously defined states and to be defined in terms of differences from previous states, thus fostering reusability. This technique based on the concept of inheritance.

[14]An analogous situation can arise in applications which employ software modules that rely on global variables. Whenever ever possible passing data by value or reference is preferred.

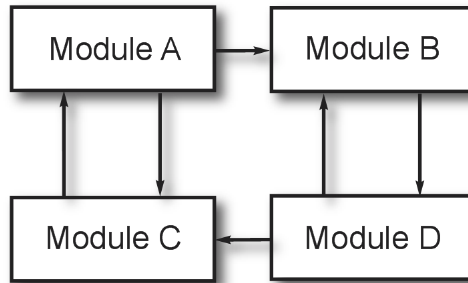


Figure 3.11: Weak coupling between modules A, B C and D.

to allow them to interactive effectively. By using loosely-coupled modules in a design, debugging can often be significantly reduced and design modifications/trouble-shooting, in the field, can be greatly facilitated.[15] There are, of course, systems in which some modules, by necessity, require extremely tight coupling in order to function effectively, e.g., in cases in which error-free communication is required and/or high-speed data transfer rates are involved.

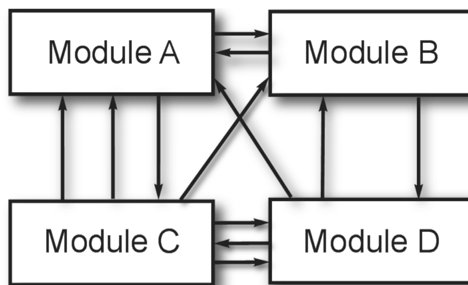


Figure 3.12: Tight coupling between modules A, B C and D.

Cohesion is a measure of the degree to which a set of tasks/procedures within a module are related. There are a variety of cohesion types, e.g.,

- *Coincidental cohesion* - Procedures/tasks just happen to be grouped within a module, but the interdependence between such procedures/tasks is weak.
- *Temporal cohesion* - Independent tasks are grouped within a module because they have some time dependencies, e.g., they must be completed within some pre-defined time period, and/or are sequentially ordered with respect to time.
- *Sequential cohesion* - A given task depends on procedures that must be ordered sequentially.
- *Functional cohesion* - The module's sole function is to carry out a specific task and the procedures within that module are restricted to those necessary to perform the task.
- *Communication cohesion* - All of the operations within a module are working on a common set of input data and/or produce the same output data.
- *Logical cohesion* - A set of tasks/procedures that are related logically, and not functionally. Typically, several logically-related tasks reside within a give module and are selected by an external user.

[15]Field trouble shooting is often based on the time honored practice of "isolating and destroying" techniques. Weakly coupled modules typically make isolating problems much easier.

- *Procedural cohesion* - Related tasks/procedures are contained within a module to ensure a particular order of execution. In such modules, it is control, and not data, that is passed from one procedure/task to another.

3.3.4 Signal Chains

The phrase *signal chain*[16] refers to a signal's path through a series of signal-processing components, of the type used in embedded systems, that acquire data signals and process them in a serial fashion. A *programmable signal chain* (PSC) is based on programmable analog devices deployed in conjunction with digital logic and a high performance CPU in the form of a microcontroller, microprocessor or DSP[17]. Such configurations are quite capable of providing embedded systems that are highly adaptive, versatile and effective for addressing a wide variety of mixed-signal applications. This is particularly important when designing systems that can benefit from such a system's ability to reconfigure itself, in real time, to meet variable operating environments and conditions.[18]

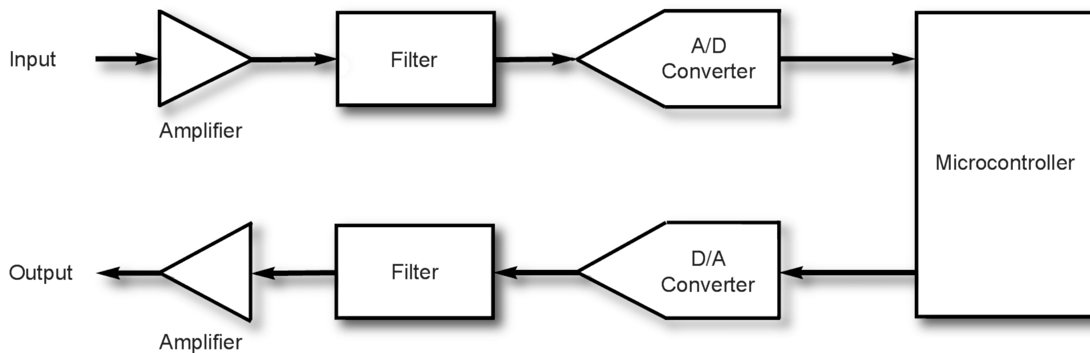


Figure 3.13: A commonly encountered signal chain.

One of the most commonly encountered signal chains is shown in Figure 3.13. The input to such a system is often from various types of sensors and the output may be to actuators, data channels, wireless transmission, or other devices. Input amplifiers, in the form of generic OpAmps, instrumentation amplifiers, lock-in amplifiers, radio frequency (RF) amplifiers, etc., are typically used to accept inputs from low level, high impedance sources and convert them into low impedance[19], high level signals. The OpAmps employed are often five-terminal devices, i.e., positive input, negative input, ground and two supply voltages, e.g., +/- 6 volts. However, there are a variety of special application amplifiers available that are designed for specific types of signal handling capable of

- demodulating low level signals,
- preparing analog signals for processing by A/D and D/A converters,

[16]The phrase *signal processing chain* is sometimes used instead of *signal chain*, but in either case refers to a series of signal-conditioning components involved in analog signal acquisition, processing and control, that are typically encountered in mixed-signal, embedded systems

[17]Digital signal processors are highly specialized microprocessors whose architecture is specifically designed to perform highly optimized signal processing functions.

[18]This ability is referred to as *dynamic reconfigurability*.

[19]Low impedance makes it possible to supply sufficient power to successive stages to avoid adversely effecting the signal,

- supplying low output impedance and high speed output and providing automatic gain control (AGC) or variable gain control (VGC),
- demodulating low level signals and compressing signals with high dynamic range[20],
- extracting a low level, differential signal from a larger common mode signal, while filtering out transient and/or other unwanted signals,

and,

- accurately reproducing input signals, by minimizing distortion of the input waveform.

Some sensors produce output signals that are in the form of *modulated carriers,* that is, the signal is transmitted as either a variable amplitude, or a variable frequency signal, or some permutation thereof, e.g., a signal embedded in a carrier is shown in Figure 3.14. This technique

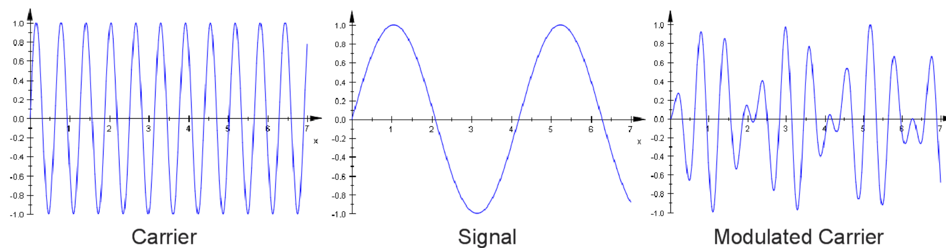


Figure 3.14: An example of amplitude modulation.

is often used to minimize the effects of amplifier noise, offset and grounding problems. One technique, employed in such cases, is to use a so-called *coupling transformer* to provide DC isolation between the sensor and the input amplifier, as shown in Figure 3.15. In such cases,

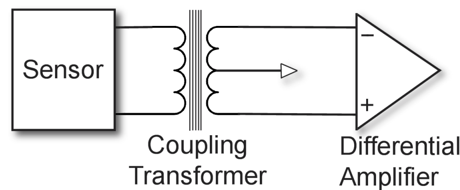


Figure 3.15: Transformer-coupled input sensor.

a center tap on the secondary, or output of the transformer, provides a common ground as a reference for the output from the secondary winding of the transformer. Note that this technique converts a single-ended input into a differential output.

Demodulation of this signal can be accomplished by use of a variety of techniques, e.g., for extremely low level signals, lock-in amplifiers capable of detecting a signal as low as 100 dB[21] below the ambient noise level can be employed. Signals may be measured as either single-ended or differential inputs. In the former case, the input signal is measured with respect to signal ground[22]

[20]Dynamic range is defined as the ratio of possible high to low signal values, either current or voltage, supported by a given device.

[21]dB is a logarithmic unit of measurement which is based on the ratio of a physical quantity with respect to a reference level. In the case of power, it is defined as $10\,log_{10}\,(\frac{P_1}{P_0})$ and for voltage as $20\,log_{10}(\frac{V_1}{V_0})$. Thus 100 dB below ambient is equivalent to -100 dB or 0.0000000001 (10^{10}) of the associated reference value.

[22]Note that signal ground may, or may not, be the that of the power supply or a common ground. The ground used can be a source of unwanted signals, i.e., noise, and careful attention must be paid in such cases to employing adequate grounding techniques.

and is sometimes coupled capacitively to the amplifier's input. Differential signals require that both the positive and negative inputs of an amplifier be used[23], and the measurement of both inputs is with respect to a common ground.

While at the block level, signal chains can be relatively simple, even the simplest of signal chains involving perhaps the measurement of an external resistance that is related to some physical parameter of interest such as temperature, pressure, flow rate, etc. can present substantive issues that must be taken into consideration. For example, non-linearities in resistance versus the value of the physical parameter being measured, temperature versus resistance variations in the sensor and the connections to the sensor[24], accuracy and precision requirements, ambient interference environment and associated adverse affects on the sensor and connections to the embedded system, variations in resistance of the interconnections between the sensor, and the embedded system, gain variations of the input stage, sensor excitation requirements, crosstalk interactions, and required filtering must also be taken into account.

3.4 Schematic view of the system

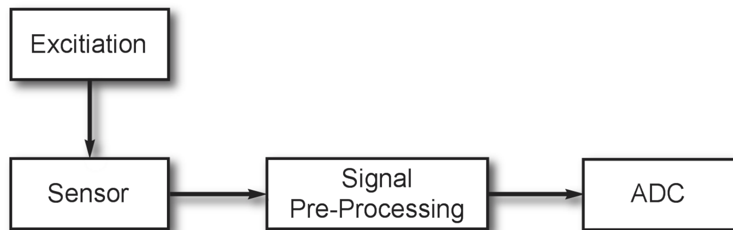


Figure 3.16: A block diagram of a temperature measuring signal chain.

As an illustrative example, a simple signal chain, related to making a temperature measurement, as shown in Figure 3.16, can be represented in the form of a block diagram. A schematic representation of this block diagram is shown in Figure 3.17. It should be noted that in practical applications, the ground connections for the sensor resistor, and ADC, are all made in close proximity when implemented in a physical system. This type of differential measurement connection reduces noise problems by virtue of the fact that common mode measurements tend to cancel out signals that are picked up by the differential input lines. An even simpler signal chain for this type of measurement is shown in Figure 3.18 in which the digital-to-analog converter has been replaced by a digital-to-current converter, thus eliminating the requirement for the reference resistor.

Temperature measurements using resistive-devices, such as thermistors[25], whose resistance is a function of the ambient temperature, tend to exhibit non-linear characteristics. As discussed, in Chapter 1, the resistance of a thermistor, as a function of temperature, can be approximated

[23]An OpAmp with both positive and negative inputs can serve as either a single-ended or differential amplifier. Single-ended applications are accomplished by simply grounding one of the amplifiers inputs and applying the input signal to the other input.

[24]Including such considerations as temperature gradients along the wires connected to the sensor and embedded system input.

[25]Thermistor is an acronym for thermal resistor which refers to a resistor who resistance is a known function of temperature. While usually a non-linear relationship exists between temperature and resistance for such devices, they can sometimes be treated as being quasi-linear, over the range of interest.

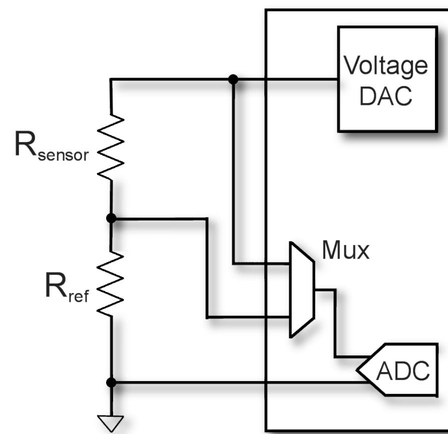


Figure 3.17: A schematic diagram of the signal chain shown in Figure 3.16.

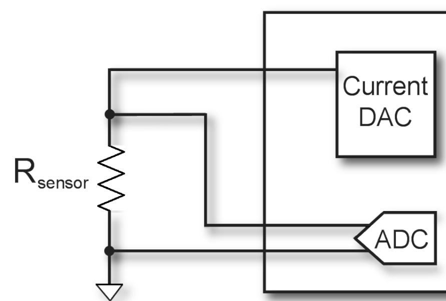


Figure 3.18: A further simplification resulting from employing a current DAC.

by the equation

$$\frac{1}{T} = A + B\,ln(R) + C[ln(R)]^3 \tag{3.14}$$

where T is the thermistor's ambient temperature, R is the measured resistance and A, B and C are constants characterizing the particular thermistor involved. Rather than solving this equation explicitly for each temperature measurement, it is simpler, and more efficient, to employ a lookup table that has the discrete values for resistance and temperature for that particular type of thermistor. If necessary, linear interpolation[26] using the coordinates of the known points on the curve can be employed, for additional quantitative detail.

Resistance temperature detectors[27] (RTDs) are a particularly useful type of temperature sensor that have the property that at 0°C the resistance is nominally 100Ω, and the rate of change of resistance with respect to temperature (dR/dT) is 3.85Ω per degree Centigrade[28]. RTDs have very low resistance and therefore the effect of any additional low-wire-resistance paths must be taken into consideration. Typically, the voltage across an RTD is measured by using either 3- or 4-wire methods. The 3-wire method is shown in Figure 3.19. The voltage measured by the ADC

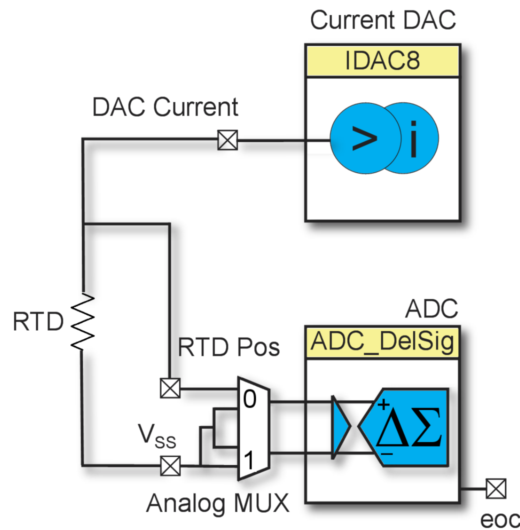


Figure 3.19: Schematic view of the 3-wire circuit in PSoC Creator.

is a combination of the drop across the RTD and the wire connecting the DAC to the resistor, and the voltage across the resistor. The current through the wire and resistor are known and the resistance of the wire connection can be measured so that the voltage drop across the wire is known.

[26] Linear interpolation is based on the idea that if two points are known on a given curve, e.g. a graph of temperature versus resistance (T vs. R), then the slope of a line drawn between the two points can be easily determined and the value of the temperature for a given resistance between those two points can be approximated by evaluating the following expression: $T = T_0 + [(T_1 - T_0)/(R_1 - R_0)]$ where (R_0, T_0) and (R_1, T_1) are known values and are chosen sufficiently close to provide the required accuracy. Linear interpolation is believed to have been used by Babylonians as early as 2000-1700 BC. [45]

[27] C.H. Meyers [47] first proposed the RTD in the form of a helically wound platinum coil on a crossed mica web inside of a glass tube. However, its thermal response time was too slow for many applications and it was subsequently supplanted by a design by Evans and Burns [20] that instead used an unsupported platinum coil which allowed it to move freely as a result of thermal expansion and contraction.

[28] Platinum RTDs are extremely accurate and stable compared to thermistors and other commonly encountered temperature sensors.

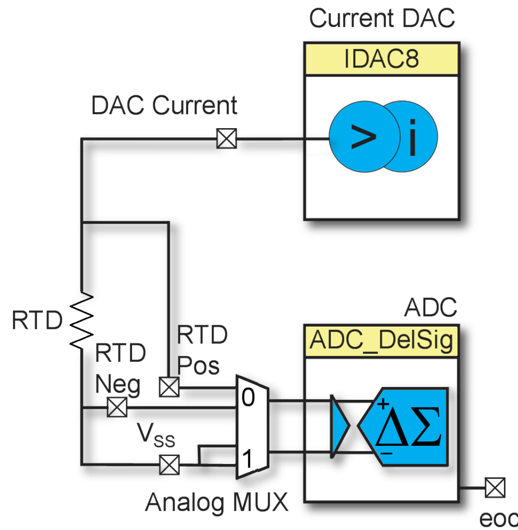


Figure 3.20: Schematic view of the 4-wire circuit in PSoC Creator.

3.5 Correlated Double Sampling (CDS)

Signal chains often involve small amplitude, and relatively low frequency, signals. The accuracy, as well as precision, of measurements of such signals can be limited by various non-deal characteristics such as offset and noise. Offset potentials[29], offset potential drift[30] and low frequency noise[31], all of which are functions of temperature, arise frequently in systems such as those shown in Figure 3.19 and 3.20. Obviously these are highly undesirable effects to have present when making sensitive measurements. Some OpAmps employ chopper stabilizing to minimize drift by periodically grounding the input(s) of the amplifier.[32]

A technique known as *correlated double sampling* (CDS) can be used to minimize these effects.[60] CDS functions as a high-pass filter which allows the (1/f) noise to be reduced and is a signal processing method that reduces unwanted effects that often occur when employing sensitive sensors. This technique is most effective in addressing slow-changing, in terms of frequency and amplitude, signals of the type encountered when using Hall-effect,[33] capacitive or thermocouple sensors.[34]

[29]Ideal OpAmps have zero output, when the input voltage differential is zero, as opposed to nonideal OpAmps that exhibit some output voltage under4 such conditions. This can be "offset" in some cases by applying a small potential to one of the inputs sufficient to assure that the output is zero volts, under such input conditions. PSoC3/5 offset voltage is spec'ed at a maximum of 2 mv.

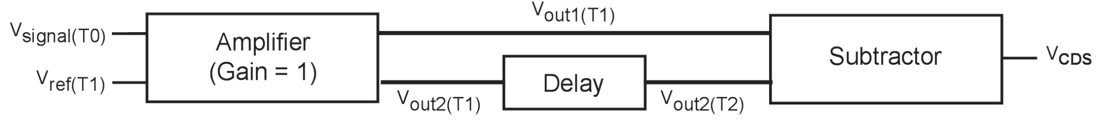
[30]Typical offset voltage drift ($TCVos$) for a PSoC3/5 OpAmp is $\approx 6\mu v/°$C ($12\mu v/°C$ maximum). PSoC3/5's delta-sigma analog-to- digital converter (ADC_DelSig) includes a feature known as Vref_Vssa. When an external reference is being supplied, the Vref_Vssa connection can be routed thorough the analog routing fabric to an external pin on the device. A connection to this pin of an external reference eliminates any offset in the reference as a result of internal IR drops in the Vssa pin and bonding wire.

[31]The noise considered in this example if classified as "1/f noise" which is found in any semiconductor device.

[32]Early operational amplifiers used in applications such as integrators employed mechanical switches that were electrically driven. Current devices employ semiconductor switches for this purpose.

[33]The output of a Hall-effect sensor is a function of the ambient magnetic field. A thin piece of conductive material is used which has two connections that are placed perpendicular to the direction of current flow through the device. An external magnetic field will cause a potential to arise between the two connections that is directly proportional to the ambient magnetic field.

[34]The basic techniques described in this section are applicable to PSoC1, PSoC3 and PSoC5. [64][73]



$$V_{out1(T1)} = V_{signal(T1)} + V_{offset(T1)} + V_{noise(T1)}$$

$$V_{out2(T1)} = V_{ref(T1)} + V_{offset(T1)} + V_{noise(T1)}$$

$$V_{out2(T2)} = V_{ref(T2)} + V_{offset(T2)} + V_{noise(T2)}$$

Figure 3.21: CDS OpAmp block diagram.

As shown in Figure 3.21,

$$v_{out1(T1)} = v_{signal(T1)} + v_{offset(T1)} + v_{noise(T1)} \tag{3.15}$$

and,

$$v_{out2(T2)} = v_{signal(T2)} + v_{offset(T2)} + v_{noise(T2)}. \tag{3.16}$$

assuming that

1. v_{ref} and v_{signal} are constant for values of t such that $t_1 <$ t $< t_{2.}$
2. v_offset has a constant value, i.e., it is not an explicit function of time.

and,

3. system noise is solely a function time.

Therefore subtracting (3.15) from (3.16) yields

$$\begin{aligned} v_{CDS} &= (v_{out(T1)} - v_{out(T2)}) && (3.17) \\ &= (v_{signal} - v_{ref}) + (v_{noise(T1)} - v_{noise(T2)}) + (v_{offset(T1)} - v_{offset(T2)}) \\ &= (v_{noise(T1)} - v_{noise(T2)}) && (3.18) \end{aligned}$$

This method is based on making two measurements, one from a sensor with an unknown input and one with an known input. Because the amplifier is assumed to be capable of producing only one output at a time, delaying the output of one signal with respect to the other, allows Eq (3.17) to be evaluated.[35] Then by subtracting the result of the known input from the unknown input, it is possible to compensate for the offset. This technique is based on first measuring the offset potential across the sensor with both inputs shorted and then measuring

$$v_t = v_{tc} + v_n + v_{ov}, \tag{3.19}$$

where v_t is the zero referenced voltage, v_{tc} is the actual thermocouple voltage, v_n ia the noise voltage and v_{ov} is the offset voltage.

Thus, for the previous zero-referenced sample

$$v_{zref} = v_n + v_{ov} \tag{3.20}$$

[35]Some applications use a sample and hold circuit followed by a subtractor, instead of employing a delay. PSoC Creator's Sample and Hold component is discussed in detail in Section 6.5.

and,

$$v_{zref_prev} = (v_n + v_{ov})Z^{-1} \tag{3.21}$$

in the continuous-time domain. It can be transformed into the discrete-time domain by employing Tustin's method[36], i.e.,

$$v_{signal} = (v_{tc} + v_n + v_{offset}) - (v_n + v_{offset})Z^{-1} \tag{3.22}$$

$$v_{signal} = v_{tc} + (v_n + v_{offset})(1 - Z^{-1}) \tag{3.23}$$

where Z is the bilinear transform, i.e.,

$$Z = \frac{(1 + \frac{sT}{2})}{(1 - sT)} \tag{3.24}$$

and,

$$T = \frac{1}{f_{sample}} \tag{3.25}$$

and therefore using Eqs. (3.24) and (3.25),

$$\frac{1}{Z} = \frac{1 - \frac{sT}{2}}{1 + \frac{sT}{2}} = \frac{\left[1 - \frac{s}{2f_s}\right]}{\left[1 + \frac{s}{2f_s}\right]} \tag{3.26}$$

so that

$$1 - \frac{1}{Z} = 1 - \frac{\left[1 - \frac{s}{2f_{sample}}\right]}{\left[1 + \frac{s}{2f_{sample}}\right]} = \frac{\left[1 + \frac{s}{2f_{sample}}\right] - \left[1 - \frac{s}{2f_{sample}}\right]}{\left[1 + \frac{s}{2f_{sample}}\right]} = \frac{2s}{(s + 2f_{sample})} \tag{3.27}$$

and therefore [64]

$$v_{signal} = v_{tc} + v_n\left[\frac{2s}{(s + 2f_{sample})}\right] = v_{tc} + v_n\left[\frac{2}{1 + (\frac{2f_{sample}}{s})}\right] \tag{3.28}$$

assuming that the offset is not a function of time. Equation (3.28) is clearly a high pass response, as shown in Figure 3.22. However, because this configuration does not reduce higher frequency noise, additional filtering may be required, e.g., an infinite impulse response (IIR) filter[37] can be used in such cases to reduce unwanted high frequency components.

A similar technique can be employed in using PSoC3/5's delta-sigma ADC in a CDS configuration as shown in Figure 3.23. In this case, the input signals, V_{signal} and V_{ref} are alternately

[36] This method is, in actuality, a conformal mapping which in the present case represents a mapping of a linear, time invariant function in the time domain, to a linear shift-invariant transfer function in the discrete-time domain. Conformal mappings preserve certain key aspects of the functions being mapped, e.g., in the present case preservation of characteristics in the frequency domain. The Tustin method is often used to provide good matching in the frequency domain between the discrete and continuous time domains, and in cases where a system's dynamics near the Nyquist frequency are of interest.

[37] These filters can be implemented as $y[n] = \sum_{k=0}^{M} x[n-k] + \sum_{k=1}^{N} y[n-k]$ where the b_k are the feedforward coefficients and the a_k are the feedback coefficients.

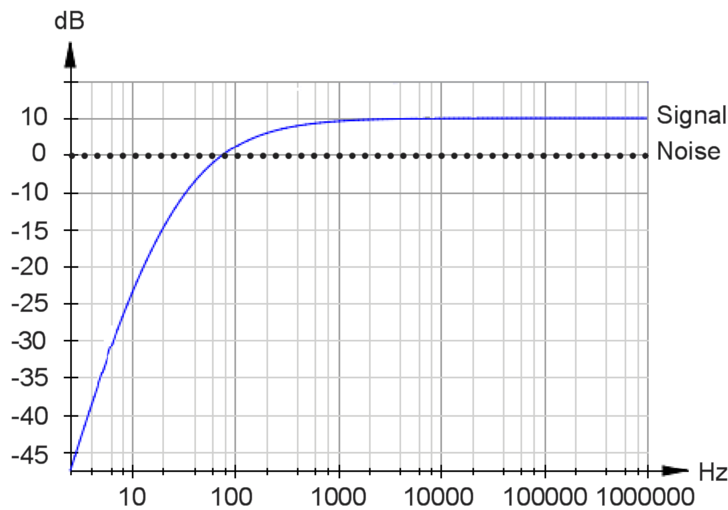


Figure 3.22: CDS frequency response.

passed to a buffer stage[38], which can introduce unwanted offset and noise[39], before being supplied to the ADC. The resulting digital forms of these two signals are then subtracted in firmware. Note that this delta-sigma ADC can be configured in either a single, or differential, input mode[40], as shown in Figures 3.24(a) and 3.24(b). Single and differential input modes can also be imple-

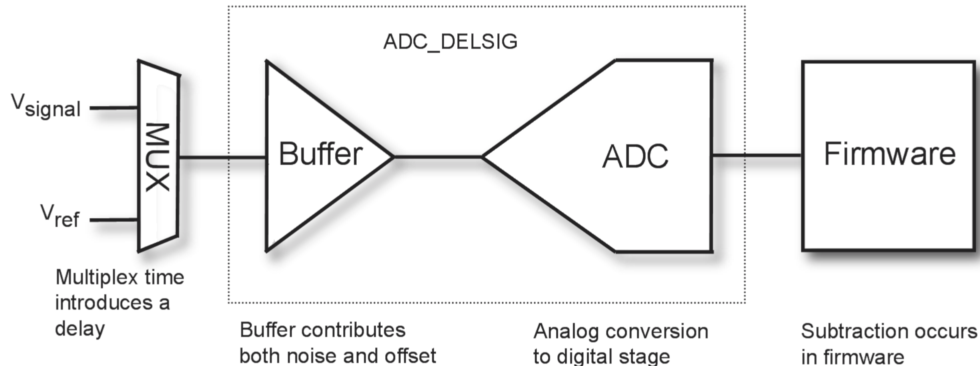


Figure 3.23: CDS implementation for PSoC3/5 *ADC_ DELSIG*.

mented using an analog multiplexer as shown in Figure 3.25.

The following is an example of the PSoC Creator source code for this application:

```
/*Get the first sample Vout1*/
AMux_1_Select(0);
```

[38]The sampling time between the two signals acts as the delay employed in the previous OpAmp example.

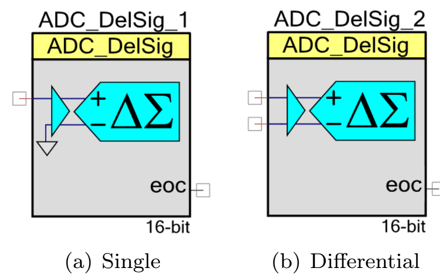
[39]Offset and noise may also be introduced by other devices in the signal path. In order to minimize such effects it is important that the reference signal and input signal both follow the same *signal path* to the extent feasible/possible.

[40]While in principle connecting an input to signal ground is equivalent to a zero input, in reality signal ground can introduce noise, so that in practical applications the differential mode is often preferable.

```
ADC_DelSig_1_StartConvert();
ADC_DelSig_1_IsEndConversion(ADC_DelSig_1_WAIT_FOR_RESULT);
iVout1 = ADC_DelSig_1_GetResult32();
ADC_DelSig_1_StopConvert();

/*Get the second sample Vout2*/
AMux_1_Select(1);
ADC_DelSig_1_StartConvert();
ADC_DelSig_1_IsEndConversion(ADC_DelSig_1_WAIT_FOR_RESULT);
iVout2 = ADC_DelSig_1_GetResult32();
ADC_DelSig_1_StopConvert();

/*perform CDS*/
iVcds = iVout1 - iVout2;
```



(a) Single (b) Differential

Figure 3.24: Single, versus differential, input mode for ADC_DelSig.

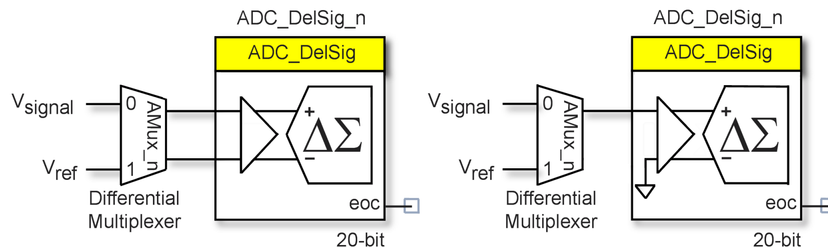


Figure 3.25: Single/Differential input using an analog multiplexer.

In the system shown in Figure 3.20, it is clear that its accuracy is solely a function of the IDAC's accuracy. Undesirable variations, i.e., deviations, in the output of the IDAC and ADC gain errors, can result from temperature dependencies. IDAC and ADC errors of the type found in this particular type of application can be reduced by introducing an additional, more accurate, resistance[41] as shown in Figure 3.26.

When making such measurements it is important to:

- Select the most appropriate sensor for the application.
- Employ a technique such as CDS to avoid offset errors[42]

[41] Commercial resistors are available whose variances are less then 0.1% , as a function of temperature.
[42] A filter can be used to remove noise when employing a thermocouple.

- Use current excitation to avoid inaccurate reference resistance.[43]
- Use a Delta-Sigma ADC with high accuracy and resolution to assure the highest possible overall accuracy.[44]

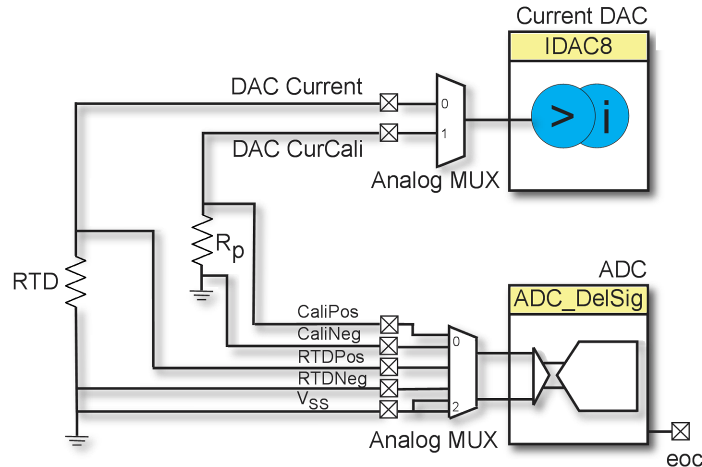


Figure 3.26: 4-wire RTD with gain error compensation.

3.6 Using components with configurable properties

The ADC and current DAC of the previous section are some of the basic components supported by PSoC Creator, and as such, they are highly configurable, as are most of the PSoC Creator components. The current source, *IDAC*, may be controlled by hardware, software or some combination of the two and can function as either a source, or a sink.

Similarly, the delta sigma, analog-to-digital converter, *ADC_DelSig*, is also highly configurable. PSoC Creator provides tabulated dialog boxes, an example of which is shown in Figures 3.28(a) and 3.28(b), to allow user-defined configuration for a given application. The conversion modes (0 - single sample, 1 - multiple samples, 2 - continuous samples or 3 - multiple samples (Turbo)), resolution (8-20 bits inclusive), conversion rate (2,000-48,000 samples per second), clock frequency[45]for each of the components in its integrated *Component Catalog* that allow the designer to specify key requirements for a design, e.g., parameters such as *power* (low, medium or high), *conversion mode* (fast filter, continuous, fast FIR), resolution $(8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)$, *conversion rate*, *clock frequency*, *input buffer gain* $(1, 2, 4, 8$ and disabled), *reference* (various forms of internal V_{ref} $[1.024 volts]$ and external reference connections), *clock frequency*, external/internal *clock source*, etc., give the designer the ability to adapt the PSoC3/5 embedded system to fit the relevant specifications and requirements of each application.

[43]If voltage excitation is employed, 4-wire measurement techniques should be used.

[44]PSoC3/5 are excellent platforms for such measurements in that functions such as a very high resolution Delta Sigma ADC, current source, voltage source, multi-pole filter and high resolution/speed digital processing are all tightly integrated within a single chip.

[45]The clock frequency is a function of the resolution and changes programmatically as a function of the conversion rate selected.

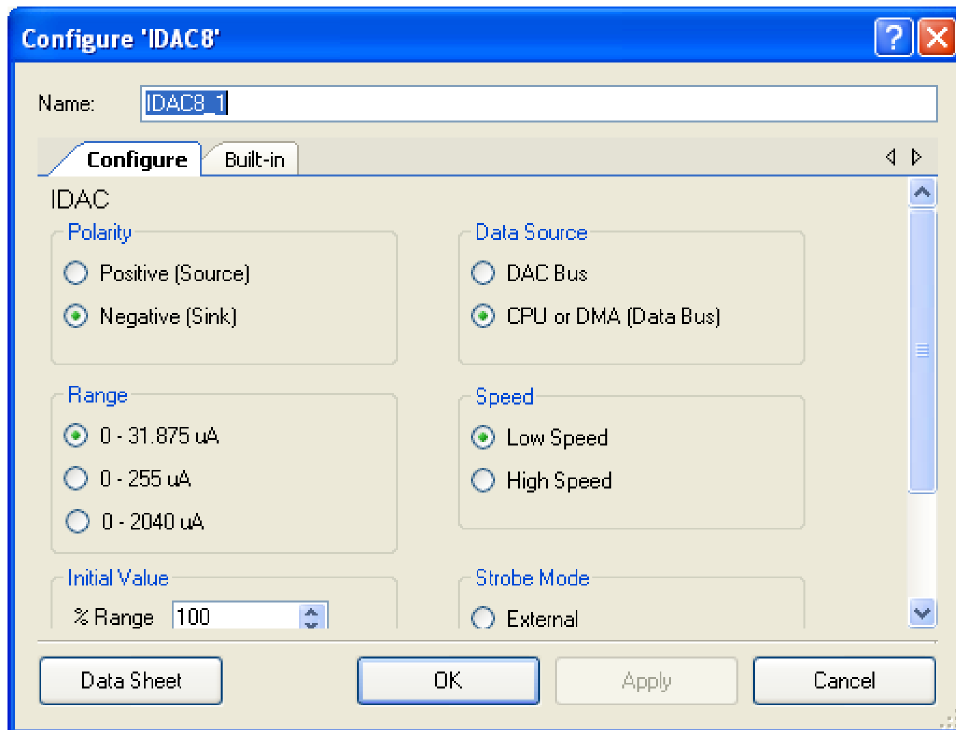


Figure 3.27: The *Configure IDAC8* dialog box in PSoC Creator.

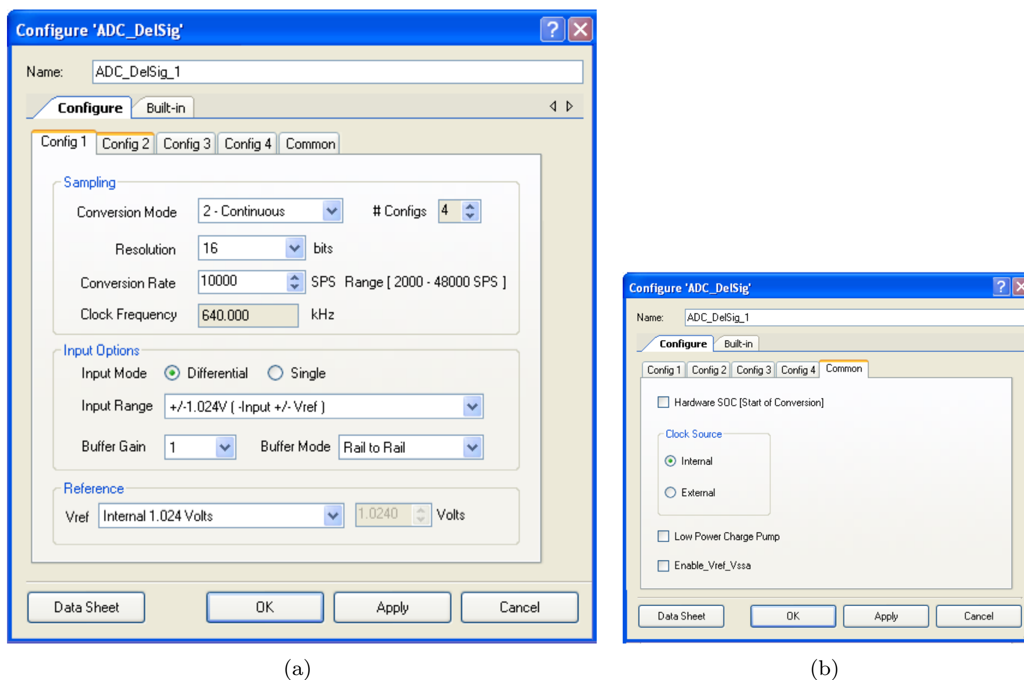


Figure 3.28: PSoC Creator *Configure ADC_Del_Sig_n* dialog boxes.

Table 3.1: Resolution vs. Conversion Rate and Clock Frequency.

Resolution	Conversion Rate (sps) (Single Sample)	Clock Frequency
8	1911- 91,701	.128037 - .6143967 MHz
9	1,543 - 74,024	.128069 -.6143992 MHz
10	1,348 - 64,673	.128060 - .6143935 MHz
11	1,154 - 55,351	.128094 - .6143961 MHz
12	978 - 46,900	.128118 - .6143900 MHz
13	806 - 38,641	.128154 - .6143919 MHz
14	685 - 32855	.128095 - .6143885 MHz
15	585 - 28054	.128115 - .6143826 MHz
16	495 - 11861	.128205 - .3071999 MHz
17	124 - 2965	.128464 - .3071740 MHz
18	31 - 741	.128464 - .3070704 MHz
19	4 - 93	.131840 - .3065280 MHz
20	2 - 46	.263680 - .3032320 MHz

3.7 Types of Resets

PSoC3/5 support power-on resets (POR)[46], hibernate resets ($\underline{HRES}$), watchdog resets ($WRES$)[47], software resets ($SRES$) and external resets ($XRES_N$)[48] via the reset module as shown in Figure 3.29. When a reset occurs, regardless of the type, all registers are restored[49] to their default states except for so-called *persistent* registers.[50] Figure 3.30 demonstrates various reset responses as a function of time with respect to the change in V_{dd}/V_{cc} as well as the time dependencies for reset during normal power-up (POR). In some designs, a low startup time is essential. In these designs, there are a number of steps that can be taken to reduce PSoC3's startup time. Gains depend heavily upon the configuration of the target device, but switching from CPU to DMA population may save on the order of 1-20 ms. Running the partially trimmed IMO at 48 MHz instead of 12 MHz will speed up most portions of startup by a factor of 4, but a fully trimmed IMO at a higher frequency will also improve startup time. Because most startup occurs under partially trimmed IMO, the benefits will not be as significant as changing the partially trimmed IMO frequency. As with increasing the partially trimmed IMO frequency, this change will increase current consumption. Much of startup is CPU or DMA limited, and these two resources will operate at the speed of the IMO. The downside to increasing the speed of the partially trimmed IMO is that device current consumption will increase. While the power supply ramp is not normally considered part of microcontroller startup, it does block the beginning of the startup procedure and performance can be improved in some cases by increasing the speed of the VDD ramps, to further minimize startup time. startup.

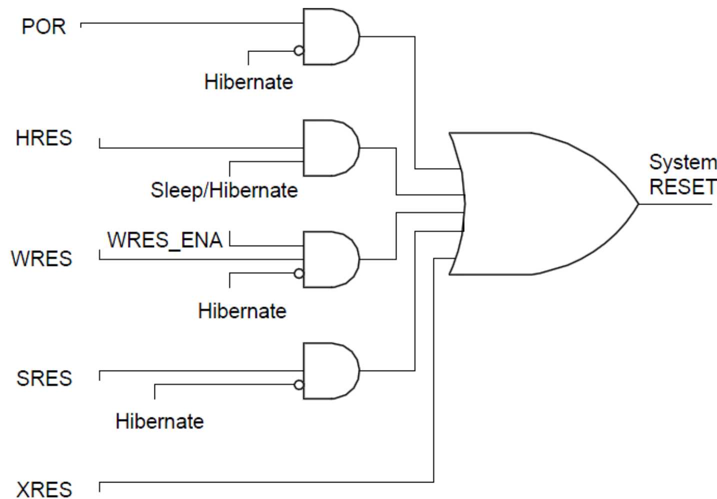


Figure 3.29: Reset module logic diagram

[46]When PSoC3/5 is powered up, it is held in reset until all of the VDDx and VCCx supplies reach the appropriate levels for correct operation.

[47]Watchdog reset is used to recover from errors that would otherwise keep the system from functioning properly and that may be recoverable if the system is "reset", that is "re-booted". The Watchdog Timer (WDT) circuit automatically reboots the system if the WDT is not continuously reset within a user-defined period of time. The

[48]If a reset pin is not required then this pin can be reprogrammed to be a GPIO.

[49]DMA is much faster than CPU intervention at populating device registers.

[50]Both the $RESET_SR0$ and $RESET_SR1$ registers contain "persistent status bits" which can only be reset under particular circumstances, e.g., in the case of a POR. Specific bits in these registers are set for each type of reset and remain set until the *tsrst_ en* bit is cleared and either a POR or a user/application induced reset occurs. However, these bits are only accessible if the *tstrst_ en* bit (bit 4) in the test controller's *TC_ TST_ CR2* register is set.

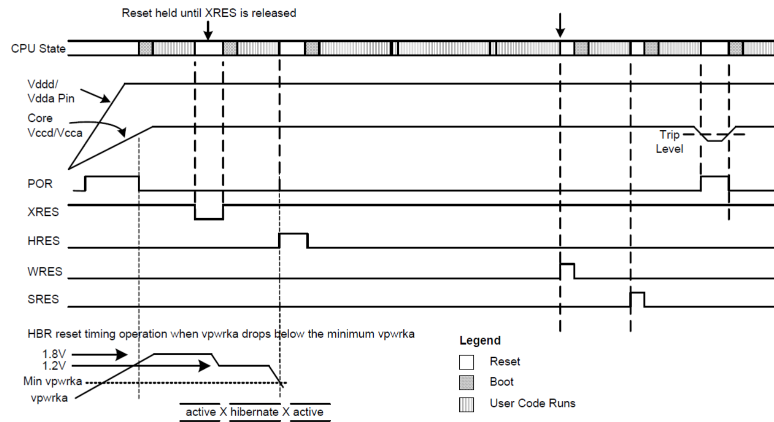


Figure 3.30: Resets resulting from various reset sources

3.8 PSoC3 Startup Procedure

Application software designed to realize an embedded system relies on operating in a known hardware environment and under the constraints imposed by a specific set of initialization parameters and conditions, when power is applied to the microcontroller. Thus, the microcontroller must be provided with code designed to cause it to enter a known state with the appropriate initialization. This is accomplished by a combination of two firmware components known, respectively, as the bootloader and a bootloadable project

Following *powerup*, or alternatively, a *reset* caused by the the XRES pin, watchdog timer, low voltage detection circuit, power-on-reset or other source, the PSoC3/5 hardware is configured by initiating the appropriate hardware startup procedures.[51] *Power-on-reset* (*POR*) occurs during the ramp-up of the supply voltage and is not released until all associated power supplies have reached their appropriate operating values. Once the POR has been released, the device enters the boot phase in which a hardware state-machine controls the basic configuration and trim of the target device, using direct memory access (DMA).

Startup begins after the reset of a reset source, or following the end of a power supply ramp.[52] There are two primary startup segments: hardware and firmware as shown in Figure 3.31.

\Once the hardware startup phase has been completed the system begins the firmware startup. The firmware loads the configuration registers, subject to the requirements set by the application and PSoC Creator, e.g., configuring the analog and digital peripherals, clocks , routing, etc. In addition, the debugging, bootloader and DMA resources are also configured.[53] Upon completion of the firmware startup, the CPU begins executing the user-authored code beginning at memory address location zero.

Register RESET_SR0 (0x46FA)[54] contains information about the status of the software reset, watchdog reset, analog HVI detector, analog LVI detector and digital LVI detector and and

[51] The device's I/O pins are placed in the high-Z drive mode while the reset is asserted and until pin behavior has been loaded.

[52] Because the power supply ramp blocks the beginning of startup, V_{dd} is referred to as *Svdd* in Cypress Semiconductor Corporation datasheets, should be taken into consideration as part of the design process.

[53] Not all of the PSoC Creator components will be fully configured following the firmware startup phase. In some cases, additional code will be required to fully activate them.

[54] Reset and voltage status register 0 (RESET_SR0)

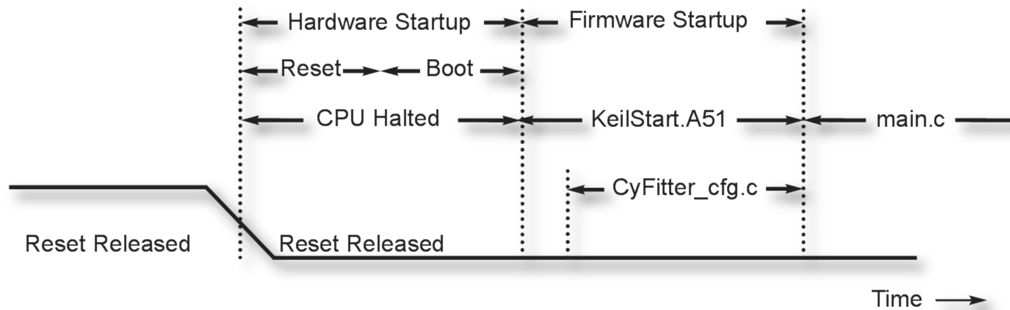


Figure 3.31: Overview of PSoC3 startup procedure.[31]

RESET_SR1 (0x46F8)[55] contains information about the status of the analog PRES, digital PRES[56], analog LPCOMP[57] and digital LPCOMP.

RESET_CR2 (0x46F6) controls the software initiated reset (SRES). Setting bit 0 of this register (*swr*) to 1 will cause a system reset that can be initiated by software, firmware, or DMA which will result in the setting of RESET_SR[5]. It will remain set until reset by the user, or until a POR/HRES[58] reset occurs.

KeilStart.A51[59] contains 8051 assembly code that is executed at the beginning of the firmware startup to configure some of the basic components in PSoC3, e.g., debugging, bootloaders, DMA endpoints and, if required, the clearing of SRAM. KeilStart. A51 code begins at memory address 0 in Flash which contains an unconditional jump to STARTUP. KeilStart also calls *CyFitter_ cfg()* which can be used by the designer to handle certain clock startup errors, e.g., bad MHz crystal, loss of PLL lock, etc., and to configure some analog device default settings. The "clear IDATA" step, shown in Figure 3.32, writes zeros to program memory allocated for IDATA.[60] The "DMAC configuration" step configures the DMA resources subject to the specification of PSoC Creator for the particular application.

The function *CyFitter_ cfg*() is invoked by $CyFitter_cfg.c$ and results in the population of a significant number of registers, as illustrated in Figure 3.33, the largest group of which are those associated with analog and digital resources. This step may be carried out under either CPU control, or via DMA[61]. A somewhat smaller group of registers are configured as a result of the *ClockSetup()* API call which results in configuration of of PSoC3's clock tree and clock resources. The specific configuration of the project's clocks is determined by PSoC Creator.[62]

After the target device has been reset, it is clocked by the fast output of the internal main oscillator (IMO) which is based on a fast reference. Once the normal reference becomes stable, the normal IMO becomes valid. The IMO begins to source the normal reference during the reset

[55]Reset and voltage detection status register 1 (RESET_SR1)

[56]Precision POR (PRES) refers to a reset that occurs based on a precision trip point. An imprecise POR (IPOR) refers to a reset that occurs during power up that keeps the target device in reset until Vdda, Vcca, Vddd and Vccd are at the values specified in the deices data sheet.

[57]LPCOMP refers to PSoC3/5's low power compare circuit.

[58]POR/HIB refers to power on reset and/or hibernate reset.

[59]KeilStart.A51 is proprietary, 8051- based source code owned by Cypress Semiconductor for incorporation in PSoC3 applications developed with Keil development tools.

[60]This memory allocation is usually for variables.

[61]The function $cfg_write_bytes_code()$ loads this group of registers by utilizing the CPU. The function $cfg_dma_dma_init()$ loads the same group of registers via DMA.

[62]The clocks tab in PSoC Creator can be accessed by double-clicking the *.cydwr* file for a project.

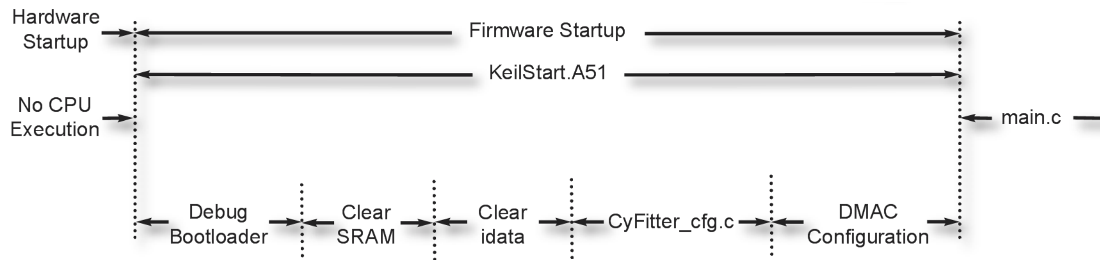


Figure 3.32: KeilStart.A51 execution steps.[31]

phase. The IMO is then running nominally at either 12 or 48 MHz, as configured by the device's non-volatile latches (NVLs)[63]. T_{IO_init} is the delay, as specified in the target device's data sheet, that determines the delay after which the pins, and other resources, begin to behave as required by the application.

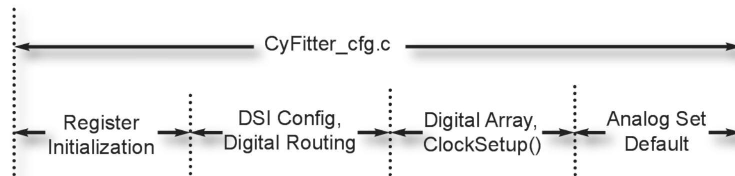


Figure 3.33: $CyFitter_cfg.c$ execution steps.[31]

The population of registers is determined by the Mode selection options in the *.cydwr* tab, in PSoC Creator, as shown in Figure 3.34. The compressed mode option causes the CPU to populate the configuration registers and store data in Flash, optimizing Flash usage, rather than startup time.The DMA mode populates the registers under DMA control blocking the CPU execution until the DMA configuration of the registers has been completed. As expected, DMA population is significantly faster than CPU population. "Clear SRAM" determines whether or not SRAM is to be cleared after a reset[64] for an IMO speed of 12 MHz. "Enable Fast IMO" selects the IMO speed as either 2 or 48 MHz, partially trimmed, i.e., slow boot or fast boot mode, respectively. It should be noted that startup code is regenerated each time a change is made to a PSoC Creator schematic, or design, resources. Thus, if the designer has made any changes to the $KeilStart.A51$, and/or $CyFitter_cfg.c$, files can be lost. In order to avoid any such loss, these files must only be edited when there is no need to perform a "generate" operation to ensure that the configuration in the automatically generated source files matches the application's design resources and schematic. The design wide resources (DWR) and schematic changes are followed by a "clean and build"[65] and then editing of the source files. The project can then be subjected to a "build" and the resulting firmware will reflect the respective edits. However, any subsequent "clean and build" actions will result in modifications to the generated source code.

[63]A Nonvolatile Latch (NVL or NV latch) is an array of programmable, nonvolatile memory elements whose outputs are stable at low voltage. It is used to configure the device at Power-on-Reset. Each bit in the array consists of a volatile latch paired with a nonvolatile cell. On POR release nonvolatile cell outputs are loaded to volatile latches and the volatile latch drives the output of the NVL.

[64]Clearing SRAM requires approximately 4500 CPU clock cycles, at 12 MHz, to clear 8kB of SRAM. However, if SRAM is not cleared but variables are initialized properly, not clearing SRAM will have no adverse effect on firmware operation.

[65]PSoC Creator's *Clean and Build Project* command causes the intermediate and output files of any previous build to be deleted prior to initiating a new build.

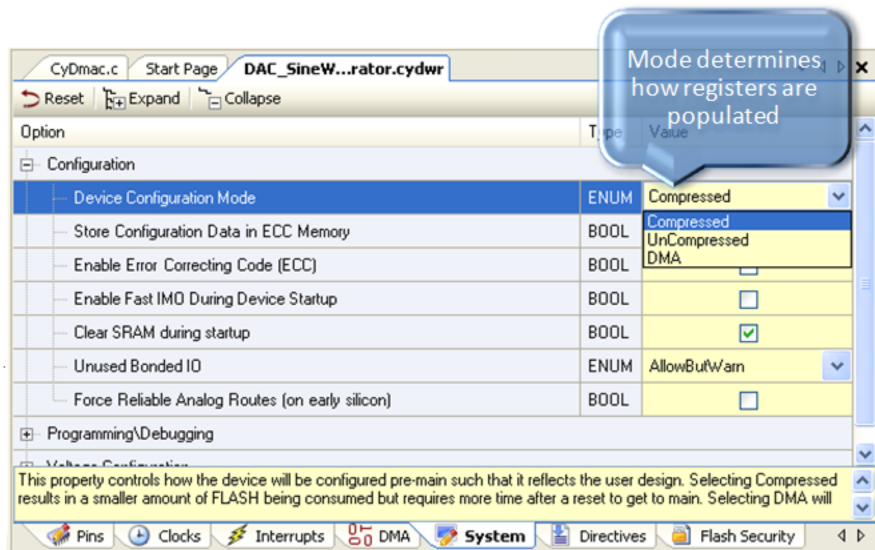


Figure 3.34: Device register mode selection.

3.8.1 PSoC3/5 bootloaders

Once the source code for an embedded system application has been compiled, linked with the appropriate libraries, and debugged[66], it is ready to be downloaded to the target device, e.g., a PSoC3 or PSoC5. This is accomplished, in part, by employing a *bootloader*.[67] A PSoC3/5 bootloader reads data from a communications port and writes it to internal Flash. In addition to downloading an application to a target during the design and manufacturing phase, the ability to download firmware upgrades and bug fixes, in the field, to a target in a noninvasive manner are often very important when employing embedded system applications.

Communication ports in common use in such cases include USB, I2C, UART, JTAG and SWD. However, USB, I2C and UART are often preferred for loading software into a system in the field, rather than SWD and JTAG.[68] In addition, many systems utilize a USB, I2C or UART communication channels, used by the embedded system, to meet other application requirements.

A *bootloader project* is application software loaded by the bootloader into the target's Flash memory.[69] The functions that can be implemented when creating a bootloader are restricted to:

- CyBtldrCommRead - read function
- CyBtldrCommWrite - write function
- CyBtldrCommStart - initiate communication
- CyBtldrCommStop - halt communication
- CyBtldrCommReset - reset the communication channel

[66] Debugging can also be carried out after the executable has been downloaded to the target.
[67] PSoC Creator provides a programmer which employs a default bootloader in the target device. However, in some cases use of this bootloader in the field is undesirable, in which case the designer must provide an appropriate bootloader for field programming/updating.
[68] USB, I2C and UART ports in an embedded system can be used for multiple purposes since they are generic communications protocols whereas SWD and JTAG are somewhat application specific..
[69] There can be only one bootloadable project in use in a PSoC3/5 at a time.

The application code, and associated data, are transferred to the target's Flash memory. The file type created by PSoC Creator for bootloadable files is **.cyacd* and consists of a five byte header, followed by the data records where the header record format consists of a

- [Four byte SiliconID][one byte SiliconRev]

followed by data records in the format given by

- [One-byte ArrayID][Two-byte RowNumber][Two-byte DataLength][N-byte Data][One-byte Checksum]

where the checksum's value is computed by summing all of the bytes, other than the checksum, and then taking the 2's complement of the resulting sum. The SiliconID is a value that identifies the target's package type and the SiliconRev is a value identifying the associated revision number.

The bootloader is responsible for accepting/executing commands, and passing responses to those commands back to a communications component. The bootloader collects/arranges the received data and manages the actual writing of Flash through a simple command/status register interface. The bootloader component is not presented in PSoC Creator as a typical component, i.e., it is not available in the *Component Catalog*. The *communications component* manages the communications protocol used to receive commands from an external system, and passes those commands to the bootloader. It also passes command responses from the bootloader back to the off-chip system.[70]

In order to create a bootloader component, and the associated code, it is necessary to create both a *bootloader* and a *Bootloadable* project in PSoC Creator. When a bootloader project is created, a *bootloader Component* is automatically created by PSoC Creator. The design typically requires dragging a communications component onto the schematic, routing I/O to pins, setting up clocks, etc.

While a standard project resides in Flash starting at address zero, a bootloader project occupies memory at an address above zero and the associated bootloader begins at memory address zero, as shown in Figure 3.35.

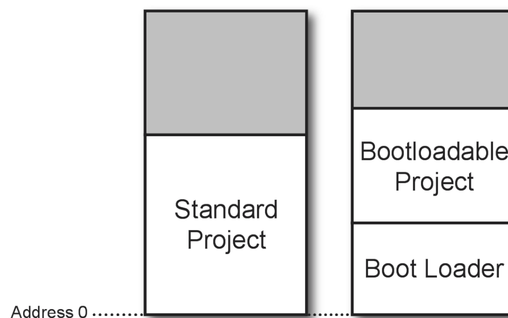


Figure 3.35: Comparison of memory maps for a standard project and a bootloader project.

The bootloader project code transfers a bootloadable project, or new code, to the Flash via the bootloader project's communications component. After the transfer has been completed, the processor is always reset, causing execution of the code to begin at memory address zero. The bootloader project is also responsible, at reset time, for testing for certain conditions and possibly auto-initiating a transfer, if the bootloadable project is non-existent, or is corrupt. At startup,

[70]The I2C is the only supported communication method for the bootloader and the hardware I2C must be selected and not the UDB-based I2C.

the bootloader code loads its respective configuration bytes. It must also initialize the stack and other resources/peripherals involved in the transfer. When the transfer is complete, control is passed to the bootloadable project via a software reset. The bootloadable project then loads configuration bytes for its own configuration; and re-initializes the stack, other resources and the peripherals for its functions. The bootloadable project can call the *CyBtldr_Load()* function in the bootloader project to initiate a transfer[71].

Whether a bootloader or bootloadable project is built, an output file is produced that contains both the bootloader and the bootloadable project. It is used to facilitate downloading both projects via either JTAG, or SWD, to Flash memory in the target device. The configuration bytes for a bootloader project are always stored in main Flash, but not in ECC Flash. However, the configuration bytes for a bootloader project may be stored in either main Flash, or in ECC Flash. The format of the Bootloadable project output file is such that when the device has ECC bytes that are disabled, transfer operations are executed in less time. This is done by interleaving records in the Bootloadable main Flash address space with records in the ECC Flash address space. The bootloader takes advantage of this interleaved structure by programming the associated Flash row once the row contains bytes for both main Flash and ECC Flash. Each project has its own checksum, which is included in the output files at project build time.

Avoiding unintended overwriting of the bootloader can be accomplished by setting the Flash protection settings for the bootloader section of Flash. When the bootloader is built in PSoC Creator, the *Output* window displays the amount of Flash memory required for the bootloader, e.g.,

```
Flash used: 6859 of 65536 bytes (10. 5%).
```

in which case the bootloader occupies 27 rows (ceiling 6859/256) of Flash, i.e., Flash locations 0x000 to 0x1B00. It is protected by highlighting this part of Flash in the Flash Security tab and setting Flash protection as *W-Full protection*.

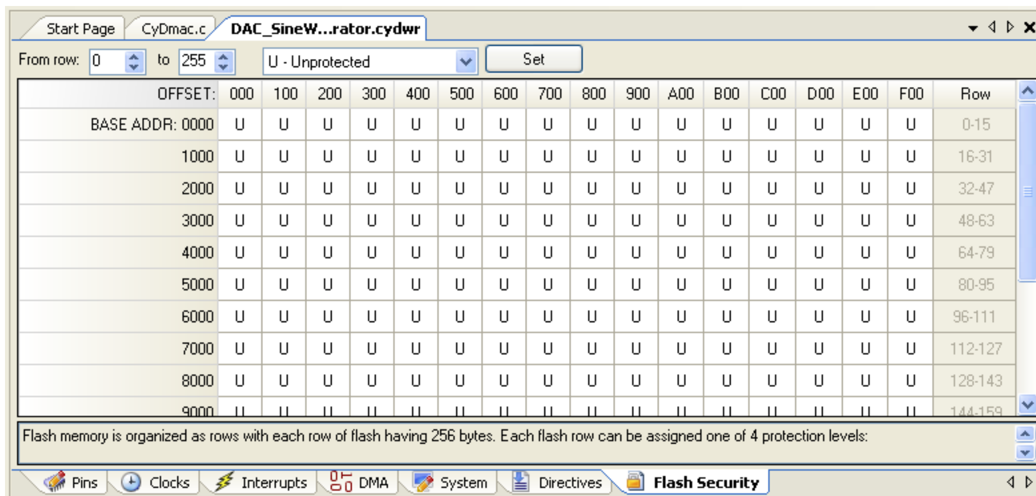


Figure 3.36: Flash security tab.

The bootloader project always occupies the bottom N, 256-byte blocks of Flash, where N is large enough to provide sufficient memory for the

[71] This results in another software reset.

- vector table for the project, starting at address 0 (PSoC5 only),
- bootloader project configuration bytes,
- bootloader project code/data,

and,

- checksum for the bootloader portion of Flash.

The relevant option is removed from the project's *.cydwr* file. The bootloader portion of Flash is protected and can only be overwritten by downloading via JTAG/SWD.

The highest 64-byte block of Flash is used as a common area for both projects. Various parameters are saved in this block, which may include the:

- entry in Flash of the Bootloadable project (4 byte address)
- amount of Flash occupied by the Bootloadable project (Number of Flash rows)
- checksum for the Bootloadable portion of Flash (a single byte)

and

- size of the Bootloadable portion of Flash (4 bytes)

The bootloadable project occupies Flash starting at the first 256-byte boundary after the bootloader, and includes the vector table for the project (PSoC5 only), and the bootloadable project code and data. Storage of the bootloadable project's configuration bytes, in either main Flash or in ECC Flash, is determined by settings in the project's *.cydwr* file. The highest 64-byte block of Flash is used as a common area for both projects. Various parameters are saved in this block, e.g., the entry point in Flash of the bootloadable project (a 4 byte address) the amount of Flash occupied by the bootloadable project (the number of Flash rows) the checksum for the bootloadable portion of Flash (a single byte) and/or the size of the bootloadable portion of Flash (4 bytes).

The only *exception vector* supported by PSoC3 is the 3-byte instruction at address 0, which is executed at processor reset.[72] Therefore, at reset the 8051 bootloader code simply starts executing from Flash address 0. In the PSoC5, a table[73] of exception vectors exists at address 0 and the bootloader code starts immediately after the table. The table contains the initial stack pointer (SP) value for the bootloader project, the address of the start of the bootloader project code and vectors for the exceptions/interrupts to be used by the bootloader. The bootloadable project also has its own vector table, which contains that project's starting stackpointer (SP) value and first instruction address. When the transfer is complete, as part of passing control to the bootloadable project, the value in the *Vector Table Offset Register* is changed to the address of the bootloadable project's table.

- **Wait for Command** - At reset, if the bootloader detects that the checksum in bootloadable project Flash is valid, then it may optionally wait for a command to start a transfer operation before jumping to the Bootloadable project code. If the selection is "yes", then the *Wait for Command Time* parameter is editable. If the selection is "no", then that parameter is grayed out. In that case an external system typically is not able to initiate a transfer, however the Bootloadable project code can still launch a transfer operation by calling *Bootloader_Start()*.[74]

[72]The interrupt vectors are not in Flash. They are supplied by the Interrupt Controller (IC).
[73]This table is pointed to by the *Vector Table Offset Register*, at address 0xE000ED08, whose value is set to 0 at reset.
[74]The default value is "yes".

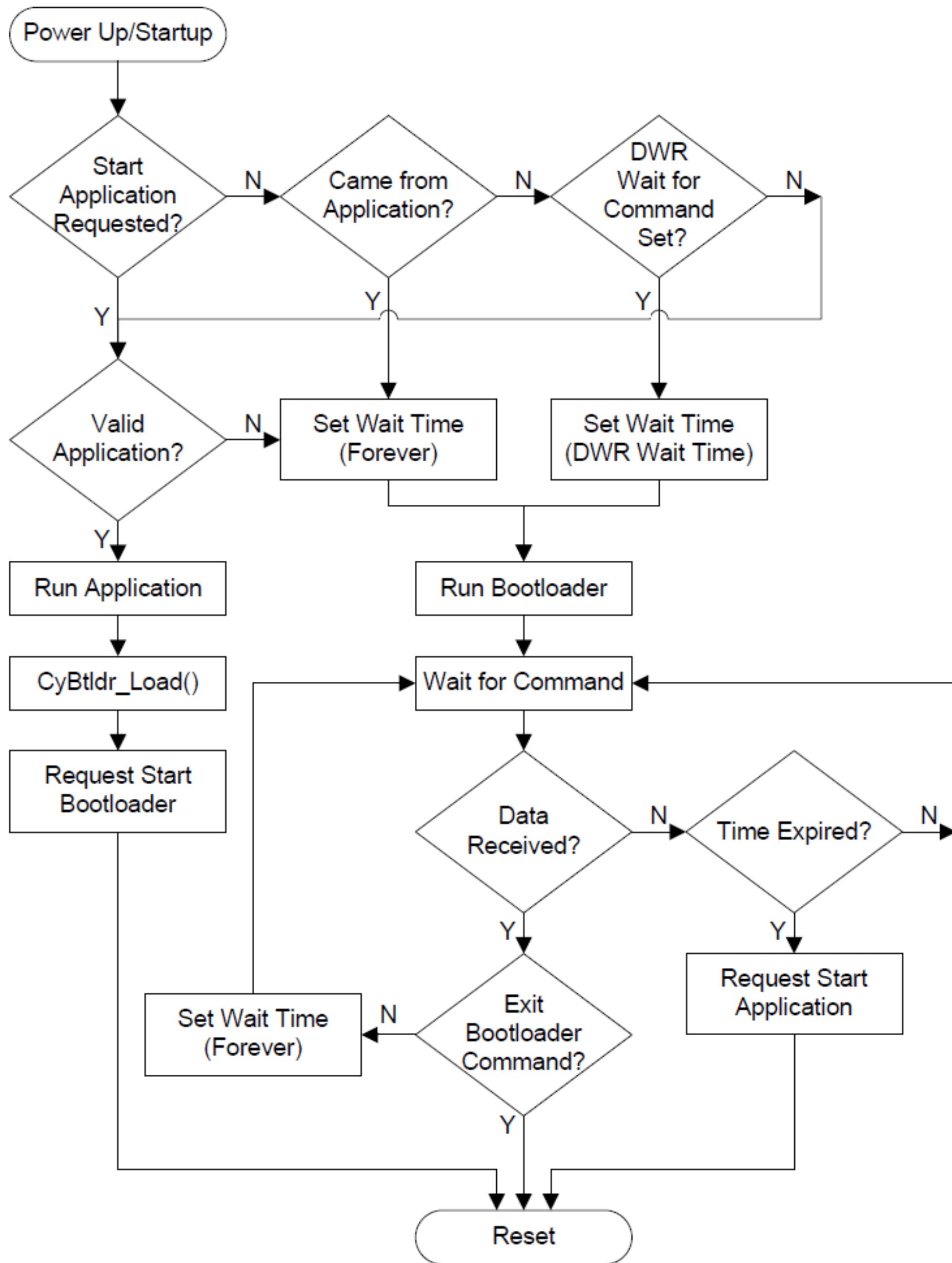


Figure 3.37: Bootloader flow chart.

- **Wait for Command Time** - At reset, if the bootloader detects that the checksum in bootloadable project Flash is valid, then it may optionally wait for a command to start a transfer operation before jumping to the Bootloadable project code. This parameter is the wait timeout period. Allowable settings are 1 -255 (inclusive), in units of 10 msec.[75]
- **I/O Component** - This is the communications component that the bootloader uses to receive commands and send responses. One, and only one, communications component must be selected. Only two-way communications components are used, e.g. a UART must have both RX and TX enabled, and an infrared (IrDA) component could not be used. A *design rule check* (DRC) exists for the case where no two-way communications component has been placed onto the bootloader project schematic. This property is a list of the available I/O communications protocols on the schematic that have bootloader support. There is typically only one communications *Component* on a bootloader project schematic, but there may be more in the case where the bootloader must also perform a custom function during the transfer.[76]

The bootloader has a public API that can only be used to launch a transfer operation from a bootloadable project. When called, a software reset occurs followed by the bootloader taking control of the CPU. Bootloadable code containing interrupts is not executed in this case. When the transfer begins, resources and peripherals are reconfigured as required and all other resources/peripherals are disabled. When the transfer has been completed, the CPU is automatically reset. *void CyBtldr_Load(void)* starts a transfer and reconfigures the device per the bootloader project. Although the CPU is reset upon completion of the transfer there is no return value. Figure 3.37 shows the flowchart for the bootloader.

3.9 Development Tools

The advent of powerful tools such as the integrated development environment (IDE) has allowed designers to create relatively sophisticated designs utilizing little more than a desktop, or portable computer and a so-called "evaluation board", e.g., of the type shown in Figure 3.38, that is based on the target device.[77] Early IDE's consisted of a rather simple text editor, an assembler and linker supported, in some respects, by relatively primitive debugging capability. In time, these system evolved to include various compilers, primitive simulators whose capabilities were generally limited to rather restrictive abilities to check a design's logic, but little else and improved debugging capability.

Debugging, a process which can be the most time consuming aspect of developing a new design, was initially limited to post examination of a region of memory after executing a program that had been downloaded to the target, single-stepping through a program one statement at a time and a rather limited capability to set breakpoints. Later IDE debuggers allowed regions of a program, arbitrary memory locations, registers, etc., to be monitored during, and post, execution to determine whether or not unanticipated, consequential conditions had occurred, as one way to isolate/trap errant code. Some IDEs allowed program variables and expressions to

[75] This parameter is editable only if the *Wait for Command* parameter is set to *yes*, otherwise it is grayed out.

[76] If only one communications component is on the schematic then it is the only one available in the DWR drop down.

[77] Evaluation, or eval, boards are provided by microprocessor/microcontroller manufacturers, often at a nominal cost, to allow designers to become familiar with a device, or family of devices, and in some cases to actually incorporate the eval board into a prototype for testing and proof of concept purposes. Such boards generally include several types of I/O connections, LEDs, various types of switches, display devices such as LED/LCD displays and additional hardware to support whatever is required for on-board programming of the target device.

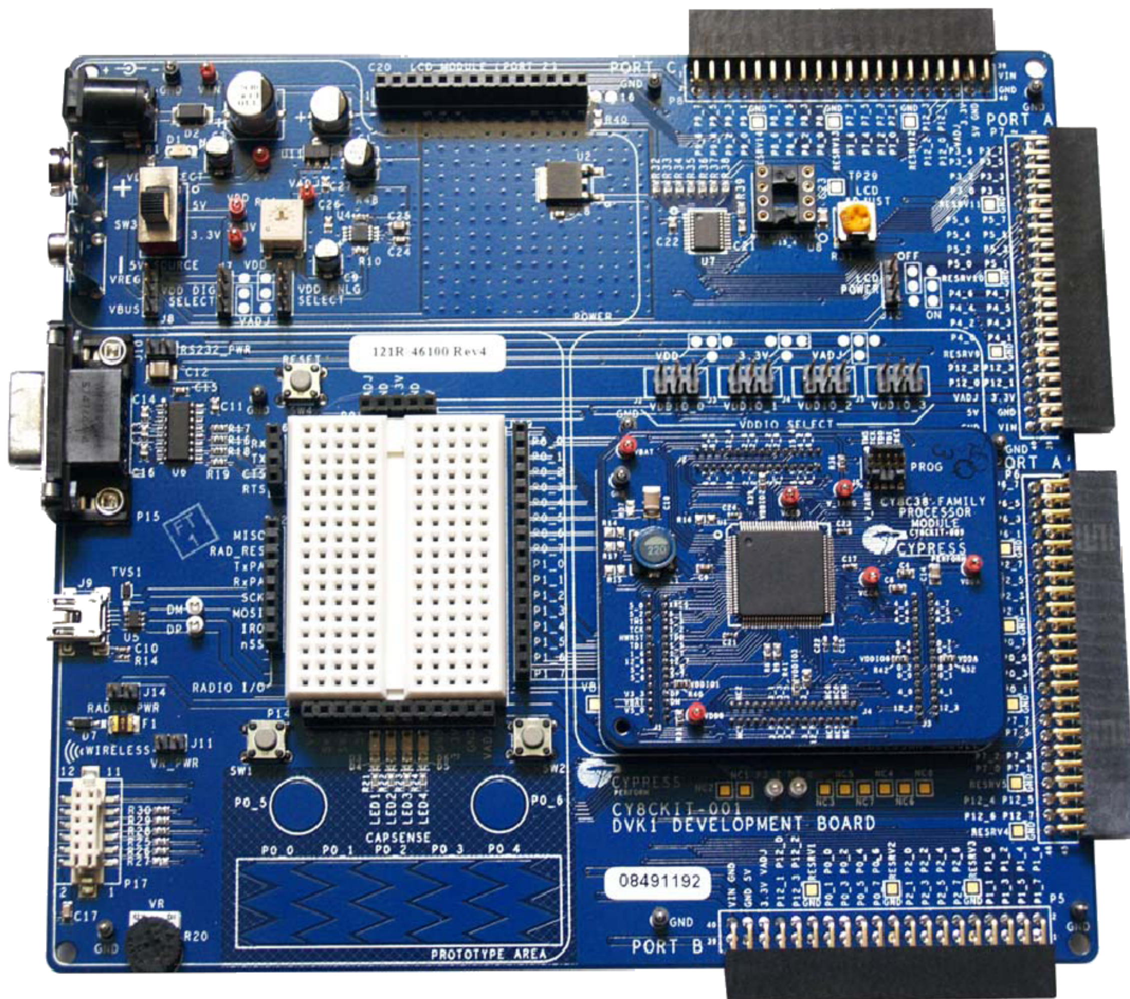


Figure 3.38: The Cypress PSoC1/3/5 evaluation board.

be "watched"[78] and evaluated during program execution.This form of debugging sometimes led to merely "moving the problem to a different place" since such techniques could substantially alter the operating conditions of the executing program, e.g., by introducing too much debugging overhead and adversely affecting the systems responsiveness and execution speed.

Current IDE's for microcontrollers, and microprocessors, tend to primarily support assembly and C language development. However, there are a few notable IDE exceptions that support languages such as BASIC[79], FORTH[80], Pascal, etc.[81] Typically, for applications utilizing C language, the associated compiler produces assembly source code as its output. The resulting assembly source code is then processed by the IDE's assembler and subsequently passed to an integral linker.[82] However, debugging, within the context of an IDE, may be restricted to single-stepping and setting of a limited number of breakpoints.

3.10 The PSoC Creator IDE

PSoC Creator's user-interface is shown in Figure 3.39. It is a combination of a highly intuitive and innovative graphical design editor and a set of sophisticated tools that are well integrated to provide rapid testing of new design ideas, quick response to hardware changes, error-free software interaction with the target's on-chip peripherals, and full access to all aspects of the design. It offers a unique combination of hardware configuration and software development in a single, unified tool. This design frees embedded designers from the innovation-killing division between hardware design and software development characteristic of other IDE systems.

PSoC Creator includes an

- integrated schematic capture for device configuration,
- extensive component catalog,
- integrated source editor,
- built-in debugger,
- C/C++/EC++/Ada compiler support,
- support for component creation (affording design reuse),
- a PSoC 3 compiler - Keil PK51 (no code size limit),
- a PSoC 5 compiler - Sourcery G ®Lite Edition from CodeSourcery,
- sophisticated and reliable bootloading,
- parameter dialogs for comparators, OpAmps, IDACs, VDACs, etc.,
- a static timing checker,
- PSoC 3 instruction cache support,

[78] A *watch window* can be used to evaluate and display variables, registers and/or expressions that involve simple variables, array variables, struct variables, registers and assignments. This window is updated immediately following each halt event and displays the name, value, address, type and radix of the parameter being watched. Memory locations can also be watched.

[79] Beginners All-Purpose Symbolic Instruction Code (BASIC) is an interpreter originally developed by Thomas Kurtz and George Kemeny, in 1964, at Dartmouth College and placed in the public domain. Subsequently, various incarnations were developed, as interpreters or compilers, some of which were compilers that are still used to develop applications for microcontrollers, e.g., BASCOM by MCS for Atmel and 8051 architectures.

[80] VFX FORTH for Windows.

[81] IDEs exist for Ada, C/C++, C#, Eiffel, Fortran, Java and JavaScript, Pascal and Object Pascal, Perl, PHP, Python, Ruby, Smalltalk, etc., but not all are either designed or suitable for embedded system development.

[82] A linker, sometimes referred to as a linkage editor, is used to link-edit various object files into a single file that can be used to produce the resulting executable.

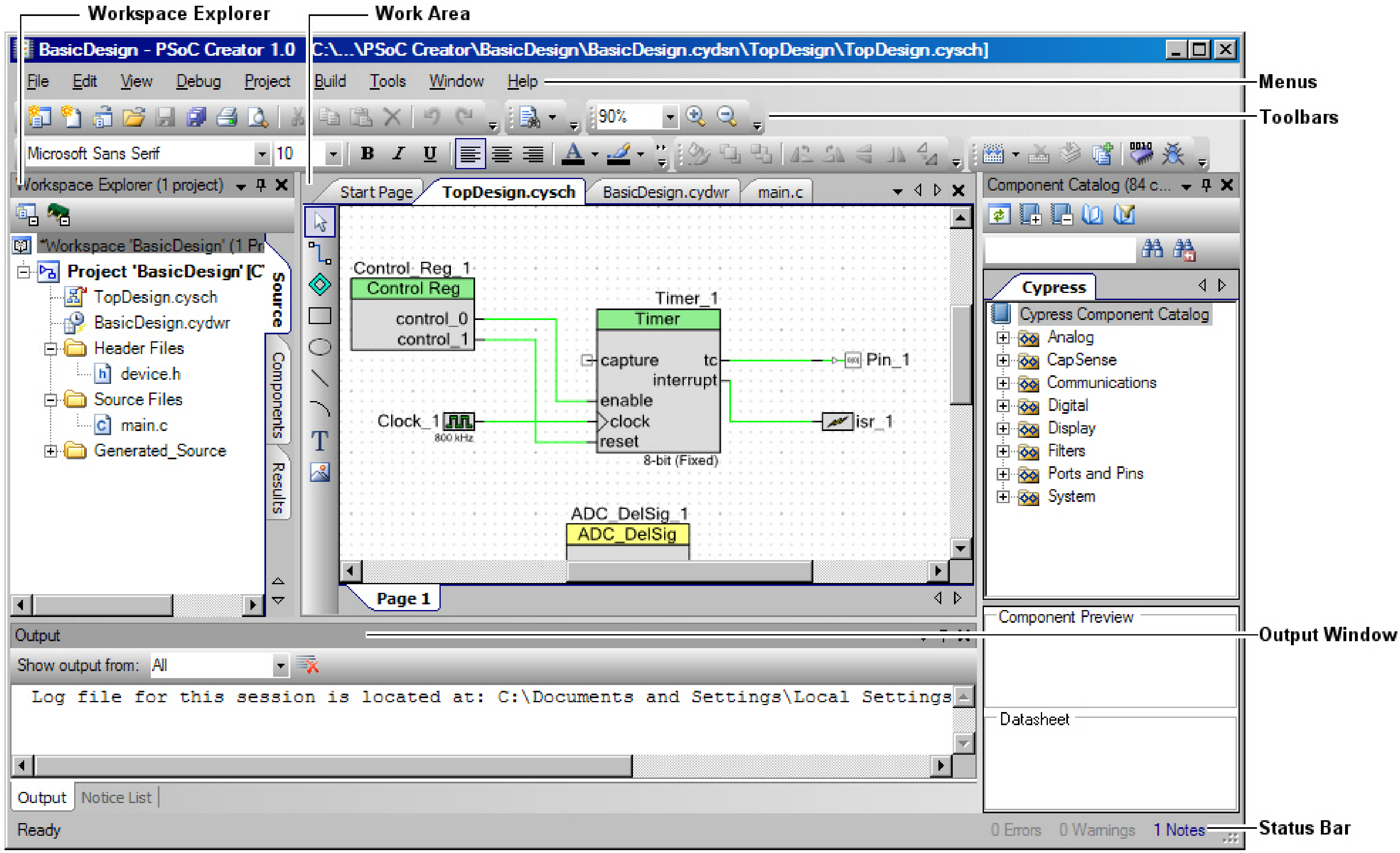


Figure 3.39: The PSoC Creator framework.

- a *Generate Application* command/button,

and,

- automated, support case reporting.

When PSoC Creator is opened, it displays the *Start Page* and provides the user with access to recent projects, new project initiation and information on available updates. Links to tutorials, help files, forums, application notes and the reference-design, build projects available online from Cypress' website (www.cypress.com) are also displayed. If a hardware development kit is attached to the designer's PC it is detected by a scan initiated by PSoC Creator to determine the development kit present, and customizes PSoC Creator's *View* for that hardware kit.

3.10.1 Workspace Explorer

PSoC Creator employs a number of dockable windows and allows such windows to be hidden, at the designer's option, via a toggable, pushpin icon option located in the upper right side of the window. When the window is hidden, a small tab remains that upon the occurrence of a mouse-over causes the respective window to reappear. The *Workspace Explorer* window, shown in part in Figure 3.40, has three tabs: Source, Components and Results. The *Source* tab displays the source and header files for a project in terms of a tree-like structure. Source files displayed in this mode consist of the files generated by PSoC Creator and those introduced by the designer. The *Components* tab displays the components belonging to each project. The *Results* tab is a dynamic listing of files resulting from the most recent build, e.g., programming file, debugging file (if different form the programming file) and in some cases a device file, code generation report, list files and/or map files.

3.10.2 PSoC Creator's Component Library

The *Component Library* includes a wide variety of analog, CapSense. communication, digital, display, filter, port/pin and system components. The designer simply drags each component from the component library to PSoC Creator's work canvas, as shown in Figure 3.41 and connects the various components as required. Double clicking on a component on the canvas causes a dialog box to appear with the available user-selectable options for the device and access to the component's datasheet. When the components have been selected and interconnected as required for a particular design, a build can be initiated. *Warnings*, *Errors* and *Note* are then displayed in the *Notice List* window.

3.10.3 PSoC Creator's *Notice List* and *Build Output* Windows

The *Notice List* window, shown in Figure 3.42 combines notices (errors, warnings, and notes) from many sources into one centralized list. If a file and/or error location is shown, double-clicking the entry will display the error, or warning. There are also buttons to *Go To Error* or *View Details.* This window is usually located at the bottom of the PSoC Creator framework and often in the same window group as the Output window.[83]

The *Notice List* contains the following columns:

- *Icon* - Displays the icons for the error, warning, or note. A specific row may also contain a tree control containing individual parts of the overall message.
- *Description* - Displays a brief description of the notice.

[83]It is possible for a build to fail for no apparent reason and should there be no indication of the cause of failure in the Notice List, the designer should check the Build Output window to determine the cause.

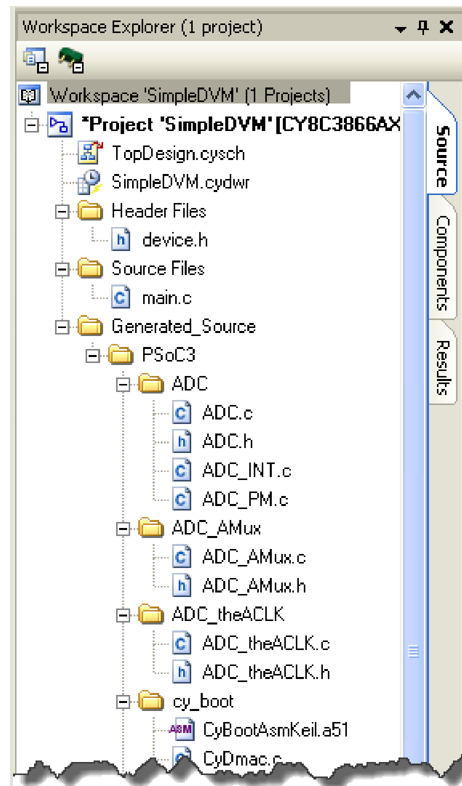


Figure 3.40: Workspace Explorer.

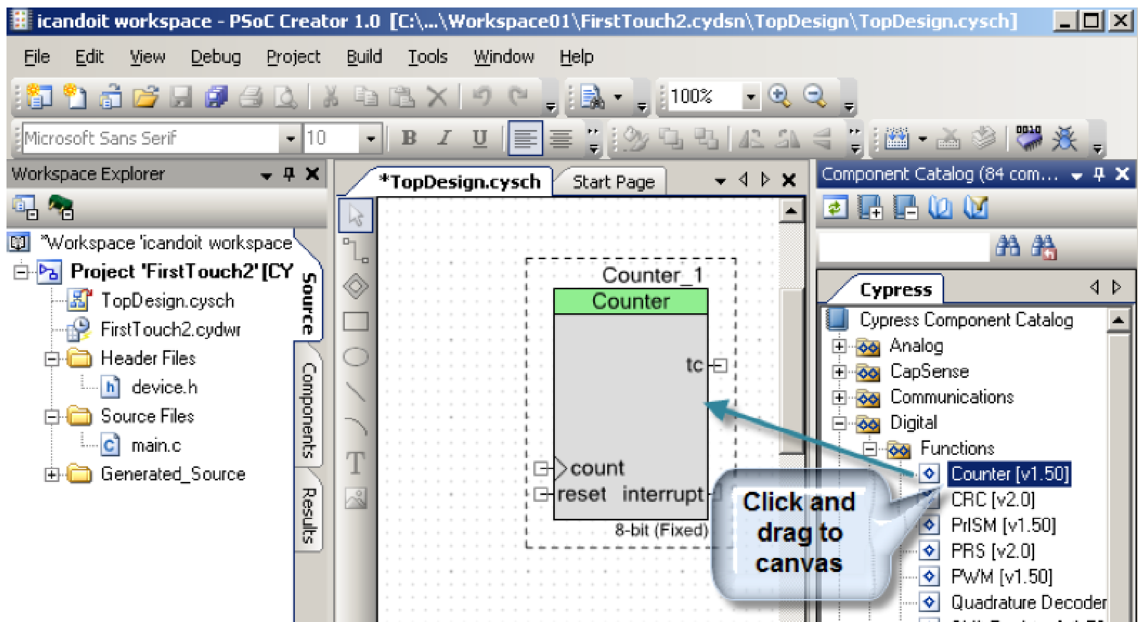


Figure 3.41: Adding a component to a design.

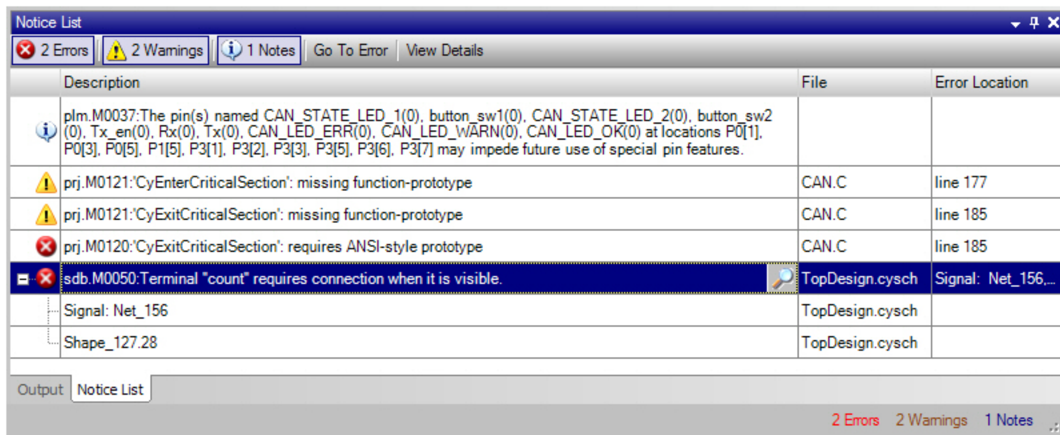


Figure 3.42: PSoC Creator's *Notice List* window.

- *File* - Displays the file name where the notice originated.
- *Error Location* - Displays the specific line number or other location of the message, when applicable.

The number or errors, warnings, and notes also displays on the PSoC Creator Status Bar

- *Errors* indicate there is at least one problem that must be addressed before a successful build can occur. Typical errors include: compiler build errors, dynamic connectivity errors in schematics, and Design Rule Checker (DRC) errors. Errors from the build process remain in the list until the next build.
- *Warnings* report unusual conditions that might indicate a problem, although they may not preclude a successful build.
- *Notes* are informational messages regarding the latest build attempt.

The *File* and *Error Location* columns indicate the file in which an error/warning occurred and its location within that file. The three buttons above the *Notice List* labeled *Errors*, *Warnings* and *Notes* can be used to hide/display items in the notice list for each of the three categories.[84] Double clicking on an error/warning in the *Notice List* opens the associated window and highlights the error. Selecting an error, or a warning, by double clicking on it in the *Notice List* will cause the associated file/screen to open. The *View Details* button will open a window with additional information about the selected warning/error. As design wide resource and schematic errors are fixed, the Dynamic Rules Checker runs and removes the error/warning from the *Notice List*. Other types of errors will not be removed from the *Notice List*, until the next build occurs. Clicking the *Output* tab causes the window to display the various build, debugger, status, log and other messages, as shown in Figure 3.43.

3.10.4 Design-Wide Resources

PSoC Creator provides a design-wide resource (DWR) system that allows the designer to manage all of the resources included in a particular design from one location, as shown in Figure 3.45. Supported resources include clocks, DMA, interrupts, pins, system and directive. Each design has its own default DWR file, with its file type specified as *.cydwr*, and its filename is the same as the project's name. If the *.cydwr* file is deled for any reason the default values will be used. The

[84]These buttons are labeled with the number of errors, warnings and notes, respectively.

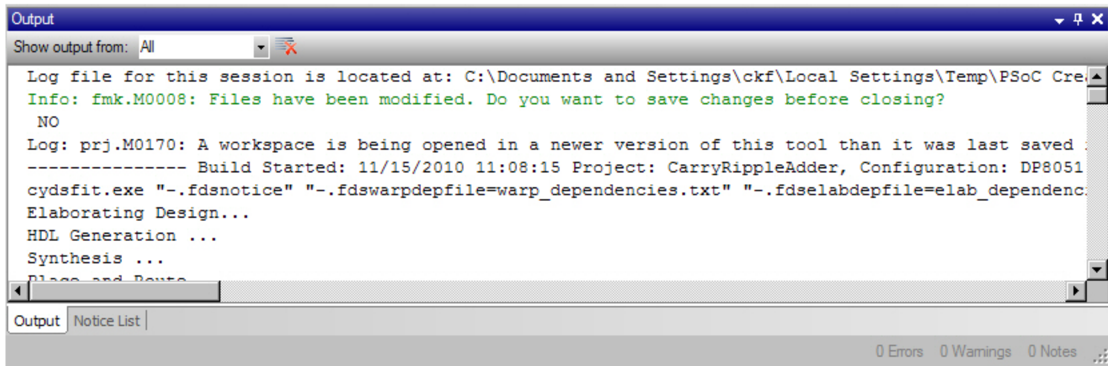


Figure 3.43: PSoC Creator's *Output* window

pin editor, shown in Figure 3.45, allows the pins to be assigned and/or locked[85] prior to the build process' place and route operations. Double clicking the *.cydwr* file in Workspace Explorer's *Source* tab causes the DWR window to open and display the pin editor by default.

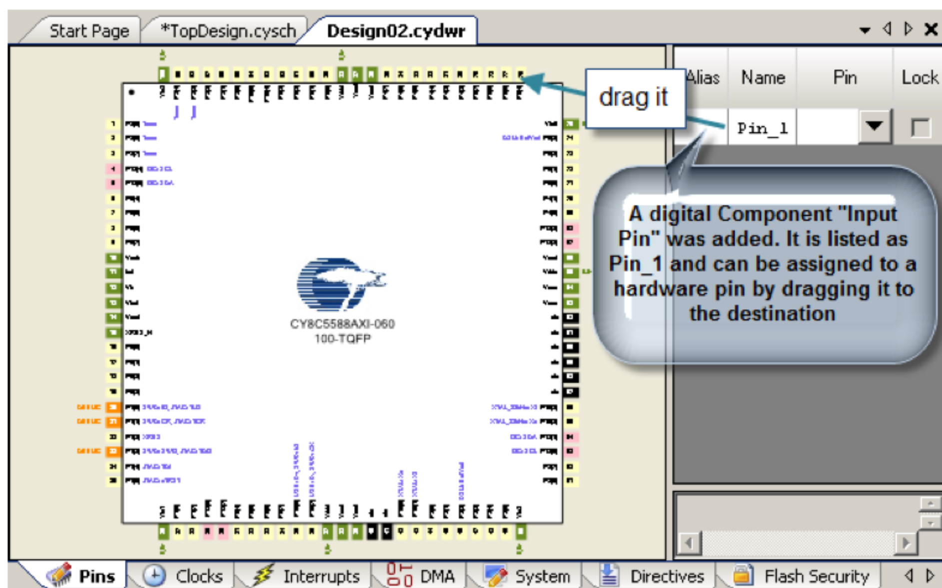


Figure 3.44: Pin assignment.

A signal table is presented in the pin editor that shows the name of each signal, any alias assigned to an individual logic pin or logical port, user-forced pin assignments[86] and an indication of whether or not a particular pin is locked. Pin assignment is illustrated in Figure ref.

[85]Locked pins are constrained to previously specified pin locations. All others are assigned during the build process.

[86]These assignments will not be changed by a build. This column can also be used to make a pin assignment by selecting the desired physical pin from an integral drop-down list. A "-" indicates no assignment has been made for a given signal as does the white background color.

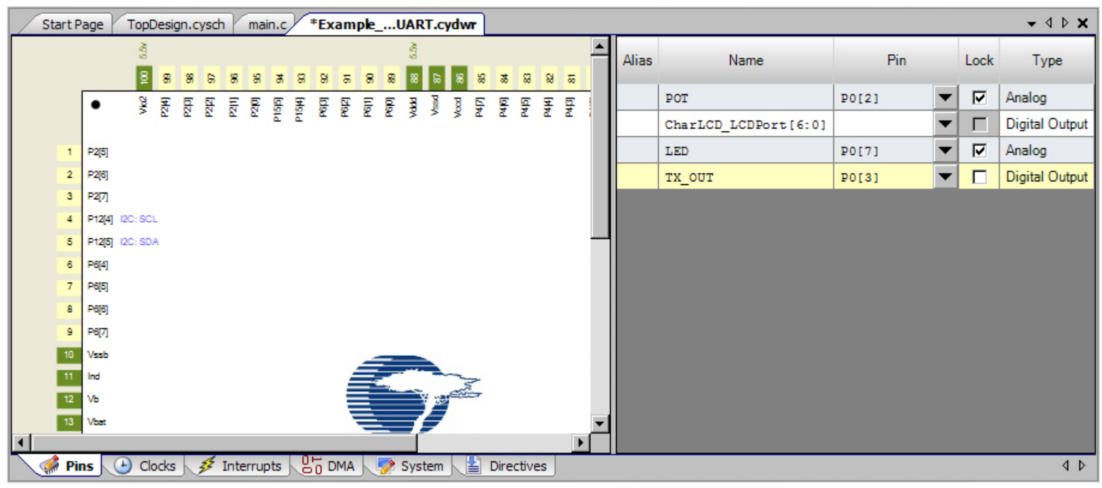


Figure 3.45: The pin assignment table in the DWR window.

3.10.5 PSoC Debugger

PSoC Creator's built-in debugger supports the following commands:

- *Execute Code/Continue* is used to start/continue, start a build if the project is out-of-date, update the status bar's message to indicate that the debugger is starting, program the selected target with the latest version of the projects code and start the debugging session.
- *Halt Execution* halts the target.
- *Stop Debugging* ends the debugging session.
- *Step Into* is used to execute a single line of the source code. If the line is a function call, the break in execution will occur at the first instruction in the function, otherwise a break will occur at the next instruction.
- *Step Over* is used to execute the next line of source code. If the next line is a function call, execution of the function will not occur.
- *Step Out* completes execution of the current function and halts at line of source occurring immediately after the function call.
- *Rebuild and Run* halts the debugging session, recompiles the project, programs the target device and reinitiates the debugger.
- *Restart* resets the program counter (PC) to zero and causes the processor to enter a run state.
- *Enable/Disable All Breakpoints* toggles all of the breakpoints in the workspace.

3.10.6 Creating Components

Although PS0C Creator has an extensive catalog of components, e.g., OpAmps, ADCs, DACs, comparators, a mixer, UARTS, etc., it is possible to add new components. Components can be implemented using several methods, via a schematic, C code or by using Verilog. A *schematic macro* is a mini-schematic that consists of existing components such as clocks, pins, etc. Components created in this manner can consist of multiple macros and macros can have instances,

including the component for which the macro is being defined. PSoC Creator's *Component Update Tool* is used to update instances of components on schematics. When a macro is placed on a schematic the "macroness" of the placed elements disappears. The individual parts of the macro become independent schematic elements. Since there are no "instances" of macros on a schematic, the *Component Update Tool* has nothing to update. However, a schematic macro itself is defined as a schematic. That schematic may contain instances of other components that can be updated via the Component Update Tool.

3.11 Creating a PSoC3 Design

The following example presents the basic steps required in building PSoC3/5 designs using PSoC creator.[87] Following installation and opening of PSoC Creator, navigate to the *New Project* window, shown in Figure 3.46, This design will involve only three components: a Delta Sigma

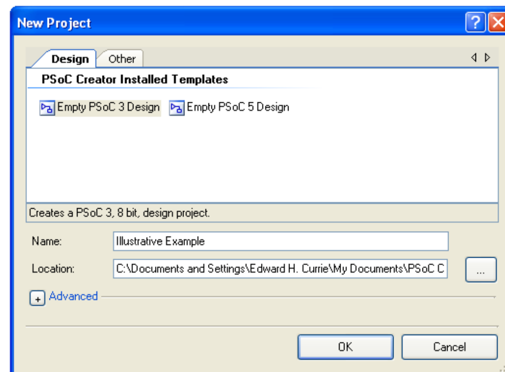


Figure 3.46: PSoC Creator *New Project* Window.

ADC[88], LCD display and analog pin, as shown in Figure 3.47. These components are dragged

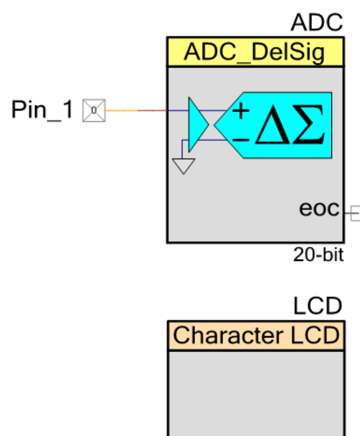


Figure 3.47: PSoC Creator's Analog Pin, LCD and ADC_DelSig components.

[87] A more detailed example is presented in Chapter 7.

[88] This ADC has 8-20 bits of resolution that can be defined in PSoC Creator menus and/or under software control.

from the Component Catalog to the *Workspace Canvas*. The wire tool[89] can then be used to connect the analog pin to positive input terminal of ADC_DelSig. The *Configure 'ADCDelsig'* dialog box, shown in Figure 3.48 is used in this example to select the *Resolution* as 20-bits, the

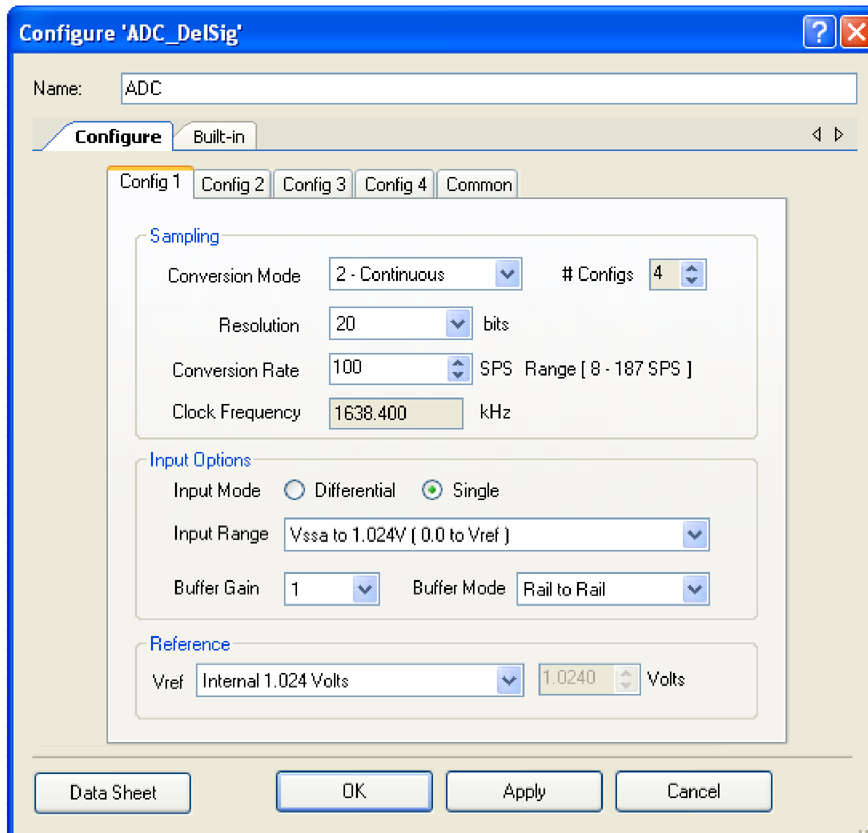


Figure 3.48: ADC_DelSig settings for the simple voltmeter example.

Conversion Rate as 100 samples per second (SPS)[90], the input mode as *single* and the input range as V_{ssa} to 1.024V(0.0 to V_{ref}).

The designer can either select the target manually, or use the Start Auto Select button, shown in Figure 3.51 to programmatically select the appropriate target device, assuming that the target is connected to PSoC Creator. However, in either case, the designer must manually select the associated *Device Revisions* type for the target as *Production, ES2, ES3*[91].

Selecting an analog input pin, and connecting it to the ADC_DelSig's input, completes the physical connections for this design. Double clicking on the associated *.cydwr* tab causes the pin layout for the target device to be displayed as shown in Figure 3.49. A *Build* command [92] is then invoked and main.c clicked on opens the tab *main.c* tab as shown in Figure 3.50. and the screen shown in Figure 3.50. Once the source code has been entered as shown below, and following successful compilation and linking, the resulting executable code can be downloaded to

[89]This tool can be activated by using the keyboard's W key

[90]This selection automatically causes the sampling range to be restricted to a range of 8-187 samples per second (SPS).

[91]The default type is *Production*.

[92]Or alternatively, a *Clean and Build Project* command.

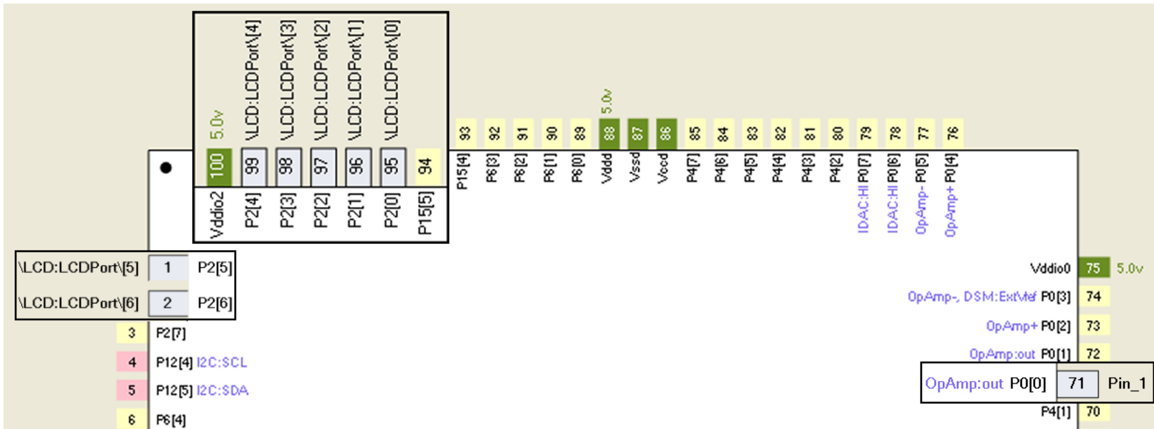


Figure 3.49: Pin connections for the target device.

Illustrative Example - PSoC Creator 1.0 [C:\...\PSoC Creator\Illustrative Example\Illustrative Example.cydsn\main.c]

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include <device.h>

void main()
{
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    /* CYGlobalIntEnable; */ /* Uncomment this line to enable global interrupts. */
    for(;;)
    {
        /* Place your application code here. */
    }
}

/* [] END OF FILE */
```

Figure 3.50: PSoC Creator's main.c tab.

the target by invoking the *Program* option in the *Debug* menu.

The source code for this example is shown below and consists of requiring that the result be expressed as 32-bits and be displayed as a floating point value on the LCD screen. The ADC and LCD must be *started* which requires power to be applied and that they both be initialized. The cursor position is set at $(0, 0)$ and the following message "PSoC Voltmeter" will be displayed on the LCD. A start conversion command will be sent and the system will then enter an infinite loop gathering input readings converting and scaling each input value creating a formatted string and then displaying the result on the 2nd line of the LCD, at which point the process repeats, ad infinitum...

```
/* ========================================
 * PSoC3_Voltmeter
 *
 * Simple project to read a voltage between
 * 0 and 1 volts and display it on an LCD.
 *
 * ========================================
*/
#include <device.h>
#include <stdio.h> /* printf is needed for printing output */

void main()
{
    int32 adcResult;  /* Result to be 32-bit*/
    float adcVolts;   /* Result will be displayed as a floating point value */
    char  tmpStr[25]; /* 25 character temporary string */

    ADC_Start();  /* Initialize and start the ADC */
    LCD_Start();  /* Initialize and start the LCD */
    LCD_Position(0,0);     /* Display message beginning at location (0,0) */
    LCD_PrintString("PSoC VoltMeter");
    ADC_StartConvert();    /* Start ADC conversions */

    for(;;)   /* Loop forever */
    {

      if(ADC_IsEndConversion(ADC_RETURN_STATUS) != 0) /*Data available?
 */
       {
          adcResult = ADC_GetResult32() ;              /* Get Reading (32-bit) */
          adcVolts = ADC_CountsTo_Volts(adcResult);    /* Convert to volts & scale */
          sprintf(tmpStr,"%+1.3f volts", adcVolts);    /* Create formatted string */

           LCD_Position(1,0);                          /* 2nd line of the LCD */
        LCD_PrintString(tmpStr);                       /* Display the result  */
       }
    }
}
```

Note that *sprintf* is used, in this example, to store the resulting string in a buffer named *tmpStr*, as opposed to *printf* which would result in the string being written to the output stream. *LCD_PrintString* subsequently outputs *tmpStr* to the LCD. Many of the components provided by PSoC Creator must be initialized by a start-device instruction. Following entry of the source code for the application into PSoC Creator's editor[93] the *Device Selector* is used to select the target device as shown in Figure 3.51. Pin assignment for the target device is under the control of the designer. The LCD and input pins for this particular design are set as shown in Figure 3.52.

[93] Alternatively, the source can be created by other editors.

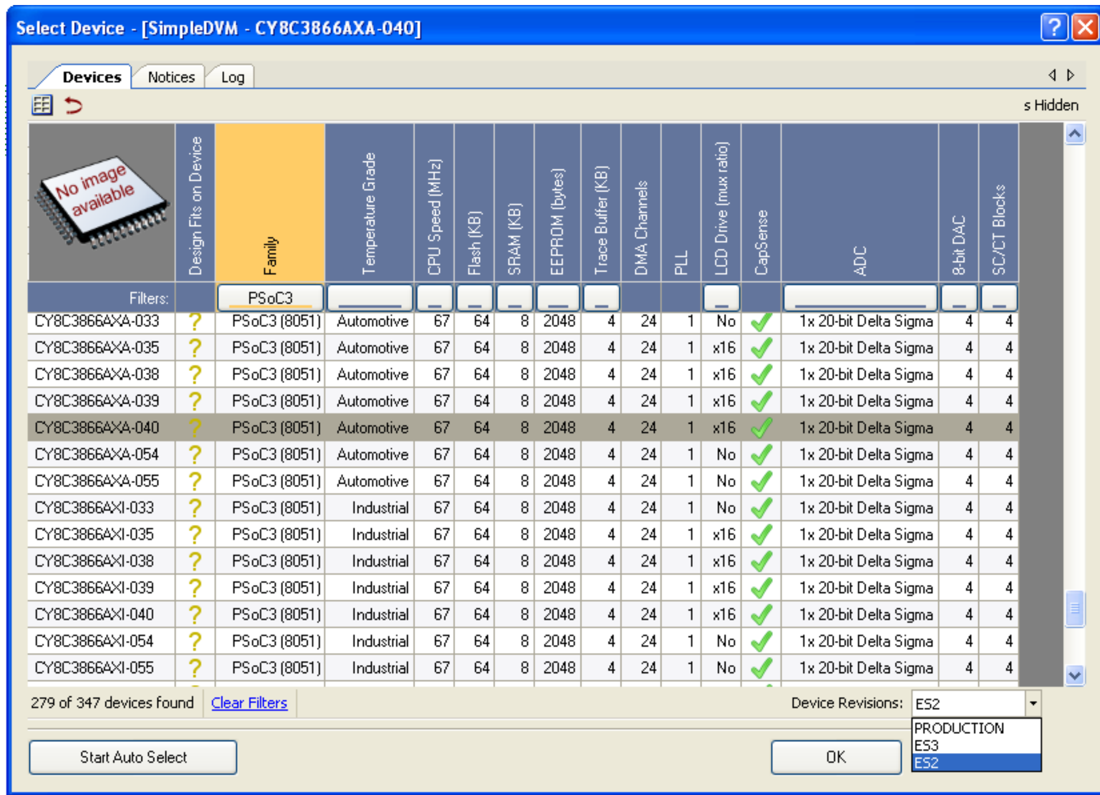


Figure 3.51: This table is used to select the target device/revision type(s).

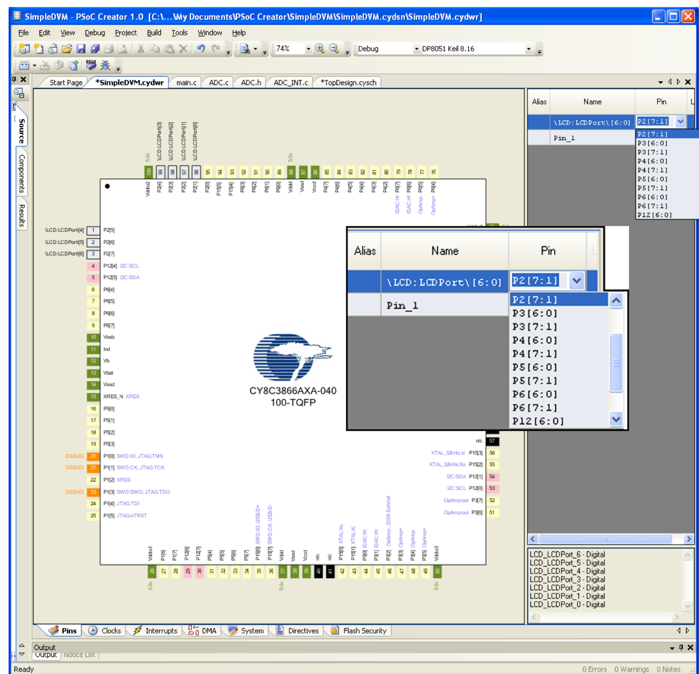


Figure 3.52: Pin assignment for the target device.

3.11.1 Design Rule Checker

PSoC Creator's *Design Rule Checker* (DRC) evaluates the design based on a collection of predetermined rules in the project database. The DRC points to potential errors ,or "rule" violations, in the project that might pose problems and displays the related messages in the *Notice List* window. Some connectivity and dynamic errors update as soon as changes are made to the design, while other errors update following load and save operations.

3.12 The Software Tool Chain

PSoC Creator's integrated development environment (IDE) includes an editor, compiler, assembler, linker, debugger and programmer. The editor is used to enter and/or modify a text file referred to as a *source file*.[94]

After navigating to *Tools>Options>Text Editor*, various options can be set to include line numbers, set the tab size, enable soft tabs/column guides, highlight the current line and set highlight colors for saved/unsaved changes. The editor's *Find and Replace* command options can be set to display informational messages and to automatically populate *Find What* with text from the editor. In addition to supporting C language development, assembly language programming is also supported, either as a separate assembly source file,[95] or as assembly language instructions within a C source file.[96]

The page background can be set by navigating to *Tools>Options>Design Entry>General* and selecting an appropriate *Canvas Background Color*. Terminal options include *Always Enable Terminal Name Dialog*, *Always Show Terminals*, *Schematic Analog Terminal Color*, *Schematic Digital Color*, *Symbol Analog Terminal Color*, *Symbol Digital Terminal Color*, *Terminal Connector Indicator Color*, *Terminal Contact Color*, *Terminal Font* and *Terminal Font Color*. The colors of the major, and minor, grid lines can also be set as can the *Show Grid* and *Show Grid as Lines* options. Analog and digital *Wire Colors*, *Wire Bus Size*, *Wire Dot Size*, *Wire Font*, *Wire Font Color* and *Wire Size* can be chosen by the user. It is also possible to add user-defined sheet templates, *Show Hidden Components* and *Enable Param Edit Views.*

Project management options include setting the *Project location, Always Show the Error List window if a build has errors*, *Always display the workspace in the Workspace Explorer*, *Display the Output window when a build starts*, *Reload open documents when a workspace is opened* and *Reload the last workspace on startup*.

The *Programmer/Debugger* options[97] include: *Ask before deleting all breakpoints*, *Require source files to exactly match the original version* and *Evaluate xx*[98] *children upon expand in variable view*, The *Default Radix* can be set as *Hexadecimal Display, Octal Display, Decimal Display* or *Binary Display.* options include *On Run/Reset run to Reset Vector*, *Main* or *First Breakpoint* and *When inserting software breakpoints, warn: Never*, *On First* or *On Each*, *Disable Clear-On-Read, Automatically reset device after programming, Automatically show disassembly,*

[94]PSoC Creator-compatible source code can also be created by external, third party text editors.

[95]To create a separate assembly file, right click on the project name in the *Workspace Explorer* and select *Add New Item.* Select *8051 Keil Assembly File* and provide a name for the file.This will create an assembly source file, with the extension .a51, in the S*ource Files* in the project.

[96]Inline assembly code is placed between the two directives, #pragma asm and #pragma endasm in the C source file. Right click on the C source file in *Workplace Explorer* and select *Build Settings.* Select the *General* option under *Compiler* and set the *Inline Assembly* option to *True.* The compiler will process the assembly language portion of the source file during compilation.

[97]Navigate to *Tools>Options.*

[98]"xx" is a integer value provided by the user.

after programming if no source is available and Allow debugging even if build failed.

Specific debugger options include *Show Settings for: Breakpoint Windows*, *Call Stack Window*, *Debug Intellipoint*, *Disassembly Window*, *Locals Window*, *Memory Window*, *Registers Window* or *Watch Window.* A wide variety of fonts is provided for debugging and the font size is variable from 6-24 points. *Item foreground* and *Item background* are also user-selectable for *Display items: Plain text*, *Changed Text*, *Changing Text* and *Address Text.*

MiniProg3[99] options include *Applied Voltage: 5.0 V, 3.3 V, 2.5 V, 1.8 V* or *Supply Vtarg; Transfer Mode JTAG, SWD, SWD/SWV or Idle; Active Port 10 Pin or 5 Pin; Acquire Mode: Reset, Power Cycle or Voltage Sense. Debug Clock Speed* is selectable as *200 Hz*, *400 Hz*, *800 Hz*, *1.5 MHz*, *1.6 MHz*, *3.0 MHz*, *3.2 MHz* or *4 MHz. Acquire Retries* is also user selectable. The Environment options include: *Detect when files are changed outside this environment*, *Auto-load changes, if saved, At Startup Show Start Page, external Application Extensions File Extensions and Require Components Update Dialog Check for up-to-date components when a project is loaded*.

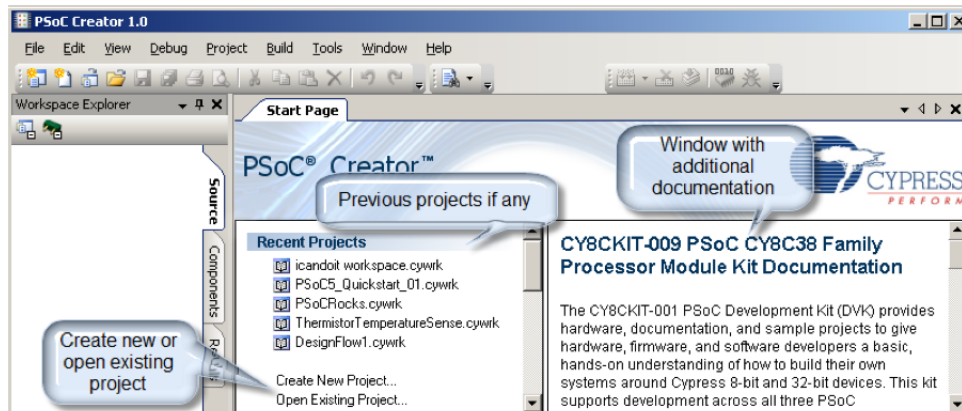


Figure 3.53: *Start Page* in PSoC Creator.

3.13 Opening or Creating a Project

A *project* in PSoC Creator contains all of the information about a given design. When PSoC Creator is invoked, it displays a *Start Page,* as shown in Figure 3.53, that allows the user to either open a previous project or begin a new one. It scans the system for installed development kits and even if none are installed, it will still, if possible, try to configure a device and generate code. However, debugging a project does require the presence of hardware. The basic steps involved in creating any application in the PSoC Creator development environment consists of the following:

- Creation or opening of an existing project - a project consists of a group of files, e.g.,
  1. *TopDesign.cysch* - a schematic layout of the project
  2. main.c - a file con

[99] The PSoC MiniProg3 is an all-in-one programmer for PSoC 1, PSoC 3 and PSoC 5 architectures, that also functions as a debug tool for PSoC 3 and PSoC 5 architectures, and a USB-I2C Bridge for debugging I2C serial connections and communicating to PSoC devices. It supports the following protocols: SWD, JTAG , ISSP and USB-I2C.

- Selection of components to be used in the project - components are selected from the Cypress Component Catalog and dragged to the schematic (.cysch) window.
- Configuration of each of these components - clicking on each of the components will cause the respective dialog box to appear that contains various user options for the component.[100]
- Completion of the schematic - once the required components have been placed in the schematic window the designer can then proceed to incorporate the various interconnections between components that are required.
- Assignment of all resources modification of main.c to allow access to all components used in the design.
- Addition of firmware to main.c
- Building the project
- Downloading of the compiled project
- Debugging the project

by clicking **File>New>Project.**

3.14 Assembly Language and PSoC3

PSoC Creator supports both C and assembly language application development. An assembler translates symbolic instruction code into object code. Assembly language operation codes are incorporated in the source in the form of easily remembered mnemonics, e.g., MOV, ADD, SUB, etc.

Assembler source files consist of:

- Directives that define the program's structure and symbols.
- Assembler controls that set the assembly modes and direct flow.
- Machine instructions are the codes that are actually executed by the microprocessor.

A Linker/Locator links (joins) relocatable object modules created by the assembler or compiler, resolves public and external symbols, and produces absolute object modules, as shown in Figure 3.54. It is also capable of producing a listing file containing a cross reference of external/public symbol names, program symbols and other information.

Figure 3.54: The linking process.

PSoC Creator's integral assembler, AX51, is a multi-pass, macro assembler that translates x51 assembly code source files into object files that can then be combined or linked using PSoC Creator's integral linker/locator, LX51, to produce an executable in the form of an absolute object module in an Intel hex file format. The object module generated by the LX51 Linker is an absolute object module that includes all of the information required for initializing global variables, zero-initializing global variables, program code and constants, as well as symbolic

[100]The configuration and performance characteristics for a given component are defined by values placed in PSoC resource registers associated with the component.

information, line number information, and other debugging details and the relocatable sections assigned and located at fixed addresses.

3.15 Writing Assembly code in PSoC Creator

There are two options available for using assembly code in PSoC Creator projects, viz., create a separate assembly source file, or place inline assembly in a C source file. To create separate assembly code source file: Right click on *Project name* in the Project Explorer and then select *Add new item*, select *8051 Keil Assembly File* and provide a name for the file. This will create an assembly source file with a .a51 file extension in the Source Files folder in the project. Assembly code can then be added to this file using standard 8051 instruction codes.[101]

Inline assembly code can be used by placing the assembly code inside the directive *#pragma asm* and *#pragma endasm*[102], e.g.,

```
extern void test ();
void main (void)  {
    test ();
#pragma asm
    JMP   \$   ; endless loop
# pragma endasm
}
```

In the *Project* explorer, right click on the source file that has the inline assembly and select *Build Settings*. Select the Compiler option and set the value for *Inline Assembly* parameter to *True*. The inline assembly code will then be processed during compilation.]

Assembly language source files consist of lines of instructions of the following general form:

```
label:    mnemonic operand, operand
```

```
$ITLE(Example Assembly Program)
        CSEG     AT 00000h
        JMP $
        END
```

where $TITLE is a directive[103] and CSEG and END are control statements. The assembler supports symbols which consist of up to 31 characters, inclusive. Supported characters include A-Z, a-z, 0-9, underscore and ?.

Symbols can be defined in the following ways:

```
NUMBER_ONE     EQU    1
TRUE_FLAG      SET    1
FALSE_FLAG     SET    0
```

Labels can be used in an assembly language program to define a place, i.e., address, in a program or data space. Labels must begin in the first text field in a line and be terminated by a colon

[101]See PSoC Creator: *Help > Documentation > Keil > Ax51 Assembler User Guide* whihc provides instructions, template, etc., for additional information on assembly language programming.

[102]Pragmas are used in the source code to provide special instructions for the compiler.

[103]There are two types of directives: primary and general. Primary directives occur in the first few lines of the source file and affect the entire source file. General directives can occur anywhere within the source file and may be changed during assembly.

(:). No more than one label may occur per line. and once defined they must not be redefined. Labels can be used the same way a program offset is used within an instruction. Labels can refer to program code, to variable space in internal or external data memory, or can refer to constant data stored in the program or code space. Labels can also be used to transfer program execution to another location.

Labels are defined as follows:

```
ALABEL:    DJNZ    R0, ALABEL
```

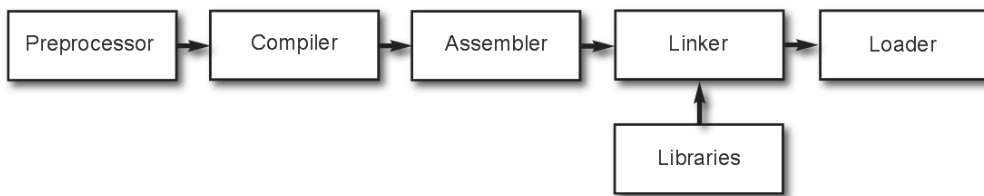


Figure 3.55: The Preprocessor, Compiler, Assembler, Linker and Loader chain.

PSoC3/5's assembler supports the following directives:

- CASE - Enable case-sensitive symbol names. (Primary)
- COND - Include conditional source lines skipped by the preprocessor. (General)
- DATE - Specify the appropriate date in the listing file. (Primary)
- DEBUG - Include debugging information in the listing file. (Primary)
- DEFINE - Defines C preprocessor symbols (command line). (Primary)
- EJECT - Insert a form feed into the listing file. (General)
- ELSE - Assemble the current block, if the condition of a previous *IF* is false. (General)
- ELSEIF - Assemble the current block, if the condition is true and a previous IF is false. (General)
- ENDIF - Ends an IF block. (General)
- ERRORPRINT - Specify the file name for error messages. (Primary)
- GEN - Include all macro expansions in the listing file. (General)
- IF - Assemble block, if the condition is true. (General)
- INCDIR - Set additional include file paths. (Primary)
- INCLUDE - Include the contents of another file. (General)
- LIST - Include the the assembly source text in the listing file. (General)
- MACRO - Enable preprocessor expansion of standard macros. (Primary)
- MOD51 - Enable code generation and define SFRs for classic 8051 devices. (Primary)
- NOAMAKE - Exclude build information from the object file. (Primary)
- NOCASE - Disable case-sensitive symbol names. (All symbols are converted to uppercase.) (Primary)
- NOCOND - Exclude conditional source lines skipped by the preprocessor from the listing file. (Primary)

- NODEBUG - Exclude debugging information from the listing file. (Primary)
- NOERRORPRINT - Disable error messages output to the screen. (Primary)
- NOGEN - Exclude macro expansions from the listing file. (General)
- NOLINES - Exclude line number information from the generated object module. (Primary)
- NOLIST - Exclude assembly source text from the listing file. (General)
- NOMACRO - Disable preprocessor expansion of standard macros. (Primary)
- NOMOD51 - Suppress SFRs definitions for an 8051 device. (Primary)
- NOOBJECT - Disable object file generation. (Primary)
- NOPRINT - Disable listing file generation. (Primary)
- NOREGISTERBANK - Disables memory space reservation for register banks. (Primary)
- NOSYMBOLS - Exclude the symbol tables from the listing file. (Primary)
- NOSYMLIST - Exclude subsequently defined symbols from the symbol table.
- NOXREF - Exclude the cross-reference table from the listing file. (Primary)
- OBJECT - Specifies the name for an object file. (Primary)
- PAGELENGTH - Specifies the number of lines on a page in the listing file. (Primary)
- PAGEWIDTH - Specifies the number of characters on line in the listing file. (Primary)
- PRINT - Specifies the name for the print file. (Primary)
- REGISTERBANK - reserves memory space for register banks. (Primary)
- REGUSE - Specifies registers modified for a specific function. (General)
- RESET - Set symbols to false that can be tested by IF or ELSEIF.
- RESTORE - Restore settings for the LIST and GEN directives. (General)
- SAVE - Save settings for the LIST and GEN directives. (General)
- SET - Sets symbols, that may be tested by IF or ELSEIF, to true or a specified value. (General)
- SYMBOLS - Include the symbol table in the listing file. (Primary)
- SYMLIST - Include subsequently defined symbols in the symbol table.
- TITLE - Specifies the page header title in the listing file. (Primary)
- XREF - Include the cross-reference table in the listing file. (Primary)

3.16 Big Endian vs. Little Endian[104]

Little endian and big endian refer to the ordering of bytes for a particular data format, e.g., big endian refers to situations in which the most significant byte (MSB) occurs first and the least significant byte occurs last. Conversely, little endian implies that the least significant byte occurs first and the most significant byte occurs last.[105]

The PSoC3 Keil Compiler uses the big endian format for both 16-bit and 32-bit variables. However, the PSoC3 device uses little endian format for multi-byte registers (16-bit and 32-bit register). When the source and destination data are organized in different "endian-ness", the DMA

[104] Jonathan Swift allegedly originated the concept of of ended-ness in Gulliver's Travels. It arose as a result of the royal edict regarding which end of an egg should be cracked open.

[105] Some architectures allow the endianess to be changed programmatically, e.g, ARM.

transaction descriptor can be programmed to have the bytes endian-swapped while in transit. The *SWAP_EN* bit of the *PHUB.TDMEM[0..127].ORIG_TD0* register specifies whether an endian swap should occur. If *SWAP_EN* is 1 then an endian swap will occur and the size of the swap is determined by the *SWAP_SIZE* bit of *PHUB.TDMEM[0..127].ORIG_TD0* register. If *SWAP_SIZE = 0* then the swap size is 2 bytes, meaning that every 2 bytes are endian-swapped during the DMA transfer. The code snippet of the TD configuration API to enable byte swapping for 2 bytes of data is given below.

```
CyDmaTdSetConfiguration(myTd, 2, myTd, TD_TERMOUT0\_EN | TD\_SWAP_EN);
```

If SWAP_SIZE = 1 then the swap size is 4 bytes, meaning that every 4 bytes are endian-swapped during the DMA transfer. The code snippet of the TD configuration API to enable byte swapping for 4bytes data is given below.[106]

```
(myTd, 4, myTd, TD\_TERMOUT0\_EN | TD\_SWAP\_EN | TD\_SWAP\_SIZE4);
```

3.17 Reentrant Code

Reentrant code is defined as code that can be shared by multiple processes contemporaneously. In the handling of interrupts, it is fairly common to interrupt a function and allow another process to access the function, e.g., as in the case of a function being called from both the main code and from an interrupt service routine. Declaring a function reentrant preserves the local variables used in the function, when the function is invoked multiple times. In embedded systems, RAM and stack space is often limited and performance is a major concern which mitigates against calling the same function multiple times concurrently.

Functions, including Component APIs, written using the C51 compiler are typically ***NOT*** reentrant. The reason for this limitation is that function arguments and local variables are stored in fixed memory locations due to limited size of the 8051 stack. Recursive calls to the function use the same memory locations, so that arguments and locals could get corrupted.

Reentrant functions can be called recursively, and simultaneously, by two or more processes and are often required in real-time applications, or in situations where interrupt code, and non-interrupt code, must share a function. In spite of the fact that functions are not reentrant by default in PSoC Creator, functions can be declared reentrant by creating a *reentrancy file* (**.cyre*) that specifies which functions are to be treated as reentrant.[28] Specifically, each line of this file must be a single function name.

To create a **.cyre* file for a project the following steps are required:

1. Right click on a project in the Workplace Explorer and select *Add >New Item*
2. Select the *Keil Reentrancy File* to open the file in the editor, as shown in Figure 3.56.
3. This opens a blank page in the code editor with the filetype *.cyre*. Enter the name of each function to be treated as reentrant, e.g., *ADC_Start*, *PWM_Start*, etc., as a single function name per line.

While the *cyre* file cannot be used for reentrant functions that are user-defined, it can be used for PSoC Creator generated APIs. In the case of user-defined functions that are to be treated as

[106] Unlike the PSoC3 KEIL compiler, the PSoC5 compiler uses little endian . Hence the DMA byte-swapping must be disabled when the code is ported to a PSoC5 device.

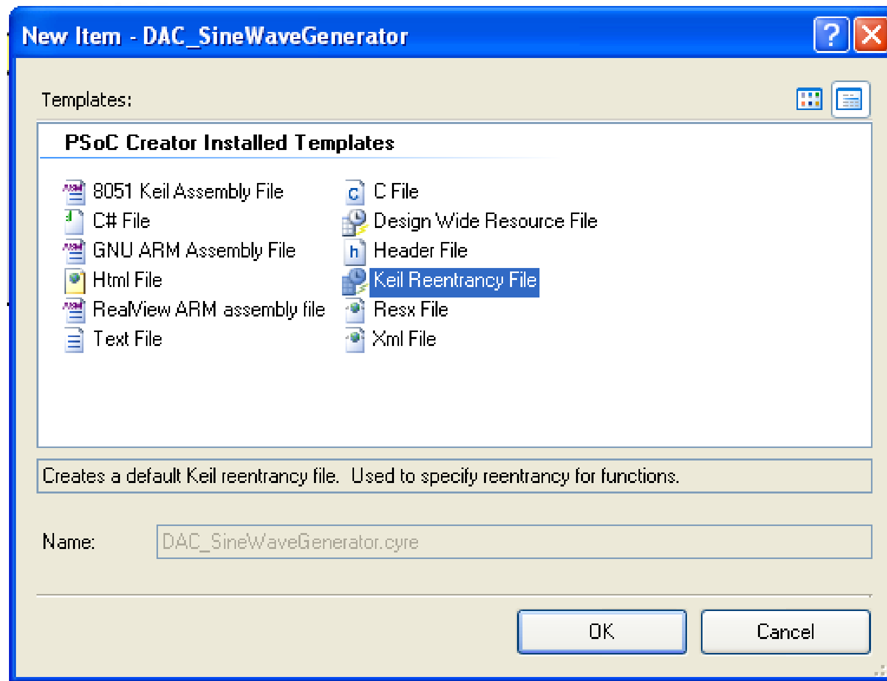


Figure 3.56: New item dialog in PSoC Creator

reentrant, it is necessary to specify the *CYREENTRANT #define* from *cytypes.h* as part of the function prototype. This will evaluate to the reentrant keyword, e.g.,

```
void Foo(void) CYREENTRANT;
```

If a custom component requires reentrancy, the function is declared reentrant by using the *ReentrantKeil* build expression.[107], e.g.,

```
void INSTANCE_NAME{Foo(void)} = ReentrantKeil(INSTANCE_NAMR_Foo);
```

The Keil compiler can be used to determine which functions should be reentrant, if the optimization level is set to 2, or higher and assuming that the functions had not been declared as reentrant in the source file(s). A build will result in the Keil compiler issuing a warning for any such functions that can be called simultaneously. A function should only be marked as re-entrant when the compiler allocates RAM space for the function, in addition to it being called concurrently. An example of a typical warning output by the Keil linker is

```
Warning: L15 MULTIPLE CALL TO FUNCTION
NAME: \_MYFUNC/MAIN CALLER1: ?C\_51STARTUP
CALLER2: ISR\_1\_INTERRUPT/ISR\_1
```

which results from the function *MyFunc* being called from both *main()* and the interrupt service routine isr_1.[108]

[107]By default, the function will be a standard function, unless it is listed in the reentrancy file.
[108]If the function in question is an API function that is to be added to the *.re file, then the function name used in the file should not begin with an underscore and should be expressed as the original case sensitive name for the function.

The Keil compiler establishes a special stack for storage of the reentrant functions arguments and local variables. The associated stack pointer is used to handle multiple calls to the function in a way that assures that each of the calls is handled correctly.[109] The reentrant stack is created in *xdata*, *pdata*, or *idata* space[110], depending on the function's memory model type[111], i.e., *small*[112], *compact*[113], or *large*[114]. Unlike the 8051's hardware stack, the reentrant stack grows downward and therefore it should be initialized at a high address in memory that insures that variables located in lower, fixed-memory locations are not overwritten, as the stack grows. The reentrant stack that corresponds to the memory model being used must be enabled for initialization and the top-of-the-stack address specified in the *KeilStart.A51* file. In this file, the large model, reentrant stack is enabled, In turn. PSoC Creator initializes the large model, reentrant stack pointer to point to the top of SRAM, by default.[115]

```
IBSTACK        EQU 0
XBPSTACK       EQU 1
XBPSTACKTOP    EQU CYDEV_SRAM_SIZE
PBPSTACK       EQU 0
```

Depending on the application, similar changes can be made to the reentrant stack pointer for other memory models. It should be borne in mind that using reentrant code techniques in PSoC3 applications may offer some definite advantages, e.g., a significant reduction in function overhead. However, these same techniques do introduce the additional overhead required to support reentrancy, can result in overwriting other variables in lower memory, and should not be used in firmware, if it requires a significant use of SRAM.

3.18 Building an executable: Linking, Libraries and Macros

Once the source code has been completed, the project can be compiled by either invoking *Build All Projects* in the *Build* menu, or by pressing the *F6* function key, on the keyboard. As the build progresses, any associated errors and/or warnings will be displayed in the *Notice List* window. If the build was successful the message *Build Succeeded* will be displayed.[116]

If the compilation and linking are successful the message *Build Succeeded:* followed by the date and time. The linking phase, among other things, binds symbolic addresses to absolute addresses and shared libraries[117] to specific addresses. All the code in PSoC Creator's *Generated Source* tree is compiled into a single library as part of the build process and the compiled library is linked with the user code.[118]

[109]The stack pointer associated with reentrant functions should not be confused with the 8051's hardware stack pointer (SP) SFR whose value is stored in the SFR register in the 8051 CPU.

[110]*idata*, *pdata* and *xdata* refer to memory located on the chip (RAM), memory addressed with an 8-bit address on an external memory page and external memory (RAM) addressed with a 16-bit address, respectively.

[111]The default memory model is large for stack space (xdata).

[112]The PSoC3 small memory model places function variables and local data segments in internal memory. Although this model imposes a small memory space, it does provide very efficient access to data objects.

[113]The PSoC3 compact model results in all function/procedure variables and local data segments residing in an external memory page (256 bytes) that is addressable via $@R0/R1$.

[114]The large memory model (8051) causes all variables, local data segments to reside in external memory

[115]The use of the large memory model (8051) requires more instruction cycles to access the "large" external (xdata) memory space.

[116]If the *Notice List* window is not visible, the number of errors, warnings and notes are displayed on the status bar.

[117]Binding may be either static or dynamic. In the former case the binding takes place at link time and the latter occurs at runtime. Shared libraries improve runtime and conserves memory.

[118]The GCC Implementation for PSoC5 uses all of the standard GCC libraries, i.e., libcs3, libc, libcs3unhosted, libgcc, which are linked in by default.

Start Page | CyDmac.c | DAC_SineW...rator.cydwr

Add Design-Wide Clock... | Delete Design-Wide Clock | Edit Clock...

Type	Name	Domain	Desired Frequency	Nominal Frequency	Accuracy (%)	Tolerance (%)	Divider	Start on Reset	Source Clock
System	USB_CLK	DIGITAL	48.000 MHz	? MHz	±0	-	1	☐	IMOx2
System	Digital_Signal	DIGITAL	? MHz	? MHz	±0	-	0	☐	
System	XTAL_32KHZ	DIGITAL	32.768 kHz	? MHz	±0	-	0	☐	
System	XTAL	DIGITAL	33.000 MHz	? MHz	±0	-	0	☐	
System	ILO	DIGITAL	? MHz	1.000 kHz	±20	-	0	☑	
System	IMO	DIGITAL	3.000 MHz	3.000 MHz	±1	-	0	☑	
System	BUS_CLK (CPU)	DIGITAL	? MHz	24.000 MHz	±1	-	1	☑	MASTER_CLK
System	MASTER_CLK	DIGITAL	? MHz	24.000 MHz	±1	-	1	☑	PLL_OUT
System	PLL_OUT	DIGITAL	24.000 MHz	24.000 MHz	±1	-	0	☑	IMO
Local	clock_1	DIGITAL	960.000 kHz	960.000 kHz	±1	±5	25	☑	Auto: MASTER_CLK

Pins | Clocks | Interrupts | DMA | System | Directives | Flash Security

Output

Figure 3.57: PSoC Creator Clocks tab.

PsoC Creator provides a comprehensive library of components that can be incorporated into an embedded design. The designer can create additional libraries by using the PSoC Library template. The process begins with the selection of a *Symbol* that represents the new component. After right clicking on the library project[119], *Add Component Item* is selected and a dialog box will appear which allows the designer to select the *Symbol Wizard*. Selecting *Add New Terminals* allows the input and output pins to be defined. It is also possible to specify where in the *Component Catalog* the new component is to be displayed. The functionality of the new component can then be implemented in the form of a schematic, schematic macro[120] or Verilog file.

In addition to the functionality provided for pins as part of the Pins component, a library of pin macros is provided in PSoC Creator's *cypins.h* file. These macros make use of the port pin configuration register that is available for every pin on the device. Macros for read and write access to the registers of the device are also provided. These macros are used with the defined values made available in the generated cydevice.h, $cydevice_trm.h$ and cyfitter.h files.

3.19 Running/Fixing a Program (Debugger Environment)

Debugging is an important aspect of any embedded system development project. PSoC Creator's debugging capability includes real time, full speed, in-circuit emulation using an in-circuit emulator (ICE). This capability allows the designer to monitor an application at the source code level, on a line-by-line-of-source-code basis for both C and assembly language applications. In addition to support for breakpoints, watch variables and *dynamic event points*[121], PSoC Creator also supports the ability to view CPU registers, Flash, RAM and registers and has an 128 kB

[119]This can be located by selecting the *Component* tab of *Workplace Explorer*.

[120]A schematic macro is a mini-schematic allows a new component, that can be multiple macros with multiple elements, e.g., existing components, pins, clocks, etc., to be implemented that includes multiple macros. Macros can have instances (including the component for which the macro is being defined), terminals, and wires. Schematic macros are typically created to simplify usage of the components.

[121]Dynamic event points are a type of complex breakpoint that allows multiple events to be monitored, sequenced and logically unified.

trace buffer[122]. Traces can be turned on, or off, during program execution, via the use of dynamic event points. Trace display options include saving the trace buffer as a file and viewing, saving and/or printing the trace display in the form of an HTML file. This allows the designer to produce a report that can be external to PSoC Creator.

A status bar at the bottom of the PSoC Creator screen displays ICE-related status information. Single-stepping allows the program execution to be executed on a line-by-line basis at the source code level. The contents of the program counter, accumulator, stack pointer or time stamps corresponding to each "step" are stored and displayed in the trace buffer. User-selectable locations in the program source code, called "breakpoints", allow the program to run until it encounters a "breakpoint", at which point program execution halts.[123] PSoC Creator's menu, and/or icon options, allow the program to be restarted at that point. Once a breakpoint is encountered, program execution halts and the CPU updates registers and variable values.

Because the C compiler emits assembly code, PSoC Creator also supports assembly-level C debugging. In this mode, *single-step instruction*, *step-over-a-procedure, step- out-of-a-procedure* and *step-into-assembly* and C-level breakpoint capability are also supported.

In order to provide addition performance benefits, three modes of code optimization[124] are supported by PSoC Creator, viz.,

1. Code compression
2. Elimination of unused User Module (area) APIs
3. Multiply/Accumulate at the hardware level

The types of errors most commonly encountered in developing embedded systems fall into the following categories:

- Corruption of memory by errant code
- Improper use of pointers,
- Hardware design errors,
- Inadequate interrupt handling

and

- So-called "off-by-one" errors

While PSoC Creator provides excellent facilities for identifying, isolating and locating errors, good coding practices should be employed to minimize debugging time.

[122]A trace buffer of this types maintains a record of the most recent instructions that were executed in a time sequenced 128k buffer. This allows the designer to follow the precise order of execution of instructions while the system was operating in real time and at full speed.

[123]It should be noted that while the program is halted at user-specific locations in the program code, the code at that location is not executed until execution resumes.

[124]Optimization, in the present context, refers primarily ro reduction in code size and execution speed. Many compilers such as the Keil compilers offer various levels of optimization.

3.20 Programming the Target Device

PSoC3/5 can be programmed by using the Cypress MiniProg3. This device supports ISSP[125], SWD[126] and JTAG[127] programming protocols, as well as I2C[128], SWV[129], ISSP[130], SWD[131] and JTAG interfaces therefore it serves as a protocol converter between a PC and the target device, when connected as shown in Figure 3.58. It should be noted that it is designed to communicate

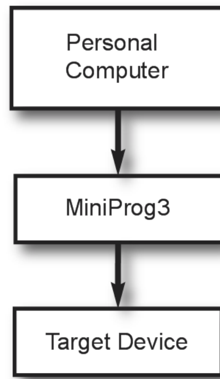


Figure 3.58: MiniProg3 handles protocol translation between a PC and the target device.

with target devices that use only I/O voltages in the range from 1.5 to 5.5 volts.

MiniProg3 has five LEDs that indicate status, and are labeled as follows:

- Busy (Red) - indicates that an operation, such as programming or debugging, is in progress.
- Status (Green) - indicates that the MiniProg3 has been "enumerated" on the USB bus and will flash when it receives USB traffic.
- Target Power (Red) - indicates that the MiniProg3 is supplying power to to the target's connectors.
- Aux (Yellow) - Reserved.
- Unlabeled (Yellow) - indicates configuration of the MiniProg3. It flashes during initial configuration of the MiniProg3 and illuminates continuously when a configuration error has occurred. Following a configuration error, the MiniProg3 must be disconnected from the USB port and reconnected.

[125] *In-System Serial Programming* (ISSP) is a Cypress legacy interface used to program the PSoC1 family of microcontrollers.

[126] SWD uses fewer pins of the device than JTAG. MiniProg3 supports programming and debugging PSoC 3/5 devices, using SWD.

[127] *JTAG* is supported by many high end microcontrollers, including the PSoC 3/5 families. This interface allows multiple JTAG devices to be daisy-chained.

[128] A common serial interface standard is the *Inter-IC Communication* (I^2C) standard by Philips. It is mainly used for communication between microcontrollers and other ICs on the same board, but can also be used for intersystem communications. MiniProg3 implements an I2C multimaster host controller that allows the exchange of data with I2C-enabled devices, on the target board. This feature may be used to tune CapSense designs.

[129] The Single Wire Viewer (SWV) interface, is used for program and data monitoring, where the firmware may output data in a method similar to "printf" debugging on PCs, using a single pin. MiniProg3 supports monitoring of PSoC 3/ PSoC 5 firmware, using SWV, through the 10-pin connector and in conjunction with SWD only.

[130] In-System Serial Programming (ISSP) is a Cypress legacy interface used to program the PSoC1 family of microcontrollers. MiniProg3 supports programming PSoC1 devices through the 5-pin connector only.

[131] *Serial Wire Debug* (SWD) provides the same programming and debug functions as JTAG, except for boundary scanning and daisy chaining.

3.21 Intel Hex Format

The Intel Hexadecimal 8-bit object format is a representation of an absolute binary object file in an ASCII format. Firmware that is produced by PSoC Creator, for the PSoC3/5 microcontrollers, is downloaded in the Intel Hex format, i.e., the downloaded file consists of six parts as shown in Figure 3.59 and defined as:

Start Code (Colon Character)	Byte Count (1 Byte)	Address (2 Bytes)	Record Type (1 Byte)	Data (N Bytes)	Checksum (1 Byte)

Figure 3.59: The Intel hex file format.

1. *Start Code* - a single character, viz., an ASCII colon (:)[132]
2. *Byte Count* - two hex digits representing a single byte that specifies the number of bytes in the data field.
3. *Address* - four hex digits, represented by two bytes, that specify the starting address of the memory location for the data.
4. *Record Type* - two hex digits with values between 00 and 05, inclusive, that define the data field.

   PSoC Creator generates the following record types:

   - 00 indicating a data record containing data and a 16-bit address
   - 01 indicating an end of a file record referred to as a *file termination record*. This record contains no data and can only occur once for each file.
   - 04 indicates an extended linear address record for full 32-bit addressing. The address field is defined as 0000 and the byte count as 2 bytes. The two data bytes represent the upper 16 bytes of a 32-bit address, when combined with the lower 16-bit address of the 00 record.
5. *Data* - a sequence of N data bytes and represented by 2N hex digits.
6. *Checksum* - two hex digits representing a single byte which is the least significant byte of the two's complement of the sum of the values of all of the fields, except for the first and last fields, i.e. except for the start code and the checksum.

Examples of the various record types used by PSoC Creator are:

- :0200000490006A - an extended linear address record, as indicated by the value in the record type field (04). The associated address field is 0000 representing two data bytes. The upper 16-bit portion of the 32-bit address is given by 9000, therefore the base address is 0x90000000 and 6A is the value of the checksum.
- :0420000000000005F7 - this represents a data record, as indicated by the value 900 in the record data type field, and the byte count is 04, i.e. there are four data bytes in this record (00000005). The lower 16-bit value of the 32-bit address, as specified in the address field of this record, is 2000 and F7 is the *checksum* for this record.
- :00000001FF - this is the last record and is therefore an end-of-file record, as defined by the value 01 in the record type field.

[132]The American Standard Code for Information Interchange (ASCII) is a numerical representation of alphabetic and special characters originally developed as a seven bit telegraph code. It consists of numeric definitions for 128 characters, 94 of which are "printable' and 33 are non-printable such as control characters (line feed, carriage return, etc.) and the "space". The numeric value for colon in the ASCII code is 58 (03AH).

3.21.1 Organization of Hex File Data

The hex file generated by PSoC Creator contains different types of data[133], e.g., the main Flash data, ECC data, Flash protection data, customer nonvolatile latch data, write-once latch data and metadata.[134] All of this information, including metadata, is stored at specific addresses, as shown for example in Figure 3.60 for PSoC5. This allows the designer to identify which data is meant for what purpose.

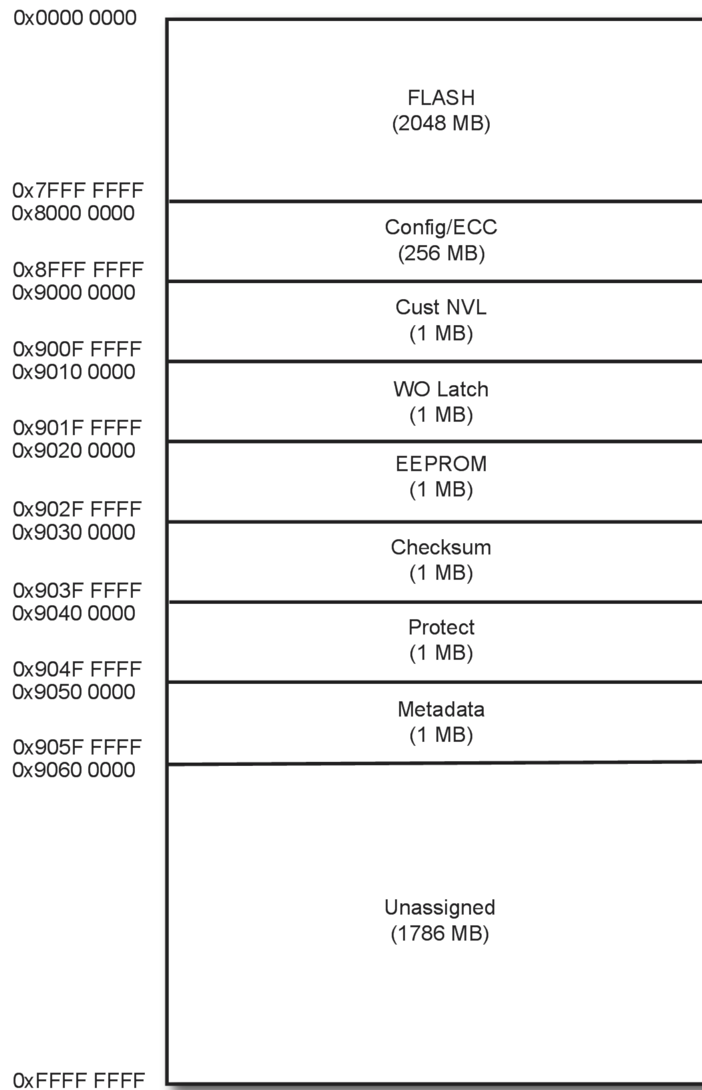


Figure 3.60: Hex file memory locations for PSoC5.

- 0x0000 0000 Ũ Flash Row Data: The main *Flash data* starts at address 0x0000 0000 of the

[133]The data records are in big-endian format (MSB byte in lower address), e,.g., the checksum data is address 0x90300000 of the hex file, and the metadata at address 0x9050 0000 of hex. The data records in the rest of the multi-byte regions in a hex file are all in little-endian format (LSB byte in lower address).

[134]Metadata is information contained in the hex file that is not used for programming. It is used to maintain the data integrity of the hex file and store silicon revision and JTAG ID information of the device.

hex file. Each record in the hex file contains 64 bytes of actual data arranged into rows of 256 bytes. This is because each Flash row of the device is of length 256 bytes. The last address of this section depends on the Flash memory size of the device for which the hex file is intended.[135]

- 0x8000 0000 - *Configuration Data (ECC)*: PSoC 5 devices have an error correcting code (ECC) feature, which is used to correct, and detect, bit errors in main Flash data. There is one ECC byte for every eight bytes of Flash data. Thus, there are 32-bytes of ECC data for each row of flash. There is an option to use the ECC memory to store configuration data if the error correcting feature is not required. The ECC enable bit, in the device configuration *NV latch* (bit 3 of byte 3), can be used to determine, whether or not, ECC is enabled. The *NV latch* data byte is located at address 0x90000003. PSoC Creator generates this section of the hex file only if the ECC option is disabled. If this section is present in the hex file, the data needs to be appended with the main Flash data during the Flash programming step. For every 256 bytes in Program Flash, 32 bytes from this section are appended. The last address of this section depends on the device Flash memory capacity. A device with 256 KB of Flash memory has 32 KB of ECC memory. In which case, the last address is 0x80007FFF.
- x9000 0000 - *Device Configuration NV Latch Data*: There is a 4-byte device configuration, nonvolatile latch that is used to configure the device, even before the reset is released. These four bytes are stored in addresses starting from 0x90000000. One important bit in this NV latch data is the ECC enable bit (bit 3 of byte 3 located at address 0x90000003). This bit determines the number of bytes to be written during a Flash row, write process.
- 0x9010 0000 - *Secured Device Mode Configuration Data*: This section contains four bytes of the write-once nonvolatile latch data that is to be used for enabling device security.[136] PSoC Creator generates all four bytes as zero, if the device security feature has not been enabled, to ensure that there is no accidental programming of the latch with the correct key. Failure analysis support may be lost on units after this step is performed with the correct key.
- 0x9030 0000 Ũ *Checksum Data*: This 2-byte checksum data is the checksum computed from the entire Flash memory of the device (main code and configuration data, if ECC is disabled). This 2-byte checksum is compared with the checksum value read from the device to check if correct data has been programmed. Though the *CHECKSUM* command, sent to the device, returns a 4-byte value, only the lower two bytes of the returned value are compared with the checksum data in the hex file. The 2-byte checksum in the data record is in Big-endian format (MSB byte is first byte).
- 0x9040 0000 Ũ *Flash protection data*: This section contains data to be programmed to configure the protection settings of Flash memory. Data in this section should be arranged in a single row, to match the internal Flash memory architecture. Because there are two bits of protection data for each main Flash row, a 256 KB Flash (which has 1024 rows, 256 rows in each of four 64K Flash arrays) has 256 bytes of protection data.
- 0x9050 0000 Ũ *Meta Data*: The data in this section of the hex file is not programmed into the target device. It is used to check the data integrity of the hex file, store the silicon revision value for which the hex file is intended, and so on. The different data in this section is tabulated as shown in Table 3.2.

[135]Reference should be made to the respective device datasheet, or the Device Selector menu in PSoC Creator to determine the specific FLASH memory size for different part numbers.

[136]Programming the write-once NV latch with the correct 32-bit key locks the device. This step should only be performed, if all prior steps passed without errors.

Table 3.2: PSoC 5 hex file metadata organization.

Starting Address	Data Type	Number of Bytes
0x9050 0000	Hex file version	2 (big-endian)
0x9050 0002	JTAG ID	4 (big-endian)
0x9050 0006	Silicon Revison	1
0x9050 0007	Debug Enable	1
0x9050 0008	Internal Use	4

- *Hex file version*: This 2-byte data (big-endian format) is used to differentiate between different hex file versions, e.g., if new metadata information or EEPROM data is added to the hex file generated by PSoC Creator, there is a need to distinguish between the different versions of hex files. By reading these two bytes it is possible to ascertain which version of the hex file is going to be programmed.
- *JTAG ID*: This field has the 4-byte JTAG ID (big-endian format), which is unique for each part number. The JTAG ID read from the device should be compared with the JTAG ID present in this field to make sure the correct device for which the hex file is intended is programmed.
- *Silicon Revision*: This 1-byte value is for the different revisions of the silicon that may exist for a given part number. The byte stored in the hex file should match the value in the chip's *MFGCFG.MLOGIC.REV_ID* register.
- *Debug Enable*: This byte stores a Boolean value indicating, whether or not, debugging is enabled for the program code. (0/1 implies that debugging disabled/enabled.)
- *Internal Use*: This 4-byte data is used internally by PSoC Programmer software. It is not related to actual device programming and need not be used by third-party hardware programmers..

3.22 Porting PSoC3 Applications to PSoC5

As discussed previously, PSoC3 and PSoC5 are both powerful microcontrollers, the former being based on an 8-bit, 8051 class of microprocessor architecture (33 MIPS) and the latter on a 32-bit, ARM Cortext-M3 (100 DMIPS)[137]. Some of the more important differences between the two architectures are shown in Table 3.3.

There may be cases in which it would be of interest to port a PSoC3 application [22] to a PSoC5 environment, perhaps to gain additional performance benefits, take advantage of the 32-bit architecture, deploy components unique to PSoC5, etc.[138] Although the memory maps for the two devices are quite different, in part, as a result of the differences in their respective CPU architectures, the initial phase of such a port can be accomplished by simply using PSoC

[137]MIPS refers to the millions of CPU instructions executed per second and is not as quantitative a measure of the speed of a processor as DMIPS, which refers to the millions of Dhrystones executed per second. The Dhrystone benchmark is a small integer-based program that is an established benchmark for processors of all types. While both are of some use in comparing processors, they re not the final arbiter of a processor's potential performance in a specific application.

[138]8051 assembly cannot, as a practical matter, be ported directly to the Cortex-M3 space.

Table 3.3: Comparison of key differences between PSoC3 and PSoC5.

Feature	PSoC3	PSoC5
Processor	8-bit 8051	32-bit ARM Cortex-M3
ADC	One DelSig ADC	One DelSig ADC Two SAR ADCs
Flash	Up to 64 KB (inclusive)	Up to 256 KB (inclusive)
RAM	Up to 8 KB (inclusive)	Up to 64 KB (inclusive)
Cache	No	Instruction Cache
EMIF	Data Memory	Data/Code Memory

Creator and navigating to *Project* > *Device Selector*, selecting the targeted PSoC5 device from the table shown in Figure 3.61 and then rebuilding the project.[139]

However, if the port is to be optimized in the new target environment, a number of factors need to be taken in consideration, e.g., PSoC3's memory map consists of three different code spaces as shown in Figure 3.62.

1. The 8051 internal data space, which is part of the 8051 core, contains 256 bytes of RAM and 128 bytes of special function registers (SFRs). This space is accessed by the *fast* registers and bit instructions and the location of the 8051 hardware stack ($\geqslant 256$ bytes).
2. The external data space, while internal to the PSoC3, is external to the 8051 core. All SRAM, Flash[140], registers and EMIF[141] addresses are mapped into this space. The assembly instruction MOVX is used to access the external data space which is 16 Mb in size and requires a 24-bit access address.
3. The code space is 64K bytes of Flash memory and it is here that the 8051 instructions reside.

The PSoC5 memory space is based on a 32-bit, linear memory map, as shown in Figure 3.63. PSoC5's SRAM is located in the memory space defined by [0x1FFF8000, 0x20007FFF] and is centered on the boundary between the code and SRAM memory spaces. The rest of the code space is occupied by Flash beginning at memory address 0. PSoC5 registers are located in the peripheral space(s) and the EMIF addresses are located in the external RAM space. The Keil compiler uses the big endian format for 16- and 32-bit variables for PSoC3 and little endian for PSoC5 multi-byte variables.[142]

3.22.1 CPU Access

PSoC Creator supports the following macros to allow register access with byte swapping. These macros are for accessing registers mapped in the first 64K bytes of the 8051 external data space:

```
CY_GET_REG8(addr)
CY_SET_REG8(addr, value)
```

[139]PSoC Creator supports three compilers and this this procedure is based on the assumption that a compatible target compiler is used.
[140]Flash is mapped into this space primarily for DMA data access.
[141]External memory interface (EMIF).
[142]Unlike the PSoC3 Keil 8051 compiler, all PSoC5 compilers use little-endian format.

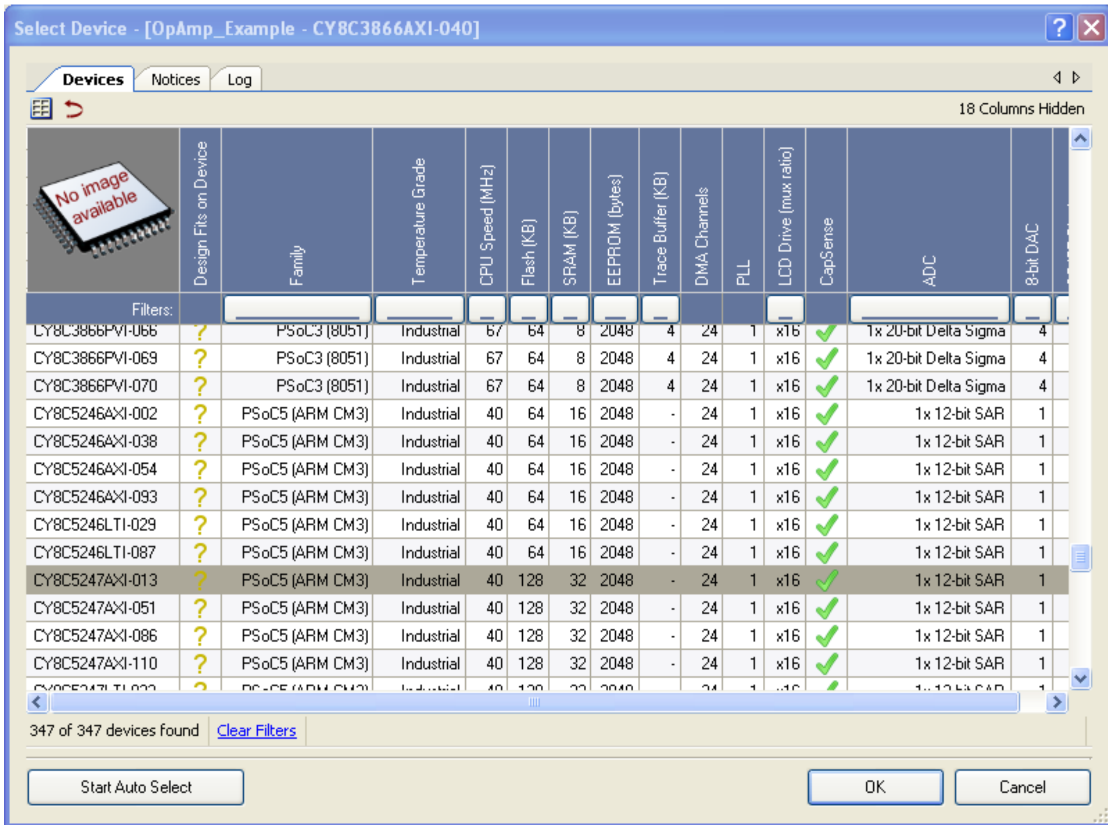


Figure 3.61: Selection of a new PSoC5 target device.

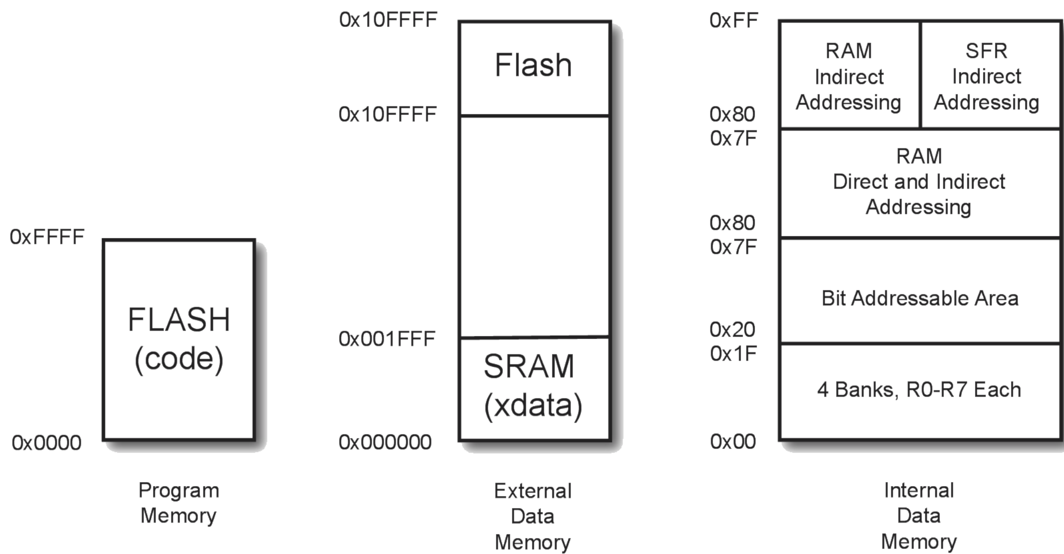


Figure 3.62: The PSoC3 (8051) memory map.

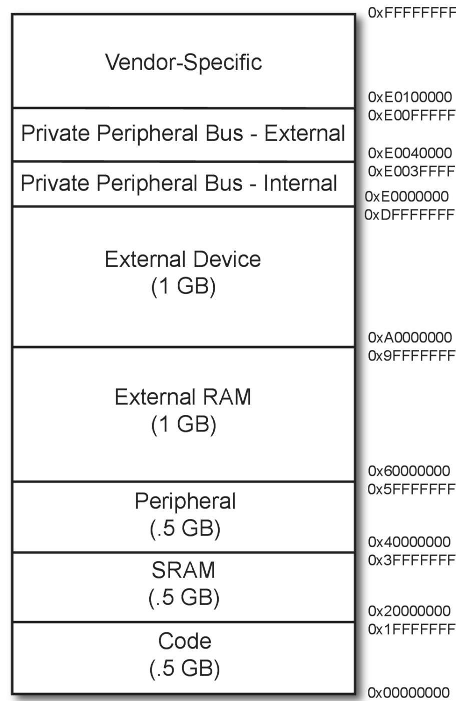


Figure 3.63: The PSoC5 (Cortex-M3) memory map.

```
CY_GET_REG16(addr)
CY_SET_REG16(addr, value)
CY_GET_REG24(addr)
CY_SET_REG24(addr, value)
CY_GET_REG32(addr)
CY_SET_REG32(addr, value)
```

The following macros can be used to access registers mapped above the first 64K bytes of 8051 external data space

```
CY_GET_XTND_REG8(addr)
CY_SET_XTND_REG8(addr, value)
CY_GET_XTND_REG16(addr)
CY_SET_XTND_REG16(addr, value)
CY_GET_XTND_REG24(addr)
CY_SET_XTND_REG24(addr, value)
CY_GET_XTND_REG32(addr)
CY_SET_XTND_REG32(addr, value)
```

and they handle endian format translation correctly and can be ported directly to PSoC5 compilers.

3.22.2 Keil C 8051 Compiler Keywords (Extensions)

Keil has added a number of important extensions to the set of keywords provided by standard C, e.g.,

- *_at_* - variables can be located at absolute memory locations[143] using

```
<< memory_type >> type variable_name_at_constant;
```

  where *memory_type* is the variable's memory type, *type* is the variable type, *variable_name* is the name of the variable and *constant* is the variable's address.

- *alien* - used to invoke PL/M-51 routines from C functions by first declaring them external with the alien function type specifier, e.g.,

```
extern alien char plm_func (int, char);
char c_func (void)  {
int i;
char c;
for (i = 0; i < 100; i++) {
c = plm_func (i, c);         /* call PL/M func */
}
return (c);
}
```

  To create C functions that may be invoked from PL/M-51 routines, the alien function type specifier must be used in the C function declaration. For example:

```
alien char c_func (char a,  int b)  {
    return (a * b);
}
```

  Parameters, and return values of PL/M-51 functions, may be bit, char, unsigned char, int, and unsigned int. Other types, including long, float, and all types of pointers, can be declared in C functions with the *alien* type specifier. However, these types must be used with care because PL/M-51 does not directly support 32-bit binary integers, or floating-point numbers.

  Public variables declared in a PL/M-51 module are available to C programs by declaring them external, like any C variable.

- *bdata* - bit-addressable objects can be addressed as bits or words. Only data objects that occupy the bit-addressable area of the 8051 internal memory fall into this category.

- *bit* - defines a single bit variable[144], e.g.,

```
bit name << = value >>
```

  where *name* is the name of the bit variable and *value* is the value to be assigned to the bit.

  Bit variables are stored in a segment in the 8051 internal memory space. In most cases, bit variables can be defined, and accessed, in the same manner as any other variable type:

```
bit doneFlag = 0;
```

  To port bit variables to PSoC 5 the bit type can be redefined as

```
#define bit uint8
```

  The following is an example of the use of the bit type:

[143]The absolute address following the *_at_* keyword must conform to the physical boundaries of the memory space for the variable. The Cx51 Compiler checks for and reports invalid address specifications.

[144]All bit variables are stored in a bit segment located in the internal memory area of the 8051, which is 16 bytes long. Therefore a maximum of 128 bit variables may be declared within any one scope.

```
  static bit done_flag = 0;    /* bit variable */

  bit testfunc (               /* bit function */
  bit flag1,                 /* bit arguments */
  bit flag2)
{
         .
         .
         .
  return (0);                  /* bit return value */
}
```

- *code* - Program (CODE) memory is read-only. Program memory may reside within the 8051 MCU, be external, or both. Although 8051 architecture supports up to 64K Bytes of program memory, program space can be expanded using code banking. Program code, including all functions and library routines, is stored in program memory. Constant variables may also be stored in program memory. The 8051 executes programs stored in program memory only. Program memory may be accessed from C programs using the code memory type specifier.
- *compact* - A function's arguments and local variables are stored in the default memory space specified by the memory model. It is possible to specify which memory model to use for a single function by including the *small*, *compact*, or *large* function attribute in the function declaration.

  For example:

```
#pragma small         /* Default to small model */

extern int calc (char i, int b) large reentrant;
extern int func (int i, float f) large;
extern void *tcp (char xdata *xp, int ndx) compact;

int mtest (int i, int y)           /* Small model */
  {
     return (i * y + y * i + func(-1, 4.75));
  }
int large_func (int i, int k) large /* Large model */
  {
     return (mtest (i, k) + 2);
  }
```

  The advantage of functions using the SMALL memory model is that the local data and function argument parameters are stored in the internal 8051 RAM. Therefore, data access is very efficient. Because internal memory is limited, the small model may not satisfy the requirements of a very large program, in which case, other memory models must be used. In such cases, a function can be declared that uses a different memory model, as shown above.

  By specifying the function model attribute in the function declaration, it is possible to specify which of the three possible reentrant stacks, and the associated frame pointers, is to be used.[145]

- *data* - this memory specifier always refers to the first 128 bytes of internal data memory.[146]

[145] Note that stack access in the SMALL model is more efficient than in the LARGE model.
[146] Variables stored at that location are accessed using direct addressing.

- *far* - this keyword allows variables and constants to be accessed in external memory using 24-bit addresses. For variables, *far* memory is limited to 16 megabytes. Objects are limited to 64K, and cannot cross a 64K boundary. Constants (ROM variables) are limited to 16 megabytes.
- *idata* - this memory specifier refers to all 256 bytes of internal data memory, but it requires indirect addressing which is slower than direct addressing.
- *interrupt* - interrupts can be used for counting, timing. detecting external events and sending/receiving data via a serial interface.[147]
- *large* - selects the large memory model in which all variables and local data segments procedures/functions are maintained in external memory.
- *pdata* - this memory type is used only for declaring variables and is indirectly accessed by 8-bit addresses of one page of 256-byte page of external data 8051 RAM.
- *_priority_* - this keyword specifies a task's priority, e.g.,

```
void func (void) _task_ num _priority_ pri
```

  where *num* is a task ID number and *pri* is the tasks priority.
- *reentrant* - allows functions to be declared reentrant and therefore called recursively, e.g.,

```
int calc (char i, int b) reentrant  {
     int  x;
     x = table [i];
     return (x * b);
 }
```

  Small, *compact* and *large* model reentrant functions simulate the reentrant stack in *idata*, *pdata* memory and *xdata* memory, respectively. Bit-type function arguments may not be used and local bit scalars are also not available. The reentrant capability does not support bit-addressable variables. Reentrant functions must not be called from *alien* functions and cannot use the alien attribute specifier to enable PL/M-51 argument passing conventions. A reentrant function may simultaneously have other attributes, such as using an interrupt, and may include an explicit memory model attribute (*small*, *compact*, *large*).

  Return addresses are stored in the 8051 hardware stack. Any other required PUSH and POP operations also affect the 8051 hardware stack. Although reentrant functions using different memory models may be intermixed, each reentrant function must be properly prototyped and include its memory model attribute in the prototype. This is necessary for calling routines to place the function arguments in the proper reentrant stack.[148] For example, if *small* and *large* reentrant functions are declared in a module, both *small* and *large* reentrant stacks are created along with two associated stack pointers (one for *small* and one for *large*).
- *sbit* - defines a bit within a special function register (SFR). It is used in one of the following ways:

```
sbit name = sfr-name ^ bit-position;
sbit name = sfr-address ^ bit-position;
sbit name = sbit-address;
```

  where *name* is the name of the SFR bit, *sfr-name* is the name of a previously-defined SFR, *bit-position* is the position of the bit within the SFR, *sfr-address* is the address of an SFR and *sbit-address* is the address of the SFR bit, e.g.,

[147] Thirty two interrupts are located in the jump table from address 0003h - 00FBh, inclusive.
[148] Each of the three possible reentrant models contains its own reentrant stack area and stack pointer.

```
/* define the sbit */
sbit PIN1_6 = SFRPRT1DR^6;
/* access the sbit */
PIN1_6 = 1;
```

The *sbit* keyword is used in PSoC3 for faster access to bits in certain registers, but they cannot be used in PSoC5. Instead the C bit manipulation operators and macros should be used, e.g.,

```
CY_SET_REG8(CYDEV_IO_PRT_PRT1_DR,
CY_GET_REG8(CYDEV_IO_PRT_PRT1_DR) |
0x40;
```

It is often necessary to access individual bits within an SFR and the *sbit* type provides access to bit-addressable SFRs and other bit-addressable objects, e.g.,

```
sbit EA = 0xAF;
```

This declaration defines *EA* as the SFR bit at address *0xAF*, which is the *enable all* bit in the *interrupt enable* register.

Storage of objects accessed using *sbit* is assumed to be little-endian (LSB first). This is the storage format of the *sfr16* type, but it is opposite to the storage of int and long data types. Care must be taken when using *sbit* to access bits within standard data types. Any symbolic name can be used in an *sbit* declaration. The expression to the right of the equal sign specifies an absolute bit address for the symbolic name. There are three variants for specifying the address:

```
sbit name = sfr-name ^ bit-position;
```

The previously declared SFR (*sfr-name*) is the base address for the *sbit* and it must be evenly divisible by 8. The bit-position, which must be a number from $0 - 7$, follows the carat symbol (^) and specifies the bit position to access, e.g.,

```
sfr  PSW = 0xD0;
sfr  IE = 0xA8;
sbit OV = PSW^2;
sbit CY = PSW^7;
sbit EA = IE^7;
```

```
sbit name = sfr-address ^ bit-position;
```

A character constant (*sfr-address*) specifies the base address for the *sbit* and must be evenly divisible by 8. The bit-position (which must be a number from 0-7) follows the carat symbol (^) and specifies the bit position to access, e.g.,

```
sbit OV = 0xD0^2;
sbit CY = 0xD0^7;
sbit EA = 0xA8^7;
```

```
sbit name = sbit-address;
```

A character constant (sbit-address) specifies the address of the *sbit*. It must be a value from 0x80-0xFF, e.g.,

```
sbit OV = 0xD2;
sbit CY = 0xD7;
sbit EA = 0xAF;
```

Only SFRs whose address is evenly divisible by 8 are bit-addressable and the lower nibble of the SFR's address must be 0 or 8. For example, SFRs at 0xA8 and 0xD0 are bit-addressable, whereas SFRs at 0xC7 and 0xEB are not. To calculate an SFR bit address, add the bit position to the SFR byte address, e.g., to access bit 6 in the SFR at 0xC8, the SFR bit address would be 0xCE (0xC8 + 6). Special function bits represent an independent declaration class that may not be interchangeable with other bit declarations, or bit fields. The *sbit* data type declaration may be used to access individual bits of variables declared with the *bdata* memory type specifier. *sbit* variables must be declared outside of the function body.

- *sfr* - defines a special function register (SFR). It is used as follows:

```
sfr name = address;
```

where *name* is the name of the SFR and *address* is the address of the SFR. SFRs are declared in the same fashion as other C variables, except that the type specified is sfr rather than char or int, e.g.,

```
sfr P0 = 0x80;    /* Port-0, address 80h */
sfr P1 = 0x90;    /* Port-1, address 90h */
sfr P2 = 0xA0;    /* Port-2, address 0A0h */
sfr P3 = 0xB0;    /* Port-3, address 0B0h */
```

P0, P1, P2, and *P3* are the SFR name declarations.[149] The address specification after the equal sign must be a numeric constant. *sfr* variables must be declared outside of the function body.

- *sfr16* - defines a 16-bit special function register (SFR) and is implemented as follows:

```
sfr16 name = address;
```

where *name* is the name of the 16-bit SFR and *address* is the address of the 16-bit SFR. The Cx51 Compiler provides the *sfr16* data type to access two 8-bit SFRs as a single 16-bit SFR.

Access to 16-bit SFRs using *sfr16* is possible only when the low byte immediately precedes the high byte (little endian) and when the low byte is written last. The low byte is used as the address in the sfr16 declaration, e.g.,

```
sfr16 T2 = 0xCC;     /* Timer 2: T2L 0CCh, T2H 0CDh */
sfr16 RCAP2 = 0xCA;  /* RCAP2L 0CAh, RCAP2H 0CBh    */
```

In this example, *T2* and *RCAP2* are declared as 16-bit special function registers. The *sfr16* declarations follow the same rules as outlined for *sfr* declarations. Any symbolic name can be used in an *sfr16* declaration. The address specification after the equal sign must be a numeric constant. Expressions with operators are not allowed. The address must be the low byte of the *SFR* low-byte, high-byte pair. When writing to an *srf16*, the code generated by the Keil Cx51 Compiler writes to the high byte first and then the low byte. In many cases, this is not the desired order, and therefore, if the order in which the bytes are written is important, the sfr keyword must be used to define and access the SFRs one byte at a time to assure the order in which the SFRs are accessed. *sfr16* variables may not be declared inside a function, but instead, must be declared outside of the function body.

[149] Names for *sfr* variables are defined just like other C variable declarations and any symbolic name may be used in an *sfr* declaration.

- *small* - A function's arguments and local variables are stored in the default memory space specified by the memory model. However, it is possible to specify which memory model to use for a single function by including the small, compact, or large function attribute in the function declaration, e.g.,

```
#pragma small         /* Default to small model */

extern int calc (char i, int b) large reentrant;
extern int func (int i, float f) large;
extern void *tcp (char xdata *xp, int ndx) compact;
int mtest (int i, int y)            /* Small model */
{
return (i * y + y * i + func(-1, 4.75));
}
int large_func (int i, int k) large /* Large model */
{
return (mtest (i, k) + 2);
}
```

  The advantage of functions using the *SMALL* memory model is that the local data and function argument parameters are stored in the internal 8051 RAM. Therefore, data access is very efficient. Occasionally, because the internal memory is limited, the small model cannot satisfy the requirements of a very large program and other memory models must be used. In that case, a function must be declared that uses a different memory model. By specifying the function model attribute in the function declaration, it becomes possible to select which of the three possible reentrant stacks and frame pointers to use.[150]

- *_task_* - this keyword specifies a function as a real time task when using a real-time multitasking operating system.[151]

- *using* - The first 32 bytes of *DATA* memory (*0x00-0x1F*) are grouped into 4 banks of 8 registers each. Programs access these registers as *R0-R7.* The register bank is selected by two bits of the program status word, *PSW*. Register banks are useful when processing interrupts, or when using a real-time operating system because the MCU can switch to a different register bank for a task, or interrupt, rather than saving all 8 registers on the stack. The MCU can then switch back to the original register bank before returning. The *using* function attribute specifies the register bank a function uses, e.g.,

```
void rb_function (void) using 3
{
.
.
.
}
```

  The argument for the *using* attribute is an integer constant from 0-3. The using attribute is not allowed in function prototypes and expressions with operators are not allowed. The *using* attribute affects the object code of the function as follows:

  – The currently selected register bank is saved on the stack at function entry.

[150] Stack access in the SMALL model is more efficient than in the LARGE model.

[151] Keil's RTX51 Full and RTX51 Tiny kernels support both real-time control and multitasking to provide several operations to be executed simultaneously and carry out operations that must occur within a pre-defined period of time.

– The specified register bank is set.
– The former register bank is restored before the function is exited.

The following example shows how to specify the *using* function attribute and what the generated assembly code for the function entry and exit looks like.

```
stmt level  source
      1
      2         extern bit alarm;
      3         int alarm_count;
      4         extern void alfunc (bit b0);
      5
      6         void falarm (void) using 3  {
      7   1           alarm_count++;
      8   1           alfunc (alarm = 1);
      9   1         }


    ASSEMBLY LISTING OF GENERATED OBJECT CODE

          ; FUNCTION falarm (BEGIN)
    0000 C0D0       PUSH  PSW
    0002 75D018     MOV   PSW,#018H
                          ; SOURCE LINE # 6
                          ; SOURCE LINE # 7
    0005 0500   R   INC   alarm_count+01H
    0007 E500   R   MOV   A,alarm_count+01H
    0009 7002       JNZ   ?C0002
    000B 0500   R   INC   alarm_count
    000D  ?C0002:
                          ; SOURCE LINE # 8
    000D D3         SETB  C
    000E 9200   E   MOV   alarm,C
    0010 9200   E   MOV   ?alfunc?BIT,C
    0012 120000 E   LCALL alfunc
                          ; SOURCE LINE # 9
    0015 D0D0       POP   PSW
    0017 22         RET
          ; FUNCTION falarm (END)
```

In the previous example, the code starting at offset *0000h* saves the initial *PSW* on the stack and sets the new register bank. The code starting at offset *0015h* restores the original register bank by *popping* the original *PSW* from the stack.

The *using* attribute may not be used in functions that return a value in registers. Extreme care should be exercised to ensure that register bank switches are performed only in carefully controlled areas. Failure to do so may yield incorrect function results. Even when the same register bank is used, functions declared with the using attribute cannot return a bit value. The *using* attribute is most useful in implementing interrupt functions. Usually a different register bank is specified for each interrupt priority level. Therefore, one register bank can be employed for all non-interrupt code, a second register bank for the high-level interrupt, and a third register bank for the low-level interrupt.

- *xdata* - External data memory is read/write. Since external data memory is indirectly accessed through a data pointer register (which must be loaded with an address), it is slower than access to internal data memory. *XRAM* space is accessed with the same instructions as the traditional external data space enabled via dedicated chip configuration *SFR* registers and overlaps the external memory space.

  While there may be up to 64K Bytes of external data memory, this address space does not have to be used as memory. A hardware design can map peripheral devices into the memory space so that the program accesses, what appears to be, external data memory to program and control the peripheral.[152] The C51 Compiler offers two memory types that access external data: *xdata* and *pdata*. The *xdata* memory specifier refers to any location in the 64K Byte address space of external data memory. The large memory model locates variables in this memory space. The *pdata* memory type specifier refers to exactly one (1) page (256 bytes) of external data memory. The compact memory model locates variables in this memory space.

3.22.3 DMA Access

DMA transaction descriptors can be programmed to have bytes swapped while transferring data.[153] The swap size can be set to 2 bytes for 16-bit transfers, or 4 bytes for 32-bit transfers. The following examples handle 2- and 4-byte swaps:

```
CyDmaTdSetConfiguration(myTd, 2, myTd, TD_TERMOUT0_EN | TD_SWAP_EN);
```

and,

```
CyDmaTdSetConfiguration(myTd, 4, myTd, TD_TERMOUT0_EN | TD_SWAP_EN |
                        TD_SWAP_SIZE4);
```

respectively.

3.22.3.1 DMA Source and Destination Addresses

PSoC3 and PSoC5 have the same type of DMA controller (DMAC) which stores 32-bit addresses for both source, and destination, in two 16-bit registers. The upper half of the addresses for each DMA channel are specified by the following:

```
DMA_DmaInitalize(..., uppersrcAddr, upperDestAddr)
```

and similarly, the lower half of the addresses are specified for each transaction descriptor (TD) within a DMA channel as:

```
CyDmaTdSetAddress(..., lowerSrcAddr, lowerDestAddr)
```

The contents of a pointer variable cannot be used to provide source or destination address values, because the Keil 8051 compiler uses a 3-byte pointer, i.e., two bytes representing a 16-bit absolute address and a third byte for the memory space being used.

Source in Flash can be accessed by:

```
upperSrcAddr = (CYDEV_FLS_BASE) >> 16
SRAM for source or destination:
upperSrcAddr = 0;
upperDestAddr = 0;
```

[152] This is referred to as memory mapped I/O, in some cases.
[153] DMA byte swapping must be disabled when the code is ported to PSoC5.

and for a peripheral register, for source or destination:

```
upperSrcAddr = 0;
upperDestAddr = 0;
```

The upper half of the PSOC5 address for SRAM or peripheral register for source or destination:

```
upperSrcAddr = HI16(srcArray);
upperDestAddr = HI16(destArray);
```

and the lower half of the address by using the LO16 macro defined in "the *cytypes.h*" file:

```
lowerSrcAddr = LO16(srcArray);
lowerDestAddr = LO16(destArray);
```

Addresses can also be found by using conditional compilation:

```
#f (defined(__C51__))
upperSrcAddr = 0;
#else /* PSoC 5 */
upperSrcAddr = HI16(srcArray);
#endif
```

3.22.4 Time Delays

The *CyDelay* function, defined in *CyLib.c*, is used to generate absolute time delays. It selects the number of loop iterations based on processor type and CPU speed. The supported system function calls include:

- *void CyDelay(uint32 milliseconds)* produces a delay specified by *uint32 milliseconds.*[154] If the clock configuration is changed at run-time, then the function *CyDelayFreq* is used to indicate the new *Bus Clock* frequency. *CyDelay* is used by several components, so changing the clock frequency without updating the frequency setting for the delay can cause those components to fail. *CyDelay* has been implemented with the instruction cache assumed enabled. When the PSoC5 instruction cache is disabled, *CyDelay* will be two times larger.[155]
- *void CyDelayUs(uint16 microseconds)* produces a delay specified by *uint16 microseconds.*
- *void CyDelayFreq(uint32 freq)* sets the *Bus Clock* frequency used to calculate the number of cycles required for implementing the delay specified by *CyDelay.* The frequency used is based on the value determined by PSoC Creator at build time, by default.[156]
- *void CyDelayCycles(uint32 cycles)* results in a delay for the specified number of cycles using a software delay loop.

It should be borne in mind that software delays can be affected by interrupts so care must be exercised in their use. If more accurate delays are required a timer or PWM can be used. A simple assembly language delay can be implemented by loading a value into the accumulator and decrementing it until the value becomes zero. If multiple delays are needed, the value to be decremented for a given delay can be loaded from a LUT.

---

[154]The delay is based on the clock configuration entered in PSoC Creator By default.

[155]*CyDelay* functions implement simple software-based delay loops that are designed to compensate for the bus clock frequency and other factors, e.g., function entry and exit when the delay time is relatively mall.

[156]0: Use the default value, non-0: Set frequency value.

3.23 Reentrant Code

The Keil compiler assumes that functions are not reentrant by default, and therefore, fixed memory locations in RAM are used to store the function's local variables. If the function must be called from different threads (like main and interrupt handler), or recursively, then it must be specifically defined as a reentrant function:

```
/* reentrant function declaration */
void delay (uint32) reentrant;
/* reentrant function definition */
void delay (uint32 x) reentrant
{
        . . .
}
```

PSoC 5 compilers define functions as reentrant and do not support the keyword *reentrant*. To port functions with this keyword to PSoC 5 *reentrant* can be ignored by redefining it as

```
#define reentrant /**/
```

The PSoC 3 Keil compiler provides the various keywords to place variables in different 8051 memory spaces, as shown in Figure 3.64. Keywords such as *code*, *idata*, *bdata* and *xdata,* that

Memory Space	Keyword
Internal RAM	bit, data, idata, bdata
Internal SFRs	sfr, sbit
External Memory	xdata
Code (Flash)	code

Figure 3.64: Keil keywords and related memory spaces.

locate variables in different 8051 memory spaces, can also be ignoring when porting from PSoC3 to PSoC5 by redefining them in a similar manner.

3.24 Code Optimization

Execution speed and code size are often two paramount concerns when designing an embedded system. Historically, designers have often tried to resort to departing from C, and higher level languages when seeking additional optimization and resort to assembly language, particularly in cases which involve microcontrollers with 8051 class microprocessors. Advances in compiler technology have made it possible to write highly efficient C code in terms of memory requirements and speed. The Keil compiler has a number of Keil-specific keywords that have been added to support optimization, so that assembly language code may be obviated. However, these keywords are not necessarily supported by compilers for other processors such as the Cortex-M3 in PSoC5.[157] The Keil Compiler supports several levels of optimization with Level 2 being the default level in PSoC Creator. Level 3 optimizes the compiled code with respect to code size by deleting redundant MOV operations, which in some cases have a significant impact on both code size and speed.

[157] PSoC Creator supports a number of equivalent macros to facilitate porting code from PSoC3 to PSoC5

The 8051 core is a 256-byte address space that contains 256 bytes of SRAM plus a large set special function registers (SFRs), as shown in Figure 3.65, and the 8051 is most efficient when it utilizes this memory. As shown in the figure, the lower 128 bytes is SRAM, and accessible both directly and indirectly. The upper 128 bytes contains another 128 bytes of SRAM that can only be accessed indirectly. The same upper address space also contains a set of SFRs that can only be accessed directly. Table 3.4 details bytes in the lower address space that can be accessed in other modes. The memory map for the first 256 bytes in the 8051's memory space is shown in Figure 3.66.

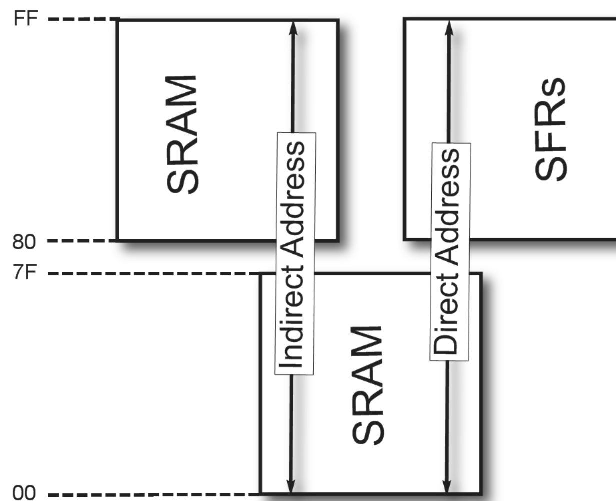


Figure 3.65: 8051 internal space layout.

Table 3.4: 8051 keyword memory space.

Memory Space	Keyword
Internal RAM	bit, data, idata, bdata
Internal SFRs	sfr, sbit
External Memory	xdata
Code (Flash)	code

3.24.1 Techniques for Optimizing 8051 Code[158]

Whenever feasible, it is advisable [1] to use bit variables for any variables that will have only binary values, i.e., 0 and not 0, and define then as being of type *bit*, i.e.

```
bit myvar;
```

[158] In this section frequent reference will be made to assembly code. However, the reader is asked to recall that any such code discussed herein is presumed, unless stated otherwise, to have been the output of PSoC's C compiler and, as such, is not hand-coded, assembly language, source code.

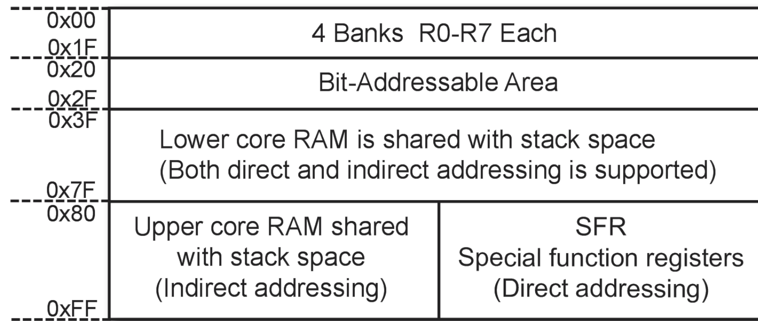


Figure 3.66: 8051 internal memory allocation

The use of bit variable allows the compiler to draw upon the complete set of 8051 bit-level assembler instructions to create very fast and compact code, .e.g,

```
myvar = ~myVar;
if (!myVar)
{
        ...
}
```

that causes the compiler to produce the following two lines of assembly code:

```
B200      CPL    myVar
200006    JB     myVar,?C0002
```

which requires 5 bytes of Flash and 8 cpu cycles.

Whenever possible, calling functions from interrupt handlers written in C should be avoided. The Keil C compiler pushes any register contents that it assumes may be changed by the ISR, which can result in a substantial amount of additional code as illustrated by the following example:

```
CY_ISR(myISR)
{
UART_1_ReadRxStatus():
}
```

that is a simple ISR, that when compiled can result in the following assembly language code:

```
   C0F0  PUSH  B
   C083  PUSH  DPH
   C082  PUSH  DPL
   C085  PUSH  DPH1
   C084  PUSH  DPL1
   C086  PUSH  DPS
 758600  MOV DPS,#00H
   C000  PUSH  ?C?XPAGE1SFR
 750000  MOV  ?C?XPAGE1SFR,#?C?XPAGE1RST
   C0D0  PUSH  PSW
 75D000  MOV PSW,#00H
   C000  PUSH  AR0
   C001  PUSH  AR1
   C002  PUSH  AR2
```

```
  C003 PUSH AR3
  C004 PUSH AR4
  C005 PUSH AR5
  C006 PUSH AR6
  C007 PUSH AR7
120000 LCALL UART_1_ReadRxStatus
  D007 POP AR7
  D006 POP AR6
  D005 POP AR5
  D004 POP AR4
  D003 POP AR3
  D002 POP AR2
  D001 POP AR1
  D000 POP AR0
  D0D0 POP PSW
  D000 POP ?C?XPAGE1SFR
  D086 POP DPS
  D084 POP DPL1
  D085 POP DPH1
  D082 POP DPL
  D083 POP DPH
  D0F0 POP B
  D0E0 POP ACC
  32   RETI
```

A better approach would be to use a flag in the ISR in the form of a global variable. The flag is simply a single bit that is read by background code accessing the register that contains the flag bit. The following is an example using a flag in the form of a global variable of type *bit* which is subsequently read by background code:

```
CYBIT flag;
CY_ISR(myISR)
{
   flag =1;
}

void main()
{
   if (flag)
       {
         flag = 0;
        UART_1+_ReadRxStatus();
                       ...
```

The ISR portion of this code results in the following assembly code:

```
D200 SETB flag
32   RETI
```

which is less than 10% of the assembly code produced by the previous example. However, it should be noted that using a flag in this manner, assumes that the status register containing the flag will be checked often enough to result in the desired operation.

Placing variables in the 8051's internal memory can produce substantial benefits. The location of variables in memory should be based on relative frequency of access, e.g., the most frequently accessed should be of type *data*, the next most frequently accessed as type *idata*, and so on, for *pdata* and *xdata.* As noted previously, because stack space is limited, the Keil compiler stores local variables in fixed memory locations and shares these locations among local variables in functions that don't call each other. Therefore, when possible, variables within functions should be local variables which allows the Keil compiler to store such variables in registers R0-R7. Loop decrementing is more efficient because it is easier to test for zero than for a non-zero value, as shown by the following examples:

```
void main()
{
   data uint8 i;
/* loop 10 times */
for (i = 10; i != 0; i--)
{
          ...
}
```

is compiled as

```
75000A     MOV i,#0AH ; i = 10
           ?C0002:
E500       MOV A,i ; i != 0
6006       JZ ?C0003
                    ...
1500       DEC i ; i--
80EF       SJMP ?C0002
           ?C0003:
```

as opposed to

```
void main()
{
   data uint8 i;
   /* loop 10 times */
   for (i = 0; i < 10; i++)
   {
         ...
   }
```

that compiles as

```
E4       CLR A ; i = 0
F500     MOV i,A
         ?C0002:
E500     MOV A,i ; i < 10
C3       CLR C
940A     SUBB A, #0AH
5006     JNC ?C0003
                  ...
0500     INC i ; i++
80EF     SJMP ?C0002
         ?C0003:
```

Bit variables can be used to dramatically improve efficiency, and bit-level assembler instructions can also be employed to implement bit-wise C operations. Some examples of setting bit variables are given by the following:

```
uint8 x;
x |= 0x10; /* set bit 4 */
x &= ~0x10; /* clear bit 4 */
x ^= 0x10; /* toggle bit 4 */
if (x & 0x10) /* test bit 4 */
{
        ...
}
```

8051 bit-level assembly instruction can be used to implement C bitwise operations by using the keyword *sbit* and the ∧ operator.[159]

One method is given by:

```
/*myVar is located in idata at 202F */
        bdata uint8 myVar;
/* this is bit 4 of myVar */
        sbit mybit4 = myVar^4;
/* set bit 4*/
        mybit4 = 1;
/* clear bit 4 */
        mybit4 = 0;
/* toggle bit 4
        mybit4 = ~mybit4;
/* test bit 4 */
        if (mybit4)
{
        ...
}
```

which can also be used for variables that are larger than 8-bits, e.g., uint16, uint32, etc. It should be noted that *sbit* and *bdata* definitions global and not local within a function. PSoC Creator provides support for *sbit* and *sfr* keywords as follows:

```
sfr PSW = 0xD0;
sbit P = PSW^0;
sbit F1 = PSW^1;
sbit OV = PSW^2;
sbit RS0 = PSW^3;
sbit RS1 = PSW^4;
sbit F0 = PSW^5;
sbit AC = PSW^6;
sbit CY = PSW^7;
```

Alternatively, a bit-addressable SFR can be used given that SFR PSW contains the program status word at D0 and is therefore directly accessible. The *sbit* keyword can be used to access each of the PSW's bits using the same technique discussed in this section, e.g.,

```
F0 = ~F0;
```

[159] In this discussion the ∧ operator is not the standard C language exclusive or (XOR).

(PSW's F0 and F1 bits are available for general purpose use.) The accumulator (ACC) and the B register can be used as temporary SFRs. However, the individual bits of each must be specifically defined, e.g.,

```
/* bit 4 of ACC SFR */
sbit A4 = ACC^4;
/* bit 3 of B SFR */
sbit B3 = B^3;
```

in which case, faster bit testing can be achieved by using

```
/* assume return value is 8 bits */
    ACC = UART_1_ReadRxStatus();
    if (A4) /* test bit 4 */
    {
            ...
    }
```

The auxiliary B resister can be used for storage to facilitate instructions, such as MUL and DIV, or to switch two 8-bit variables

```
uint8 x, y;
B = x;
x = y;
y = B;
```

Pointers are commonly used in embedded systems and their size is a function of the address space being employed, e.g., a 64K address space will require two byte pointers, while larger spaces such as those addressed by PSoC5 require 4-byte pointers to span the address space. However, PSoC3's 8051 employs several memory spaces ranging from 256-64K bytes and therefore the Keil C compiler utilizes memory-specific and generic pointers.[160] The use of memory-specific pointers is more efficient than generic pointers and therefore the latter should be used only when the memory type is unknown.[161] A 8051 generic pointer can used to access data regardless of the memory in which it is stored. It uses 3 bytes - the first is the memory type, the second is the high-order byte of the address, and the third is the low-order byte of the address. A memory-specific pointer uses only one or two bytes depending on the specified memory type.

The C keyword *const*, which be added to an array declaration or variable, is used to require that the variable not be changed but does not control where the variable is stored, e.g.,

```
const char testvar = 37;
void main()
{
char testvar2 = testvar;
```

is compiled as

```
900000  MOV DPTR,#testvar
E0      MOVX A,@DPTR ; MOVX accesses xdata space
900000  MOV DPTR,#testvar2
F0      MOVX @DPTR,A
```

[160] A generic pointer can be used to access data regardless of the memory in which it is stored. It uses 3 bytes - the first is the memory type, the second is the high-order byte of the address, and the third is the low-order byte of the address. A memory-specific pointer uses only one or two bytes depending on the specified memory type.

[161] The majority of Keil library functions take generic pointers as arguments and memory-specific pointers are automatically cast to generic pointers.

and shows that the const variable *testvar* is stored in Flash, and copied to an SRAM location initialized in the startup code. If there is insufficient SRAM to store all of the const variables, the keyword *code* (or CYCODE) must be used in the declaration, i.e.,

```
code const char testvar = 37;
void main()
{
char testvar2 = testvar;
```

and then the corresponding assembly code is given by

```
900000 MOV DPTR,#testvar
E4     CLR A
93     MOVC A,@A+DPTR ; MOVC accesses code space
900000 MOV DPTR,#testvar2
F0     MOVX @DPTR,A
```

so that the const variable *testvar* is stored in Flash.

Arrays and strings can be kept in FLASH as illustrated by the following example:

```
const float code array[512] = { ... };
code const char hello[] = "Hello World";
```

The arguments for C functions are typically passed on the CPU's hardware stack. However, the Keil compiler uses either registers *R0-R7,* or fixed memory locations for passing such arguments and does not pass arguments via the stack. The use of registers is employed because it is faster and uses fewer code bytes. The latter can be important because of the limitation of the 8051 hardware stack to 256 bytes. However, this method has some limitations, as shown in Table 3.5. If other types of arguments are involved, they can be passed in fixed memory locations. To the

Table 3.5: Argument passing via registers.

Argument Number	Char, 1 byte Pointer	Int, 2-byte Pointer	Long, Float	Generic Pointer
1	R7	R7, R6 (MSB)	R7-R4 (MSB)	R3 (mem type) R2 (MSB) R1
2	R5	R5, R4 (MSB)	R7-R4 (MSB)	R3 (mem type) R2 (MSB) R1
3	R3	R3, R2 (MSB)	-	R3 (mem type) R2 (MSB) R1

extent possible no more than three function arguments should be employed. However, there is no guarantee that the compiler will pass three arguments in registers.

Arguments of type bit are always passed in a fixed memory location in the 8051's bit space (internal memory) and cannot be passed in a register. Bit variables should be declared at the end of a function's argument list, to keep the other arguments consistent with Table 3.5. Function return values are handled as described in Table 3.6. Return values of type bit are always passed via registers. If a function argument is the return value of another function that argument should be the first in the argument list whenever possible.

Table 3.6: Function return values via registers.

Return Type	Register
Bit	Carry Flag
char, 1-byte pointer	R7
int, 2 byte pointer R7, R6 (MSB)	R7, R6 (MSB)
long, float	R7-R4 (MSB)
Generic Pointer	R3 (mem type), R2 (mem type), R1

3.25 Real Time Operating Systems

In a typical embedded system, there are often multiple tasks[162] involved with a requirement to share and exchange data between such tasks. Scheduling of tasks[163], and sharing of resources in these cases, can sometimes be greatly facilitated by introducing a real time operating system[164] so that tasks are processed subject to specific, predefined time constraints. This type of operating system is referred to as a real time operating system, or RTOS[165]. The majority of popular microcontrollers lack the memory space, execution speed, and/or other resources, to adequately support an RTOS. However, the ARM architecture of PSoC5, its clock speed (max of 67 mega Hertz), RAM space (4 gigabytes) and other resources are sufficient to support a real time operating system (RTOS), such as FreeRTOS.[166]

The role of the real time operating system is to provide an environment capable of managing the available resources, and provide a variety of services for tasks, e.g.,

- management of system resources and CPU,
- assuring that tasks are handled in a predefined manner and within the imposed time constraints,
- handling data movement and communications between tasks,
- efficiently managing RAM allocation and use,
- determining which resources can be shared and which are allocated exclusively,
- responding to events,
- assigning priorities to tasks,
- coordinating internal and external events,
- synchronizing tasks,

and,

- handling compute and I/O bound tasks.

[162]Some tasks may have to handled in parallel, others in serial fashion and these activities are referred to collectively as multi-tasking.

[163]Tasks are also referred to as as processes and in the present context, the two terms are considered equivalent.

[164]A real time operating system is a type of operating system that provides one, or more, responses within predefined time periods.

[165]Real time operating systems are also referred to as real time executives and kernels.

[166]*OpenRTOS*^{TM} is a commercially licensed and supported version of FreeRTOS that includes fully featured professional grade USB, file system and TCP/IP components. OpenRTOS is a commercial version of FreeRTOS and provided under license.

3.25.1 Tasks, Processes, Multi-threading and Concurrency

Tasks can be in various states, e.g., running, ready (pended or suspended), blocked (delayed, dormant or waiting). If the embedded system is to employ multitasking, in an optimized fashion, compute-bound and I/O-bound tasks must be assigned priorities such that the executive has a basis for assigning the ordering of execution of tasks. This approach allows lower priority tasks to be preempted by higher order tasks, by the scheduler. In the case of round-robin scheduling of tasks, tasks of the same priority are executed in a predefined order. Preemptive scheduling assigns the order of task execution based on the concept that the highest priority process, in a group of waiting tasks, is executed first, i.e., it *preempts* other tasks.

Each task is assigned the necessary resources, e.g., RAM space, a task stack, program counter, I/O ports, file descriptions, registers, etc. These resources may, or may not, be shared with other tasks. The state of a process, at any given time, is determined by the then current program counter value, data values in the task's allocated memory space(s) and/or registers. CPU time is allocated to each task by the operating system, and if tasks are to effectively/efficiently run simultaneously, the CPU must switch from one task to another[167], often whether a given task is completed or not, and then return at a later time to the incomplete tasks until each process has been completed.[168] If the CPU switches tasks fast enough, the tasks are said to be running *concurrently*, or alternatively, as *concurrent processes*. In some operating systems the CPU switches task execution at fixed intervals, a practice referred to as *time slicing*. Tasks waiting their turn to be executed are said to be in a *waiting state*. A task can be terminated either upon completion, or as a result of being *killed*[169]. Typically, a terminated process, whether completed or killed, is removed from memory and the associated resources are deallocated.

A *thread* is a set of instructions that has access to stack space and registers, and the associated *resources*, needed to carry out a task (process). Tasks can be *ready*, *blocked, running* or *terminated*. Multiple threads are used when tasks need to occur contemporaneously and are referred to as *parallel processes*. A scheduler is used to control which task is to be run, and when it is to run. A *dispatcher* starts each task, initiates intertask communications, or any interprocess communication required to exchange information between tasks. Multiple threads may be running in a single- or multi-processor environment.[170]

In the single-processor case, the processor is switched from one thread to another in a mode known as multi-threading based on *time-division multiplexing*. If multiple processors are involved, each may be running a single thread. *Multi-threading* refers to the existence of multiple threads within a given process that, although executing independently, share the resources allocated to the process.

In a multi-thread environment semaphores[171] are sometimes employed to avoid collisions when data is being modified It should be noted however, that threads *are not* synonymous with processes, or tasks, e.g.,

- Context switching from one thread to another, within a given process, is generally substan-

[167] This is typically referred to as *context switching*.

[168] It is assumed in this discussion that the CPU is operating at a sufficiently fast clock rate to able to switch among tasks while assuring that the overall system response meets the system's performance criteria. Some tasks may never be completed, while others have various lifetimes.

[169] The *killing* of a task typically involves sending a signal (message) to a process to terminate.

[170] "Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism." [36]

[171] The simplest from of semaphore is a Boolean variable or integer that signals whether or not access to a critical section of code, or a critical variable, is available.

tially faster than process context switching.

- Threads share address space, while processes have independent address spaces.
- Processes rely solely on inter-process communications to exchange data and information.
- Processes are typically independent tasks that may or may share data and./or resources.

While concurrent processing offers a number of attractive benefits that are not available from sequential code execution, it is not a panacea. As noted by Sutter and Larus [], developing concurrent systems is not an easy task, even though, as observed by Lee [36], as suggested by the world is "highly concurrent" and humans are rather adept at analyzing concurrent systems.

3.25.2 Task Scheduling and Dispatching

The RTOS contains both a scheduler and dispatcher within the RTOS' *kernel.* The operating system is responsible for management of memory, I/O, tasks, file system the file system, networking and interpretation of commands. Task control blocks (TCBs), either static or dynamic[172], are used to encapsulate the important information associated with a given task, e.g.,

- associated CPU registers
- contents of the program counter
- state of a process and an associated ID
- list of open files
- a pointer to a function

A typical RTOS employs a set classes that support kernel services invocable by the applications tasks and include support for

- *Intertask communications* - passing of information between tasks is accomplished by classes such as event flags, mailboxes[173], messages, queues[174], pipes, timers, mutexes[175] and semaphores.[176]
- *Tasks* manage program execution. While each task is independent of other tasks, tasks can interoperate via data structures, I/O and other constructs. Inter-task communications employs semaphores, messages queues, pipes, shared memory signals, mail slots and sockets.
- *Kernel service routines* process *kernel service requests* initiated by an application to provide operating system functions needed by the application.
- *Interrupts* are an important aspect of a RTOS particularly with respect to prioritization of tasks. However, prioritization is not sufficient to assure that tasks are handled in timely fashion.

Scheduling can be either clock- or priority-driven. Scheduling variables such as arrival time, computation, deadline, finish time, lateness, period and start time are used to guarantee responsiveness and minimize latency.

- *Arrival time* is defined as the point in time when a task is ready to run.

[172]Static TCB allocation implies that TCBs are created and remain whereas dynamic TCBs are typically deleted once a task has been completed, or terminated.

[173]Messages and mailboxes are employed to transmit data between a sender and a receiver.

[174]Queues are used to pass data between a producer and a consumer.

[175]Mutexes are binary flags that assure that mutually shared code is also mutually exclusive. Thus multiple tasks can use a resource but only one at a time.

[176]Semaphores are constructs used to synchronize tasks and events. The concept of semaphores was introduced by Edsger Dijkstra a Dutch computer scientist who in addition, contributed to the deprecation of the *GOTO* statement, created reverse polish notation (RPN) and a multitasking operating system known as "THE".

- *Computation time* is defined as the processor time required to complete execution of a task, in the absence of interruption.
- *Deadline* is defined as the latest time at which a task is completed.
- *Lateness* is defined as the length of time after the deadline has been passed required to complete a task.
- *Period* is defined as the minimum time that elapses between release of the CPU.[177]
- *Start time* is defined as the time at which the task begins execution.

Each task has a deadline, execution time and period associated with it, as shown in Figure 3.67. In most cases, the deadline and the period are quantitatively equal. However, a task can start at

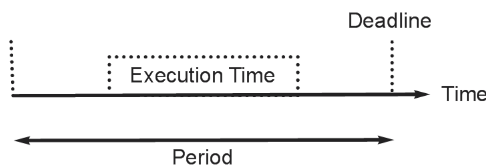


Figure 3.67: Task timing parameters.

any time within the period.

3.25.3 PSoC Compatible Real Time Operating Systems

There are a number of RTOS sources available as commercial or freeware implementations that support either PSoC3 or PSoC5. A brief description of some of these is provided in this section. In some cases the source code for the RTOS is also available, as noted.

Micriμm[178] offers a commercial version of μC/OS III for PSoC5. It has the following features/benefits:

- Relatively small footprint.[179]
- Is written in ANSI C.
- Supports a variety of user-selectable features.
- Employs *round-robin scheduling.*
- Protects critical regions by disabling interrupts while minimizing overhead and providing deterministic interrupt response.
- Supports an arbitrary, although user-selectable, number of priorities.[180]
- Is designed specifically for embedded system applications.
- Blocks NULL pointers Ensures that arguments are within allowable ranges.
- Supports user-allocation of kernel objects at run-time.
- Execution times are not a function of the number of executing tasks.
- Places no constraints on maximum task size.[181]
- Allows multiple tasks to run, at the same priority level, in a user-specified, time-slice mode.

[177]In some RTOS environments, the tasks with the shortest periods are given the highest priority.
[178]http://micrium.com. The source code is available.
[179]The footprint size is determined in part by the user-selectable features chosen.
[180]Typical embedded systems use from 32-256 levels, inclusive, of priority.
[181]Minimum task sizes are imposed.

- Places no limitations on the number of tasks, semaphores, mutexes, event flags, message queues, timer or memory partitions.
- Supports monitoring of stack growth of tasks

FreeRTOS[182] (PSoC5) is available as freeware,[183] and has the following features/functionality:

- Minimal ROM, RAM and processing overhead.
- Small footprint.[184]
- Is relatively simple.[185]
- Very scalable,
- Offers a smaller/easier, real time processing alternative for applications for which eCOS, embedded Linux (or Real Time Linux) and uCLinux are too large, not appropriate, or not available.

RTX51 Tiny[186] (PSoC5) is Keil's real-time operating system that provides an RTOS environment for programs based on standard C constructs and compiled with the Keil C51 C Compiler. Keil additions to the C language allow task functions to be declared without the need for complex stack and variable frame configuration.

RTX51 (PSoC3) provides the following features:

- Code banking Explicit task switching
- Task ready flag
- Support of CPU idle mode
- User code support in timer mode
- Interval adjustment support
- Scalability

The footprint can be minimized by disabling round-robin switching[187], stack checking and avoiding unnecessary use of system functions.

The supported functions include:

- *isr_send_signal* causes a signal to be sent to the task's *task_id.* If the task is already waiting for a signal, it is prepared for execution without starting it. Otherwise, the signal is stored in the task's signal flag.
- *isr_set_ready* places the task specified by task_id into the ready state. This function can only called only from interrupt functions. The isr_send_signal function returns a value of 0 if successful and -1 if the specified task does not exist.
- *os_clear_signal* clears the signal flag of the task specified by the *task_id.*
- *os_create_task* causes a task to be marked as ready and executed at the next available opportunity.
- *os_delete_task* stops the task identified by the *task_id* and removes it from the task list.
- *os_reset_interval* is used to correct timer problems.

[182]http://www.freertos.org. The source code is available.
[183]Documentation is available for a nominal sum.
[184]Typical kernel binary image size ranges from 4-9 Kbytes.
[185]The kernel core is contained in three C language files.
[186]http://www.keil.com
[187]Reducing round-robin task switching also reduces the data space requirements.

- *os_running_task_id* determines the *task_id* for the task that is currently running.
- *os_wait* halts the current task and waits for an event such as a time interval, a time-out, or a signal from another task or interrupt.
- *os_switch_task* allows a task to halt execution and allow another task to run. If calling task is the only task ready for execution it resumes running immediately.

When using PSoC5 with a real time operating system, it should be noted that the Cortex M3 core uses numerically low priority numbers to represent HIGH priority interrupts. When assigning an interrupt a low priority it must not be assigned a priority of 0 (or other low numeric value) because it can result in the interrupt actually having the highest priority in the system and could result in a system crash if this priority is above *configMAX_SYSCALL_INTERRUPT_PRIORITY*. The lowest priority on a Cortex M3 core is 255.[188]

If a PSoC5 application provides its own implementation of an interrupt service routine, which accesses the Kernel API, the priority must be equal to, or numerically greater than, the *configMAX_SYSCALL_INTERRUPT_PRIORITY* so in effect it has a lower priority. To install a customize interrupt service routine, call the *Peripheral_StartEx(vCustomISR)* function (where 'Peripheral' is the name of the peripheral to which the ISR relates) passing the interrupt service routine function which has its prototype declared as *C__ISR_PROTO(vCustomISR)* and the function declared with *CY_ISR(vCustomISR).*

In the function *vInitialiseTimerForIntQTests()* in *IntQueueTimer.c*, the ISR is installed using a call to *isr_ High_ Frequency_ 2001Hz_ StartEx()*. Each port # defines '*portBASE_TYPE*' to equal the most efficient data type for that processor. This port defines *portBASE_ TYPE* to be of type long.

3.26 Additional Reference Materials

There a number of valuable resources available, via www.cypress.com, that include training documents/videos, device datasheets, a technical reference manual (TRM), component data sheets, system reference guides, component author guide (CAG), application notes, example projects knowledge base forums and various forums, to assist the designer with the development of PSoC3/PSoC5 embedded systems.

The PSoC3/PSoC5 device datasheets provide a summary of the features, device-level specifications, pin-outs and fixed functional peripheral electrical specifications. The technical reference manual describes the functionality of all of the peripherals in detail and includes the associated register descriptions. The component datasheets contain the information required to select and use a component and its functional description, API documentation, assembly language and C example source code, and the relevant electrical characteristics of the component.

The system reference guide (SRG) describes the PSoC Creator *cy_boot* component. This component is automatically included in every project by PSoC Creator[189] and includes an API that can be accessed by firmware for tasks associated with

- Clocking - PSoC3/5 have flexible clocking capabilities that are controlled in PSoC Creator by selections within the Design-Wide Resources (DWR) settings, connectivity of clocking signals on the design schematic, and API calls that can modify the clocking at runtime.

[188]Different Cortex M3 vendors implement a different number of priority bits and supply library functions that expect priorities to be specified in different ways.

[189]Only a single instance can be included in a project, does not include symbolic representation and is not included in the component catalog.

- DMA - The DMAC files provide the API functions for the DMA controller, DMA channels and Transfer Descriptors. This API is the library version not the auto generated code that is generated when the user places a DMA component on the schematic. The auto generated code would use the APIs in this module.
- Flash Linker scripts[190]
- Power management[191]
- Startup code - the *cy_ boot* functionality includes a reset vector, setting up the processor to begin execution, setup of interrupts/stacks, configuration of the target device, preservation of the reset status and calling the main() C entry point.[192]

and,

- Various library functions:
  1. unit8 CyEnterCriticalSection(void) - disables interrupts and returns a value indicating whether interrupts were previously enabled (the actual value depends on whether the device is PSoC 3 or PSoC 5).
  2. unit8 CyExitCriticalSection(void) - re-enables interrupts if they were enabled before CyEnterCriticalSection was called. The argument should be the value returned from CyEnterCriticalSection.
  3. void CYASSERT(uint32 expr) - macro evaluation of an expression and if it is false, i.e., evaluates to 0, then the processor is halted. This macro is evaluated unless NDEBUG is defined, if not, then the code for this macro is not generated. NDEBUG is defined by default for a Release build setting and not defined for a Debug build setting.
  4. void CySoftwareReset(void) - forces a software reset of the device during which the startup code will detect that the reset was the result of a software reset and the SRAM memory area, indicated by corresponding arguments will not be cleared. If any of this area has initialization assignments that initialization will still occur.
  5. void CyDelay(uint32 milliseconds) - invokes a delay[193] by the specified number of milliseconds. By default the number of cycles to delay is based on the clock configuration. If the clock configuration is changed at run-time, then the function *CyDelayFreq* is used to indicate the new Bus Clock frequency. CyDelay is used by several components, so changing the clock frequency without updating the frequency setting for the delay can cause those components to fail.
  6. void CyDelayUs(uint16 microseconds) - the number of cycles to delay is, by default, based on the clock configuration. If the clock configuration is changed at run-time, then the function *CyDelayFreq* is used to indicate the new Bus Clock frequency. *CyDelayUs* is used by several components, therefore changing the clock frequency without updating the frequency setting for the delay can cause those components to fail.
  7. void CyDelayFreq(uint32 freq) - sets the Bus Clock frequency used to calculate the number of cycles needed to implement a delay with *CyDelay.*The frequency used is based, by default, on the value determined by PSoC Creator at build time.
  8. void CyDelayCycles(uint32 cycles) - the delay, determined by the specified number of cycles, is created by a software delay loop.

[190]cf. System Reference Guide, *cyboot* Component Document

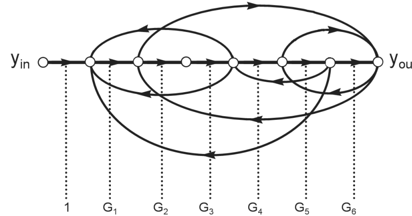
[191]ibid.

[192]Initialization of static/global variables and the clearing of all remaining static/global variables is also handled by *cy_ boot*.

[193]The *CyDelay* functions implement simple software-based delay loops that are designed to compensate for bus clock frequency and other factors. Additional factors may also influence the actual time spent in the loop, e.g., function entry and exit, and other overhead factors, may also affect the total amount of time spent executing the function. This may be especially apparent when the nominal delay time is small.

3.27 Exercises

1. Using Mason's rule, find the gain for signal flow graph shown below.



2. Explain the distinctions between and benefits of multithreading, multitasking and sequential tasking. Describe a physical system that employs all three. Why are threads said to be nondeterministic?

3. Draw a block diagram of embedded system that controls a traffic signal, pedestrian signals and activation buttons at a four-way intersection. Draw the signal graph(s) for such a system and discuss how the design would have to be modified to allow emergency vehicles the right of way.

4. Write two callable PSoC3 routines, one in C and one in assembly, that will produce a delay that can be altered programmatically to provide variable length delays. Comment on the relative speed and overhead requirements of each.

5. Sketch an example of frequency modulation of the sensor signal shown in Figure 3.14.

6. Sketch Figure 3.10 in the form of a signal graph.

7. Create a state diagram for a clock that displays minutes hours and seconds.

8. Give an example of how to use the RTX51 function calls to facilitate a C program designed to control a traffic light. Assume that the traffic control system is capable of handling emergency traffic such as fire trucks, ambulances, police vehicles, etc., on a priority basis based on the type of emergency vehicle.

9. Describe the design for a temperature measuring system that utilizes a temperature dependent resistance, such as a thermistor, whose resistance is a function solely of temperature that can be stored in a look-up table. Provide a requirements description, specification, signal flow graph, and block diagram for the design. How can such a system be implemented if the function itself has other dependencies as well, e.g., ambient pressure?

10. Assuming that the design developed in response to Exercise 9 were created for a PSoC3 and in C. What changes, if any, would be necessary in order to base the design on a PSoC5, i.e., what changes would be required to port it to a PSoC5?

Chapter 4

Communication Peripherals

Embedded systems are often required to communicate with other systems and present data on visual display devices such as LED discrete character displays and LCD screens. Whether communicating with display devices, or other local/remote systems, a wide variety of communications protocols are in common use, e.g., I2C, UART, SPI, USB, RS232, RS485, etc. Many of these protocols have certain features in common and other features which are unique to a particular protocol. Both PSoC3 and PSoC5 are capable of supporting a wide variety of such protocols.[1]

4.1 Communications Protocols

One might well ask why there is a need for so many different communication protocols,[2] particularly as they relate to microprocessors and microcomputers. The simple answer is that a typical embedded system communicates with a number of different devices each of which can have its own preferred communications interface. Data transmission from one device, and/or one location to another, typically relies upon a preferred speed of transmission, support for buffering of data[3], retention of data integrity and, if possible, error correction.

As a result many of the extant protocols have either evolved over time into various incarnations, or been replaced by newer protocols, in an attempt to address complexity concerns, speed, data transfer rates, cost, noise immunity, operating levels, interoperability challenges, networking considerations, transfer distance/times, data security/integrity and a myriad of other issues. In some applications multiple protocols are employed in the same application. In other situations, older protocols are still employed to address interfacing requirements imposed by legacy hardware ad software systems. Some protocols address peer-to-peer transmission, others are applicable to master-slave configurations and still others, various networking configurations.

Error detection schemes are often based on transmission of additional data[4] with each data block that makes it possible to determine whether or not data integrity has been maintained. When a data block, or frame, of data is received, the redundant data is used to determined

[1]In some applications, the support for a particular protocol is part of the PSoC3/5 architecture, in other cases external hardware may be required to interface PSoC3/5 with external communications channels.

[2]A communications protocol is a formal statement of of the governing rules and formats for digital communications between two or more devices. In addition to setting forth the data formats and syntax involved, the protocol typically define the parameters used for authenticating a received message and in some cases define the error detection, and correction, algorithms to be used.,

[3]Buffering of data is used as a method of holding data until the communication channel is available for transmission.

[4]Referred to as redundancy data.

©Copyright 2012 Cypress Semiconductor

the data has been changed during transmission. In some applications, when an error is detected, algorithms are applied to the data correct such errors. Often a trade off has to be made regarding simply retransmitting data from one location to another and the time required to apply an error correction algorithm.

4.2 I2C

The Inter-integrated Circuit Bus[5] (*I2C*) was originally developed by Phillips Semiconductor to support multi-master intercommunications between devices, such as integrated circuits on a printed circuit board. This makes the I2C component ideal when networking multiple devices whether all on a single board, or as part of a small system. Such systems can employ a single master and multiple slaves, multiple masters, or an arbitrary combination of masters and slaves. Such implementations can employ either fixed hardware I2C blocks, or universal digital blocks (UDBs).

I2C utilizes a two-wire, serial, bidirectional bus connected in a master-slave configuration, as shown in Figure 4.1.[6] Although originally limited to a maximum transfer rate of 100 Kbits/sec,

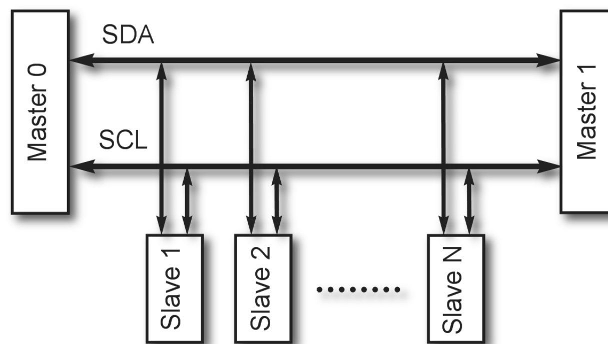


Figure 4.1: The I2C master slave configuration.

it is currently capable of operating at speeds up to and including 3.4 Mbits/sec. Any device on the bus that initiates a data transfer[7], generates the clock for that transfer and is defined as the then current *master*. The corresponding device receiving the data is the *slave*. Each of the slaves connected to the bus is assigned a unique address[8] that is used by the master to specify which slave is being addressed, to receive a given data transmission. The maximum number of slaves that can be attached to SDA/SCL is determined by the bus capacitance, e.g., 400 pF. Bit transfers are level-triggered, with one bit per clock pulse as the data rate and data changes can only take place during low clocks.

The serial data (SDL) and serial clock (SCL) signal lines are used in combination to transmit data. If both SCL and SDL are high, no data is transmitted. A high-to-low transition of the

[5]The abbreviations *I2C* and I^2C are both in common usage, when referring to the inter-integrated circuit bus.

[6]A slave is defined as any device connected to the bus that is capable of receiving data, e.g., an LCD driver, memory, keyboard driver, microcontroller, etc. In some cases, a device is capable of both receiving and transmitting data and therefore may alternately function as both a master and a slave. Thus data transfer on the bus can be in either direction. A device capable of serving as a master can also request data from another device, in which case, that master generates the clock and terminates the transfer.

[7]Including transmitting the address of the device (slave) that is to receive the data.

[8]Addressing for each device can be based on either 7- or 10-bit addressing.

©Copyright 2012 Cypress Semiconductor

SDA line, while the SCL line is high, indicates a *START* condition, which is often designated by *S*. A low-to-high transition of the SDA line, while the SCL line is high, defines a *STOP* condition designated by *P*. Only the current master is capable of generating a *START* condition and, once initiated, the bus enters a busy state until a corresponding *STOP* condition has occurred. If the device addressed by the master is busy, the master may generate a series of *START*s to maintain the bus in a busy state, until the addressed slave becomes available.

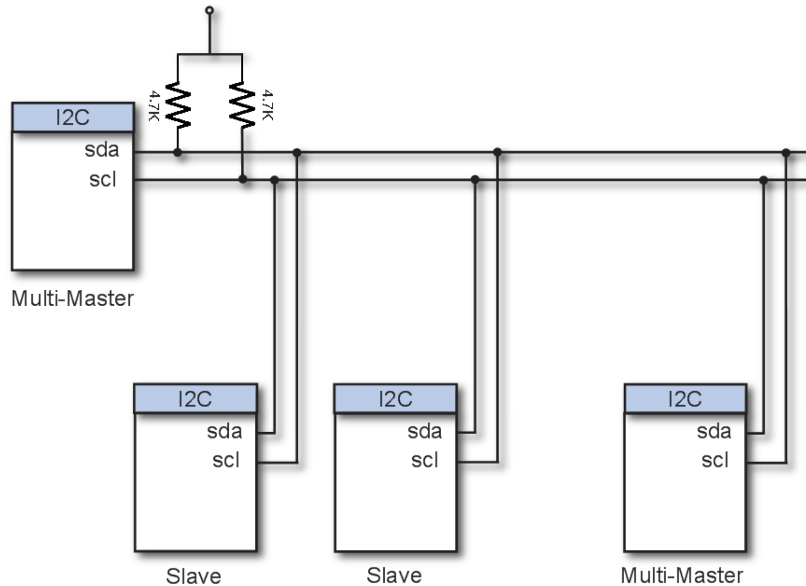


Figure 4.2: I2C master and multiple slave configuration.

A single bit of each 8-bit data byte is transferred for each clock pulse and there is no inherent limit to the number of bytes that can be transmitted in a given transmission.[9] At the end of the transmission of each byte, an acknowledgement from the receiver is required.[10] The master generates an acknowledge clock pulse and releases the SDA line which then goes high for the duration of the acknowledge clock pulse. The receiving device pulls the SDA line low during the acknowledge clock pulse. If the slave does not acknowledge, the SDA line remains high and the master can then either generate a *STOP*, to terminate the transmission, or a repeated *START* condition to start a new transfer, as shown in Figure 4.4. A slave can temporarily suspend further data transmission by placing the SCL line in a low state which results in the master entering a wait state. This capability allows the slave to perform other functions, e.g., servicing an interrupt. When the slave subsequently releases the SCL line, the next byte can then be transmitted. In an actual implementation, the two lines, SDL and SCL, are connected to pullup resistors as shown in Figure 4.2.

In order for a master to transmit data, the bus must be free. If multiple masters are used in an I2C configuration, a method must be provided to avoid two, or more, masters attempting to transmit data at the same time. This is accomplished by the use of an *arbitration technique*. Each master generates its own clock signals during data transfers on the bus. These signals can be *stretched* by either another master, as a result of arbitration, or by the slow response slave that

[9] Bytes are transferred with the *Most Significant bit* (MSb) being transferred first.
[10] This restriction is relaxed, if one or more of the devices involved is a CBUS receiver. In such cases, a third bus line is required.

©Copyright 2012 Cypress Semiconductor

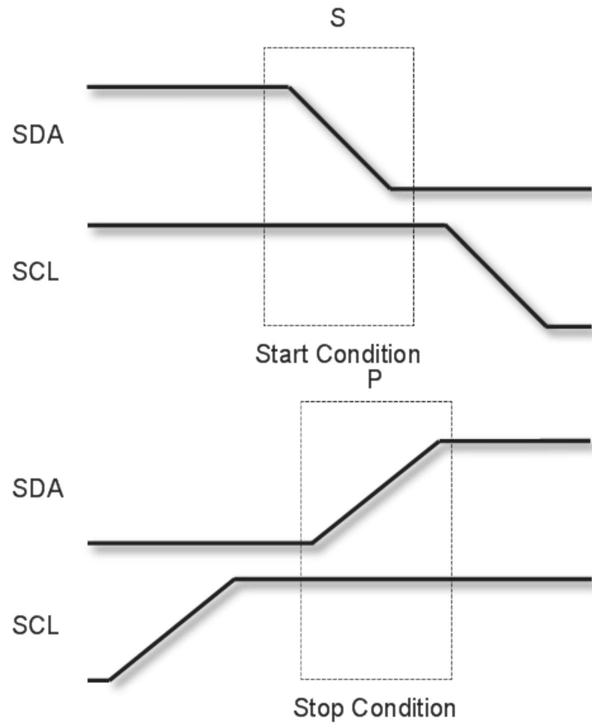


Figure 4.3: Start and Stop conditions.

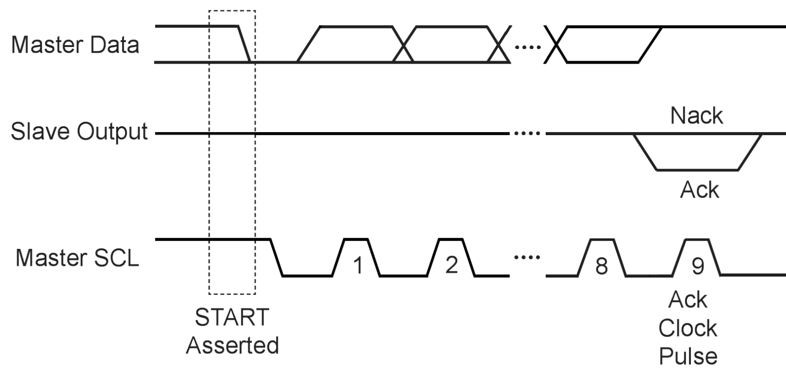


Figure 4.4: Slave ACK/NACK of single byte received.

©Copyright 2012 Cypress Semiconductor

holds down the clock line. If two, or more, masters attempt to use the bus contemporaneously, the first to introduce a one, while the other introduces a zero, will lose the arbitration and control passes to the latter master. This arbitration may continue in force for multiple bit transfers.

When multiple masters are used, it is possible for two, or more, of them to generate a *START* condition within the minimum hold time of the *START* condition. Therefore, for each byte to be transmitted, the bus must first be checked to determine whether or not it is in a *busy state*. An error is returned to the master who loses arbitration.

A PSoC3/5 master can be operated in either manual or automatic mode. In automatic mode, a buffer is employed that holds the entire transfer. If a write operation is to occur, the buffer is pre-filled with the data to be transmitted. If data is to be read from a slave, a buffer of at least the size of a packet has to allocated. In the automatic mode, the following function[11] writes an array of bytes to a slave

```
uint8 I2C_MasterWriteBuf(uint8 SlaveAddr, unit8 * wrData, uint8 cnt, uint8 mode)
```

where *SlaveAddr* is a right-justified, 7-bit slave address; *wrData* is a pointer to the array of data; *cnt* is the number of bytes to be transferred and *mode* determines how the transfer starts and stops.

Similarly, a read operation is initiated by

```
uint8 I2C_MasterReadBuf(uint8 SlaveAddr, unit8 * wrData, uint8 cnt, uint8 mode)
```

Both of these functions return status information, as shown in Table 4.1.

Table 4.1: Master status info returned by *unit8 I2C_ MasterStatus(void).*

Master Status Constants	Descriptions
I2C_MSTAT_RD_CMPLT	Read transfer complete
I2C_MSTAT_WR_CMPLT	Write transfer complete
I2C_MSTAT_XFER_INP	Transfer in progress
I2C_MSTAT_XFER_HALT	Tranfer has been halted
I2C_MSTAT_ERR_SHORT_XFER	Transfer completed before all of the bytes were transferred
I2C_MSTAT_ADDR_NAK	Slave did not acknowledge address
I2C_MSTAT_ERR_ARB_LOST	Master lost abitration during communications with slave
I2C_MSTAT_ERR_XFER	Error occurred during transfer

4.2.1 Application Programming Interface

PSoC Creator provides a set of I2C application programming interface routines (APIs) to allow dynamic configuration of the I2C component during runtime. By default, PSoC Creator assigns

[11]The use of the term *function,* in the present context and throughout this text, is a generic reference to methods, function members or member functions.

©Copyright 2012 Cypress Semiconductor

the instance name *I2C_1* to the first instance of an *I2C* component in a given design. This instance can be renamed to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following is *I2C*. All API functions assume that the data direction is from the perspective of the I2C master. A write event occurs when data is written from the master to the slave, and a read event occurs when the master reads data from the slave.

PSoC Creator supports a number of function calls that are generic for I2C slave or master operation including:

- *uint8 I2C_MasterClearStatus(void)* clears all status flags and returns the master status and returns the current status of the master.
- *uint8 I2C]_MasterWriteBuf(uint8 slaveAddress, uint8 * wrData, uint8 cnt, uint8 mode)* automatically writes an entire buffer of data to a slave device. Once the data transfer is initiated by this function, further data transfer is handled by the included ISR in byte-by-byte mode and it enables the I2C interrupt.
- *uint8 I2C_MasterReadBuf(uint8 slaveAddress, uint8 * rdData, uint8 cnt, uint8 mode)* automatically reads an entire buffer of data from a slave device. Once the data transfer is initiated by this function, further data transfer is handled by the included ISR in a byte-by-byte mode and it enables the I2C interrupt.
- *uint8 I2C_MasterSendStart(uint8 slaveAddress, uint8 R_nW)* generates a start condition and sends the slave address with a read/write bit. It also disables the I2C interrupt.
- *uint8 I2C_MasterSendRestart(uint8 slaveAddress, uint8 R_nW)* generates a restart condition and sends the slave address with a read/write bit.
- *uint8 I2C_MasterSendStop(void)* generates an I2C stop condition on the bus. If the start, or restart, conditions failed before this function was called, this function does nothing.
- *uint8 I2C_MasterWriteByte(uint8 theByte)* sends one byte to a slave. A valid start, or restart, condition must be generated before calling this function. This function does nothing, if start, or restart, conditions failed before this function was called.
- *uint8 I2C_MasterReadByte(uint8 acknNak)* reads one byte from a slave and ACKs, or NAKs, the transfer. A valid start, or restart, condition must be generated before calling this function. This function does nothing and returns a zero value, if a start or restart condition has failed before this function was called.
- *uint8 I2C_MasterGetReadBufSize(void)* returns the number of bytes that has been transferred by the *I2C_MasterReadBuf()* function. If the transfer is not yet complete, it returns the byte count transferred so far.
- *uint8 I2C_MasterGetWriteBufSize(void)* returns the number of bytes that have been transferred by the *I2C_MasterWriteBuf()*function. If the transfer is not yet complete, it returns the byte count transferred so far.
- *void I2C_MasterClearReadBufSize(void)* resets the read buffer pointer back to the first byte in the buffer.
- *void I2C_MasterClearWriteBufSize(void)* resets the write buffer pointer back to the first byte in the buffer.

4.2.2 PSoC3/5 I^2C Slave-Specific Functions

The supported slave functions are as follows:

©Copyright 2012 Cypress Semiconductor

- *uint8 I2C_SlaveClearReadStatus(void)* clears the read status flags and returns their values. No other status flags are affected.
- *uint8 I2C_SlaveClearWriteStatus(void)* clears the write status flags and returns their values. No other status flags are affected.
- *void I2C_SlaveSetAddress(uint8 address)* sets the I2C slave address.
- *void I2C_SlaveInitReadBuf(uint8 * rdBuf, uint8 bufSize)* sets the buffer pointer and size of the read buffer and resets the transfer count returned by the *I2C_SlaveGetReadBufSize()* function.
- *void I2C_SlaveInitWriteBuf(uint8 * wrBuf, uint8 bufSize)* sets the buffer pointer and size of the write buffer. This function also resets the transfer count returned by the *I2C_SlaveGetWriteBufSize()* function.
- *uint8 I2C_SlaveGetReadBufSize(void)* returns the number of bytes read by the I2C master after an *I2C_SlaveInitReadBuf()*, or *I2C_SlaveClearReadBuf()* function was executed.
- *uint8 I2C_SlaveGetWriteBufSize(void)* returns the number of bytes written by the I2C master since an *I2C_SlaveInitWriteBuf()* or *I2C_SlaveClearWriteBuf()* function was executed. The maximum return value is the size of the write buffer.
- *void I2C_SlaveClearReadBuf(void)* resets the read pointer to the first byte in the read buffer. The next byte read by the master will be the first byte in the read buffer.
- *uint8 I2C_SlaveGetWriteBufSize(void)* returns the number of bytes written by the I2C master since an *I2C_SlaveInitWriteBuf()* or *I2C_SlaveClearWriteBuf()* function was executed. The maximum return value is the size of the write buffer.
- *void I2C_SlaveClearReadBuf(void)* resets the read pointer to the first byte in the read buffer. The next byte read by the master will be the first byte in the read buffer.

4.2.3 PSoC3/5 I^2C Master/Multi-Master Slave

PSoC Creator includes a number of I2C components that support master, multi-master and slave configurations with clocks rates up to 1 megabit per second, inclusive. A typical configuration is shown in Figure 4.1 with two pullup resistors whose value depends on the applicable supply voltage, clock speed and bus capacitance.

This component has four[12] I/O connections:

- *Clock* - is used to clock the transmission of data on the I2C bus and is derived from the bus as shown in Table 4.2.

Table 4.2: Bus frequencies required for a 16X oversampling clock.

Bus	Clock
50 kbps	800 kHz
100 kbps	1.6 MHz
400 kbps	6.4 MHz
1000 kbps	16 MHz

[12]The clock and reset pins are only visible in PSoC Creator when the *Implementation* parameter is set to UDB.

©Copyright 2012 Cypress Semiconductor

- *Reset* - maintains the I2C block in a hardware reset state, thereby halting I2C communications. A software reset can be invoked by using the *I2C_Stop()* and *I2C_Start()* APIs.[13]
- *sda* - the serial data i/o channel used to transmit/receive I2C bus data.
- *scl* - the master-generated I2C clock. The slave cannot generate a the clock signal, but it can hold the clock low, suspending all bus activity until the slave is ready to send data, or ACK/NAK[14] the latest data, or address.[15]

Address decoding can be based on either hardware, which is the default case, or on software.If only a single slave is involved in the design, hardware decoding is preferable. If hardware address decoding is enabled , the I2C component will automatically NAK addresses other than its own, unless CPU intervention occurs. Each slave recognizes its unique address which is between 00x00 and 0x7F, with a default address of 0x04. A 10-bit address can be used by employing software address decoding, but requires that the second byte of the address be decoded, as well.

Signal connections for the SDA and SCL lines can be one of three possible types:

- I2C0 - SCL = SIO pin P12[0], SDA = SIO pin P12[1].
- I2C1 - SCL = SIO pin P12[4], SDA = SIO pin P12[5].
- Any (Default) - Any GPIO or SIO pins via schematic routing.

PSoC Creator supports four modes of operation:

- slave-only operation,
- master-only operation,
- multi-master which supports more than one master,

and,

- multi-master-slave which supports simultaneous multi-master and slave operation.

A slave employs two memory buffers, viz., one for data received from the master and one for the master to read data transmissions from the slave. The I2C slave read and write buffers are set by the initialization commands,

```
void I2C_SlaveInitReadBuf(uint8 * rdBuf, uint8 bufSize)
void I2C_SlaveInitWriteBuf(uint8 * wrBuf, uint8 bufSize)
```

However, these commands do not allocate memory, but instead copy the array pointer and size to the internal component variables. The arrays used for the buffers must be set programmatically because they are not automatically generated by the component. Using these functions sets a pointer and byte count for the read and write buffers. The bufSize for these functions may be less than, or equal to, the actual array size, but it should never be larger than the available memory pointed to by the *rdBuf* or *wrBuf* pointers. When the *I2C_SlaveInitReadBuf(),* or the *I2C_SlaveInitWriteBuf()* functions are called, the internal index is set to the first value in the array pointed to by *rdBuf* and *wrBuf*, respectively. As bytes are read/written by the I2C master, the index is incremented until the offset is one less than the *byteCount.* The number of bytes transferred may be determined by calling either *I2C_SlaveGetReadBufSize()* or *I2C_SlaveGetWriteBufSize()* for the read/write buffers, respectively. However, reading/writing, more bytes than are in the buffers causes an overflow error which results in the slave status byte being set.[16]

To reset the index back to the beginning of the array, i.e., zero, use the following commands:

[13]The reset input may be left floating that is by default equivalent to asserting a logic zero signal to the reset pin.

[14]ACK (acknowledged), NAK (not acknowledged) or NACK (not acknowledged) are handshaking signals.

[15]The pin that is connected to *scl* should be configured as *Open-Drain-Drives-Low.*

[16]This byte can be read via the *I2C_SlaveStatus()* API.

©Copyright 2012 Cypress Semiconductor

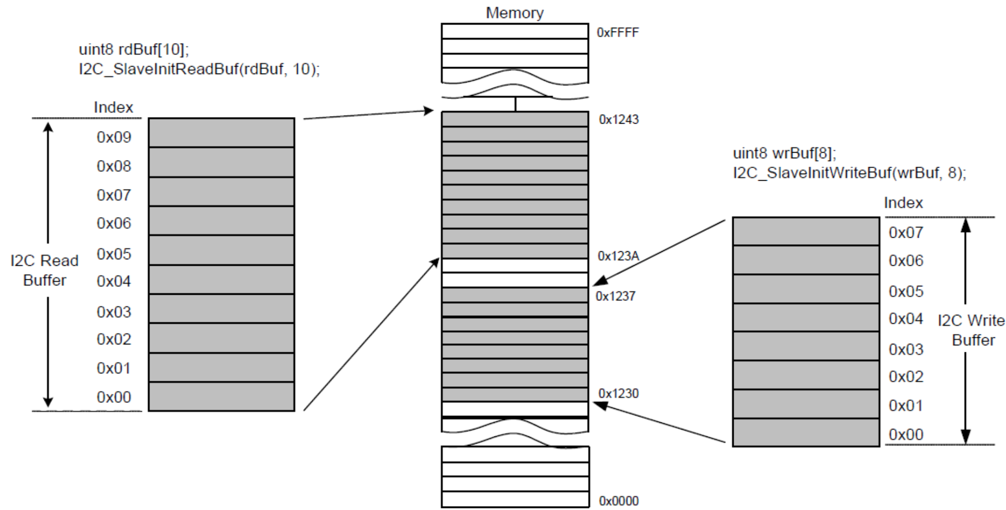


Figure 4.5: Slave buffer structure.

```
void I2C\_SlaveClearReadBuf(void)
void I2C\_SlaveClearWriteBuf(void)
```

The next byte read/write to/by the I2C SPI is the first byte in the array.[17] Multiple reads, or writes, by the I2C master continue to increment the array index until a clear buffer command occurs, or the array index exceeds the array size. Figure 4.6 shows an example where an I2C master has executed two write transactions. The first write was four bytes and the second write was six bytes. The sixth byte in the second transaction was *NAK*ed by the slave to signal that the end of the buffer had occurred. If the master tries to write a seventh byte for the second transaction, or starts to write more bytes with a third transaction, each subsequent byte will be *NAK*ed and discarded until the buffer is reset. Using the *I2C_SlaveClearWriteBuf()* function, after the first transaction, resets the index to zero and causes the second transaction to overwrite the data from the first transaction.[18]

4.2.4 Master and Multimaster Functions

PSoC3/5 Master and Multi-Master[19] operations are basically the same, with two exceptions. When operating in Multi-Master mode, the bus should always be checked to see if it is busy. Another master may already be communicating with a slave. In this case, the program must wait until the current operation is complete, before issuing a start transaction. The is accomplished by checking the appropriate return value, to determine whether or not an error condition has been set, thereby indicating that another master has control of the bus. The second difference is that, in Multi-Master mode, two masters can start at the exact same time.

[17]Before these clear buffer commands are used, the data in the arrays should be read or updated.

[18]The data in the buffer should be processed by the slave before resetting the buffer index.

[19]In a fixed-function implementation, which does not support undefined bus conditions, for PSoC 3 ES2 and PSoC 5, and Master or Multi-Master mode, if the *STOP* condition is set by the software immediately after the *START* condition, the module will generate the *STOP* condition. This occurs after the address field sends 0xFF, if a data write, and the clock line remains low. To avoid this condition, the *STOP* condition should not be set immediately after *START*. At least one byte should be transferred followed by setting *STOP* condition and after *a NAK* or *ACK*.

©Copyright 2012 Cypress Semiconductor

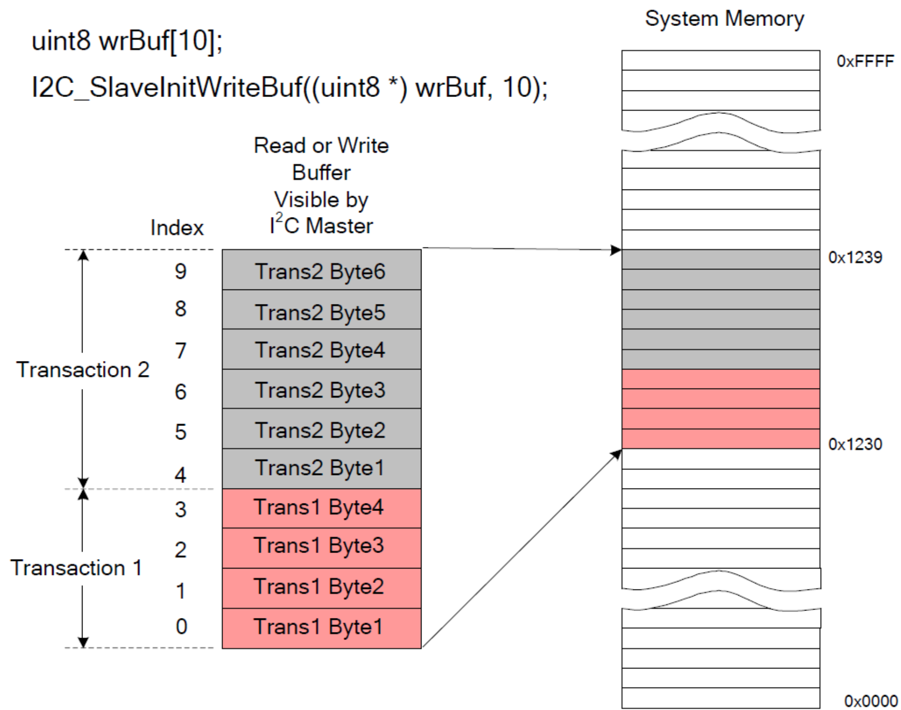


Figure 4.6: I2C write transaction.

If this happens, one of the two masters must yield control of the the bus and this is accomplished by arbitration. arbitration. A check for this condition must be made after each byte is transferred. The I2C component automatically checks for this condition and responds with an error, if arbitration is lost. Two options are available when operating the I2C master, viz., manual and automatic. In the automatic mode, a buffer is created to hold the entire transfer. In the case of a write operation, the buffer is pre-filled with the data to be sent. If data is to be read from the slave, a buffer of at least the size of the packet to be transmitted needs to be allocated. The following function will write an array of bytes to a slave in automatic mode

```
uint8 I2C\_MasterWriteBuf(uint8 slaveAddress, uint8 * xferData, uint8 cnt, uint8
mode)
```

The *slaveAddress* variable is a right-justified, 7-bit slave address ranging from 0 to 127, inclusive. The component's API automatically appends the write flag to the LSb of the address byte. The array of data to transfer is pointed to by the second parameter, *xferData* and the *cnt* parameter is the number of bytes to transfer. The last parameter, *mode*, determines how the transfer starts and stops. A transaction may begin with a restart instead of a start, or halt before the stop sequence. These options allow *back-to-back transfers* where the last transfer does not send a stop, and the next transfer issues a restart, instead of a start.

A read operation is almost identical to the write operation and the same parameters with the same constants are used.

```
uint8 I2C\_MasterReadBuf(uint8 slaveAddress, uint8 * xferData, uint8 cnt, uint8
mode);
```

Both of these functions return status. See the Table 4.1 for the *I2C_MasterStatus()* function return values. Because the read and write transfers complete in the background, during the I2C

©Copyright 2012 Cypress Semiconductor

interrupt code, the *I2C_ MasterStatus()* function can be used to determine when the transfer has been completed.

The following code snippet shows a typical write to a slave.

```
I2C_MasterClearStatus(); /* Clear any previous status */
I2C_MasterWriteBuf(4, (uint8 *) wrData, 10, I2C_MODE_COMPLETE_XFER);
for(;;)
{
   if(0u != (I2C_MasterStatus() & I2C_MSTAT_WR_CMPLT))
   {
    /* Transfer complete. Check Master status to make sure that
       transfer completed without errors. */
    break;
   }
}
```

The I2C master can also be operated manually. In this mode, each part of the write transaction is performed with individual commands.

```
status = I2C\_MasterSendStart(4, I2C_WRITE_XFER_MODE);
if(status == I2C\_MSTR\_NO\_ERROR) /* Check if transfer completed without errors */
{
    /* Send array of 5 bytes */
    for(i=0; i<5; i++)
    {
       status = I2C\_MasterWriteByte(userArray[i]);
       if(status != I2C\_MSTR\_NO\_ERROR)
          {
            break;
          }
    }
}
I2C_MasterSendStop(); /* Send Stop */
```

A manual read transaction is similar to the write transaction, except that the last byte should be *NAK*ed.

The example below represents a typical manual read transaction.

```
status = I2C\_MasterSendStart(4, I2C_READ_XFER_MODE);
if(status == I2C\_MSTR\_NO\_ERROR) /* Check if transfer completed without errors */
{
    /* Read array of 5 bytes */
   for(i=0; i<5; i++)
   {
      if(i < 4)
      {
         userArray[i] = I2C\_MasterReadByte(I2C_ACK_DATA);
      }
      else
      {
         userArray[i] = I2C_MasterReadByte(I2C_NAK_DATA);
```

©Copyright 2012 Cypress Semiconductor

```
      }
   }
}
I2C_MasterSendStop(); /* Send Stop */
```

4.2.5 Multi-Master-Slave Mode

In this mode of operation both the Multi-Master and Slave are operational. Although the component can be addressed as a slave, firmware must initiate any master-mode transfers. Enabling *Hardware Address Match* introduces some limitations with respect to arbitrage and address bytes. In the event that the master loses arbitration during an address byte, the hardware reverts to Slave-mode and the byte received generates a slave address interrupt, provided that the slave is addressed. Otherwise, the *lost arbitrage status* will no longer be available to interrupt-based functions.[20] However, the manual-based function, *I2C_ MasterSendStart()*,[21] does return correct status information, as shown in Table 4.3, for this particular case.

Table 4.3: *I2C_ MasterSendStart()* return values.

Mode Constants	Description
I2C_MSTR_NO_ERROR	Function completed without error
I2C_MSTR_BUS_BUSY	Bus is busy occurred. START condition generation not started.
I2C_MSTR_SLAVE_BUSY	Slave operation in progress.
I2C_MSTR_ERR_LB_NAK	Last byte was NAKed.
I2C_MSTR_ERR_ARB_LOST	Master lost arbitration while generating START.

4.2.6 Multi-Master-Slave Mode Operation

Both Multi-Master and Slave are operational in this mode. The component may be addressed as a slave, but firmware may also initiate master mode transfers. In this mode, when a master loses arbitration, during an address byte, the hardware reverts to Slave mode and the received byte generates a slave address interrupt.

4.2.7 Arbitrage on address byte limitations (*Hardware Address Match* enabled)

When a master loses arbitration during an address byte, the slave address interrupt is only generated if slave is addressed. In other cases, the lost arbitrage status is no longer available to interrupt-based functions. The software address detect eliminates this possibility, but excludes the *Wakeup on Hardware Address Match* feature. The manual function, *I2C_ MasterSendStart()*, provides correct status information in the case described above.

[20]Using software address detection prevents this status from being lost but excludes the *Wakeup on Hardware Address Match* feature.

[21]This function generates a *START* condition and sends the slave address with a read/write bit.

©Copyright 2012 Cypress Semiconductor

4.2.8 Start of Multi-Master-Slave Transfer

When using Multi-Master-Slave, the Slave can be addressed at any time. The Multi-Master must have time to prepare to generate a start condition when the bus is free. During this time, the Slave can be addressed and, in this case, the Multi-Master transaction is lost and Slave operation proceeds. Care must be exercised not to break the Slave operation; the I2C interrupt must be disabled before generating a start condition to prevent the transaction from passing the address stage. This action allows a Multi-Master transaction to be aborted and to start a Slave operation correctly.

The following cases are possible when disabling the I2C interrupt:

- The bus is busy (Slave operation is in progress or other traffic is on the bus) before the start generation. The Multi-Master does not try to generate a start condition. Slave operation proceeds when the I2C interrupt is enabled. The *I2C_MasterWriteBuf()*, *I2C_MasterReadBuf()*, or *I2C_MasterSendStart()* call returns the status *I2C_MSTR_BUS_BUSY*. The bus is free before start generation. The Multi-Master generates a start condition on the bus and proceeds with operation when I2C interrupt is enabled. The *I2C_MasterWriteBuf()*, *I2C_MasterReadBuf()*, or *I2C_MasterSendStart()* call returns the status *I2C_MSTR_NO_ERROR*.
- The bus is free before start generation. The Multi-Master tries to generate a start but another Multi-Master addresses the Slave before this and the bus becomes busy. The start condition generation is queued. The Slave operation stops at the address stage because of a disabled I2C interrupt. When I2C interrupt is enabled, the Multi-Master transaction is aborted from queue and Slave operation proceeds. The *I2C_MasterWriteBuf()* or
  - I2C_MasterReadBuf() call does not notice this and returns *I2C_MSTR_NO_ERROR*. The *I2C_MasterStatus()* returns *I2C_MSTAT_WR_CMPLT* or *I2C_MSTAT_RD_CMPLT* with *I2C_MSTAT_ERR_XFER* (all other error condition bits are cleared) after the Multi-Master transaction is aborted. The *I2C_MasterSendStart()* call returns the error status. *I2C_MSTR_ABORT_XFER*.

4.2.9 Interrupt Function Operation

It is possible to assign a priority to a master or slave transaction utilizing interrupts as shown by the following coding example:

- I2C_MasterWriteBuf();
- I2C_MasterReadBuf();

```
I2C\_MasterClearStatus(); /* Clear any previous status */
I2C\_DisableInt(); /* Disable interrupt */
status = I2C\_MasterWriteBuf(4, (uint8 *) wrData, 10, I2C\_MODE\_COMPLETE\_XFER);
/* Try to generate, start. The disabled I2C interrupt halts the transaction in the
address stage, if a Slave is addressed or the Master generates a start condition */
I2C\_EnableInt(); /* Enable interrupt and proceed with the Master or Slave
transaction */
for(;;)
{
if(0u != (I2C\_MasterStatus() & I2\_MSTAT_WR_CMPLT))
{
/* Transfer complete.
```

©Copyright 2012 Cypress Semiconductor

```
Check Master status to make sure that transfer
completed without errors. */
break;
}
}
if (0u != (I2C_MasterStatus() & I2C_MSTAT_ERR_XFER))
{
         /* Error occurred while transfer, clean up Master status and
            retry the transfer */
}
```

4.2.10 Manual Function Operation

Manual Multi-Master operation assumes that I2C interrupt is disabled, but it is advisable to take the following precaution:

```
I2C_DisableInt(); /* Disable interrupt */
status = I2C_MasterSendStart(4, I2C_WRITE_XFER_MODE);; /* Try to generate start
condition */
if (status == I2C_MSTR_NO_ERROR) /* Check if start generation completed without
errors */
{
/* Proceed the write operation */
/* Send array of 5 bytes */
for(i=0; i<5; i++)
{
status = I2C_MasterWriteByte(userArray[i]);
if(status != I2C_MSTR_NO_ERROR)
{
break;
}
}
I2C_MasterSendStop(); /* Send Stop */
}
I2C_EnableInt(); /* Enable interrupt, if it was enabled before */
```

4.2.11 Wakeup and Clock Stretching

The I2C block responds to transactions on the I2C bus, during sleep mode. If the incoming address matches with the slave address, the I2C wakes the system. Once the address matches, a wakeup interrupt is asserted to wake up the system and SCL is pulled low. An ACK is sent out after the system wakes up, and the CPU determines the next action in the transaction.

The I2C slave stretches the clock while exiting sleep mode, as shown by Figure 4.7. All clocks in the system must be restored before continuing the I2C transactions. The I2C interrupt is disabled before going to sleep and only enabled after the *I2C_ Wakeup()* function is called. During the time between wakeup and end of calling *I2C_ Wakeup()*, SCL line is pulled low.

```
...
      I2C_Sleep();         /* Go to Sleep and disable I2C interrupt */
      CyPmSaveClocks();    /* Save clocks settings */
      CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_I2C);
```

©Copyright 2012 Cypress Semiconductor

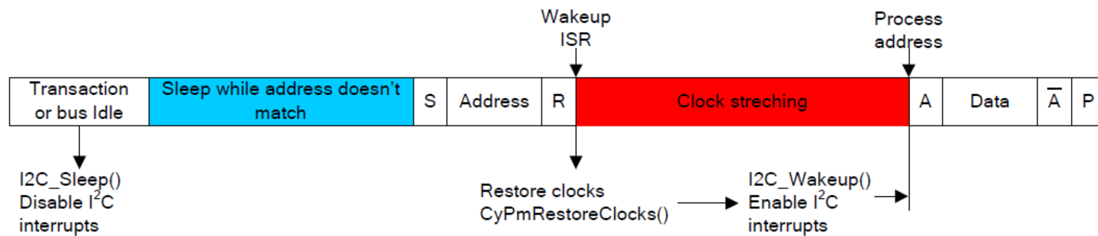


Figure 4.7: Wakeup and clock stretching.

```
    CyPmRestoreClocks(); /* Restore clocks */
    I2C_Wakeup(); /* Wakeup, enable I2C interrupt and ACK the address, till
    end of this call the SCL is pulled low */
...
```

4.2.12 Slave Operation

The slave interface consists of two memory buffers, one for data written to the slave by a master and a second for data read by a master from the slave.[22] The I2C slave read and write buffers are set by the initialization commands discussed below. These commands do not allocate memory, but instead copy the array pointer and array size to the internal component variables. The arrays used for the buffers must be instantiated because they are not automatically generated by the component. The same buffer can be used for both read and write buffers, but care must be exercised to manage the data properly.

The following functions set a pointer and byte count for the read and write buffers.

```
void I2C_SlaveInitReadBuf(uint8 * rdBuf, uint8 bufSize)
void I2C_SlaveInitWriteBuf(uint8 * wrBuf, uint8 bufSize)
```

The *bufSize* for these functions may be less than, or equal to, the actual array size, but it should never be larger than the available memory pointed to by the *rdBuf*, or *wrBuf*, pointers. When the *I2C_SlaveInitReadBuf()* or *I2C_SlaveInitWriteBuf()* functions are called, the internal index is set to the first value in the array pointed to by *rdBuf* and *wrBuf*, respectively. As the I2C master reads, or writes the bytes, the index is incremented until the offset is one less than the *byteCount*. At any time, the number of bytes transferred can be queried by calling either I2C_SlaveGetReadBufSize() or I2C_SlaveGetWriteBufSize() for the read and write buffers, respectively. Reading or writing more bytes than are in the buffers causes an overflow error. The error is set in the slave status byte and can be read with the *I2C_SlaveStatus()* API. To reset the index back to the beginning of the array, use the following commands.

```
void I2C\_SlaveClearReadBuf(void)
void I2C\_SlaveClearWriteBuf(void)
```

This resets the index back to zero. The next byte the I2C master reads or writes to is the first byte in the array. Before using these clear buffer commands, the data in the arrays should be read or updated.

Multiple reads or writes, by the I2C master, continue to increment the array index until the clear buffer commands are used or the array index tries to grow beyond the array size.

[22]Reads and writes are from the perspective of the I2C master.

©Copyright 2012 Cypress Semiconductor

Figure 2 shows an example where an I2C master has executed two write transactions. The first write was four bytes and the second write was six bytes. The sixth byte in the second transaction was NAKed by the slave to signal that the end of the buffer had occurred. If the master tried to write a seventh byte for the second transaction or started to write more bytes with a third transaction, each byte would be NAKed and discarded until the buffer is reset. Using the *I2C_SlaveClearWriteBuf()* function after the first transaction resets the index back to zero and causes the second transaction to overwrite the data from the first transaction. Make sure data is not lost by overflowing the buffer. The data in the buffer should be processed by the slave before resetting the buffer index. Both the read and write buffers have four status bits to signal that

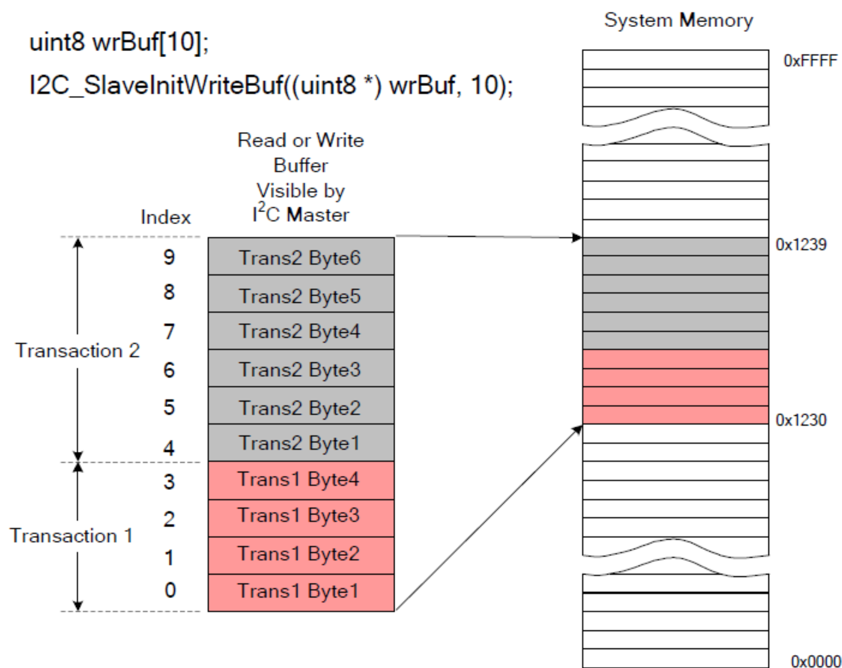


Figure 4.8: Slave buffer structure.

a transfer is complete, a transfer is in progress, and buffer overflow. Starting a transfer sets the busy flag and when the transfer has been completed, the transfer complete flag is set and the busy flag is cleared. If a second transfer is started, both the busy and transfer complete flags can be set at the same time. The values for the slave status constants are shown in Figure 4.4.

4.2.13 Start of Multi-Master-Slave Transfer

When using multi-master-slave, the slave can be addressed at any time. The multi-master must take time to prepare to generate a *Start* condition when the bus is free. During this time, the slave could be addressed and, if so, the multi-master transaction is lost and the slave operation proceeds. Care must be exercised to avoid breaking the slave operation and the I2C interrupt must be disabled before generating a *Start* condition to prevent the transaction from passing the address stage. This actions allows a multi-master transaction to be aborted and a slave operation to be started correctly.

The following cases are possible when disabling the I2C interrupt:

©Copyright 2012 Cypress Semiconductor

Table 4.4: I2C slave status constants.

Slave Status Constants	Value	Description
I2C_SSTAT_RD_CMPLT	0x01	Slave read transfer complete
I2C_SSTAT_RD_BUSY	0x02	Slave read transfer in progress (busy)
I2C_SSTAT_RD_OVFL	0x04	Master attempted to read more bytes than are in the buffer
I2C_SSTAT_WR_CMPLT	0x10	Slave write transfer complete
I2C_SSTAT_WR_CMPLT	0x20	Slave write transfer in progress (busy)
I2C_SSTAT_WR_CMPLT	0x40	Master attempted to read more bytes than are in the buffer

- The bus is busy, e.g., slave operation is in progress or other traffic is on the bus, before *Start* generation. The multi-master does not try to generate a *Start* condition. Slave operation proceeds when the I2C interrupt is enabled. The *I2C_MasterWriteBuf()*, *I2C_MasterReadBuf()*, or *I2C_MasterSendStart()* call returns the status *I2C_MSTR_BUS_BUSY*.
- The bus is free, before *Start* generation. The multi-master generates a Start condition on the bus and proceeds with operation when the I2C interrupt is enabled. The *I2C_MasterWriteBuf()*, *I2C_MasterReadBuf()*, or *I2C_MasterSendStart()* call returns the status *I2C_MSTR_NO_ERROR*.
- The bus is free before Start generation. The multi-master tries to generate a Start, but another multi-master addresses the slave before this, and the bus becomes busy. The Start condition generation is queued. The slave operation stops at the address stage because of a disabled I2C interrupt. When the I2C interrupt is enabled, the multi-master transaction is aborted from the queue, and the slave operation proceeds. The *I2C_MasterWriteBuf()* or I2C_MasterReadBuf() call does not notice this and returns *I2C_MSTR_NO_ERROR*.

  The I*2C_MasterStatus()* returns *I2C_MSTAT_WR_CMPLT* or *I2C_MSTAT_RD_CMPLT* with *I2C_MSTAT_ERR_XFER* (all other error condition bits are cleared) after the multi-master transaction is aborted. The *I2C_MasterSendStart()* call returns the error status *I2C_MSTR_ABORT_XFER*.

4.3 Universal Asynchronous Rx/Tx (UART)

PSoC Creator's UART component provides asynchronous communications is often employed to implement the RS232, or RS485[23] protocols[24]. The UART component can be configured for Full

[23]The RS485, also referred to as TIA-485 or EIA-485, protocol is similar to RS232 but differs in that it is a more noise-immune protocol than RS232 and it allows as many as 32 devices to share a common, 3-wire bus and communicate over distances as long as 4000 feet (1200 meters). The transmission path is differential (balanced) and consists of a twisted pair and a third wire which serves as ground (there is also a four-wire configuration) that provides very high noise immunity.

[24]The UART can also be employed in a TTL-compatible mode.

©Copyright 2012 Cypress Semiconductor

Duplex[25], Half Duplex[26], RX-only, or TX-only versions[27]. However, all four transmission modes have the same basic functionality differing only in the amount of resources used. Two configurable buffers, each of independent size, serve as circular receive and transit buffers that are assigned in SRAM and hardware FIFOs to ensure data integrity.

This arrangement allows allows the CPU to spend more time on critical, real time tasks than servicing the UART. In most cases, the UART is configured by choosing the baud rate[28], parity[29], number of data bits, and number of start bits. The most common configuration for RS232 is eight data bits, no parity, and one stop bit and designated as *8N1* and is the default configuration. A second common use for UARTs is in multidrop[30] RS485 networks.

The UART component supports a 9-bit addressing mode with hardware address detect, as well as, a TX output enable signal to enable the TX transceiver during transmissions. There are a number of physical-layer and protocol-layer variations of UARTs in common use including RS423[31], DMX512, MIDI, LIN[32] bus, legacy terminal protocols, and IrDA[33].

To support the more commonly used variations, the number of data bits, stop bits, parity, hardware flow control, and parity generation and detection is configurable from within PSoC Creator and under software control. As a hardware-compiled option, a clock and serial data stream can be used that transmits the UART data bits only on the clock's rising edge. An independent clock and data output can also be employed for both TX and RX. These outputs allow automatic calculation of the data CRC by connecting a CRC component to the UART.

The PSoC Creator UART component, shown in Figure 4.9, has the following features:

- 8x and 16x oversampling,
- 9-bit address mode with hardware address detection,
- Baud rates from 110 to 921,600 bits per second (bps) or arbitrary up to 4 Mbps
- Break signal detection and generation,
- Detection of framing, parity and overrun errors,
- Full duplex, half duplex, TX only, RX only, optimized hardware,
- Rx and Tx buffers = 4 to 65,535 bytes,

and,

- Two out of three voting, per bit.

[25]A full duplex system allows simultaneous transmissions in both directions over the communications path.

[26]A half duplex system allows transmissions in both directions over the communications path but not simultaneously.

[27]The UART can also be configured for more advanced protocols such as DMX512, LIN and IrDA or custom protocols.

[28]Baud rate refers to the rate at which bits are transmitted per second.

[29]Parity in the present context refers to the use of an optional bit associated with each transmitted byte which has the value 1 if the number of 1s in the byte is even and 0, otherwise. This provides a mechanism to determine, whether or not, the integrity of a byte has been compromised upon being received.

[30]Multidrop implies multiple slaves.

[31]RS423, also referred to as EIA-423 and TIA-423, is an unbalanced (single-ended) interface, that is RS232-like, and employs a single, unidirectional, driver, that is capable of supporting up to 10 slaves. It is normally implemented in integrated circuit technology and can also be employed for the interchange of serial binary signals between DTE & DCE.

[32]The LIN bus is an inexpensive single wire bus capable of operating at baud rates up to 19.2 kbits/second, employed in a master-slave configuration having a single master and on or more slaves.

[33]The Infrared Data Association (IrDA) has established a standard protocol for modulation/demodulation methods and other physical parameters associated with infrared transceivers.

©Copyright 2012 Cypress Semiconductor

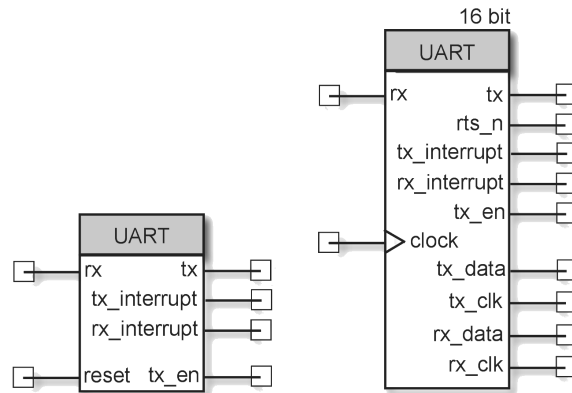


Figure 4.9: PSoC Creator UART configurations.

The UART's *clock* input determines the serial communication baud rate (bit-rate) which is one-eighth, or one-sixteenth, of the input clock frequency, depending on the value selected for the *Oversampling Rate* parameter. This input is visible, in PSoC Creator, if the *Clock Selection* parameter is set to *External Clock*. If the internal clock is selected, the desired baud rate must be selected during configuration[34]. Resetting the UART, via the *reset* input[35], places the state machines, *RX* and *TX*, in the idle state, in which case, any data that was currently being transmitted, or received, is discarded.

The *rx* input carries the input serial data from another device on the serial bus.[36] The *tx_output* connection is visible only if the Mode parameter is set to *TX Only*, *Half Duplex*, or *Full UART (RX + TX)*.[37] The *tx_en* output[38] is used primarily for RS485 communication to show that the component is transmitting on the bus. This output goes high, before a transmit starts, low when transmit is complete and shows a busy bus to the rest of the devices on the bus. The *tx_interrupt* output is the logical OR of the group of possible interrupt sources and goes high when any of the enabled interrupt sources are true.[39] The *cts_n* input[40], (_n), an active-low input, shows that another device is ready to receive data.

The rx_interrupt output[41] is the logical OR of the group of possible interrupt sources and goes high while any of the enabled interrupt sources are true. The *tx_data* output is used to shift out the TX data to a CRC component, or other logic.[42] The *tx_clk* output[43] provides the clock edge used to shift out the TX data to a CRC component, or other logic. The *rx_data* output[44] is used to shift out the RX data to a CRC component, or other logic. The *rx_clk* output[45] provides

[34]In such cased, PSoC Creator determines the necessary clock frequency for the required baud rate.
[35]This input is a synchronous reset that requires at least one rising edge of the clock, but can be left floating and the component will assign it a constant logic 0.
[36]This input is visible and must be connected if the Mode parameter is set to *RX Only*, *Half Duplex*, or *Full UART (RX + TX)*.
[37]An external pull-up resistor should be used to protect the receiver from unexpected low impulses during active *System Reset*.
[38]This output is visible when the *Hardware TX Enable* parameter is selected.
[39]This output is visible if the *Mode* parameter is set to *TX Only* or *Full UART (RX + TX)*
[40]This input is visible if the Flow Control parameter is set to Hardware.
[41]This output is visible if the *Mode* parameter is set to *RX Only*, *Half Duplex*, or *Full UART (RX + TX)*.
[42]This output is visible when the Enable CRC outputs parameter is selected.
[43]Ibid.
[44]Ibid.
[45]Ibid.

©Copyright 2012 Cypress Semiconductor

the clock edge used to shift out the *RX* data to a CRC component, or other logic.[46]

4.3.1 UART Application Programming Interface

The API routines for the UART allow the component to be configured programmatically. The following describes the interface to each function.

- *void UART_Start(void)* is the preferred method to begin component operation. *UART_Start()* sets the *initVar* variable, calls the *UART_Init()* function, and then calls the *UART_Enable()* function.
- *void UART_Stop(void)* disables the UART operation.
- *uint8 UART_ReadControlRegister(void)* returns the current value of the control register.
- void UART_WriteControlRegister(uint8 control) writes an 8-bit value into the control register.
- *void UART_EnableRxInt(void)* enables the internal receiver interrupt.
- void UART_DisableRxInt(void) disables the internal receiver interrupt.
- void UART_SetRxInterruptMode(uint8 intSrc) configures the RX interrupt sources enabled.
- *uint8 UART_ReadRxData(void)* returns the next byte received without checking the status. The status must be checked separately.
- *uint8 UART_ReadRxStatus(void)* returns the current state of the receiver status register and the software buffer overflow status.
- *uint8 UART_GetChar(void)* returns the last received byte of data and is designed for ASCII characters. It returns a uint8 where 1 to 255 are values for valid characters and 0 indicates an error occurred, or that there is no data present.
- *uint16 UART_GetByte(void)* reads the UART RX buffer immediately and returns the received character and a error condition.
- *uint8/uint16 UART_GetRxBufferSize(void)* returns the number of received bytes remaining in the RX buffer.
- *void UART_ClearRxBuffer(void)* clears the receiver memory buffer and hardware RX FIFO of all received data.
- *void UART_SetRxAddressMode(uint8 addressMode)* sets the software-controlled Addressing mode used by the RX portion of the UART.
- *void UART_SetRxAddress1(uint8 address)* sets the first of two hardware-detectable receiver addresses.
- *void UART_SetRxAddress2(uint8 address)* sets the second of two hardware-detectable receiver addresses.
- void UART_EnableTxInt(void) enables the internal transmitter interrupt.
- *void UART_DisableTxInt(void)* disables the internal transmitter interrupt.
- *void UART_SetTxInterruptMode(uint8 intSrc)* configures the TX interrupt sources to be enabled (but does not enable the interrupt).
- *void UART_WriteTxData(uint8 txDataByte)* places a byte of data into the transmit buffer to be sent when the bus is available without checking the TX status register. Status must be checked separately.

[46]Ibid.

©Copyright 2012 Cypress Semiconductor

- *uint8 UART_ReadTxStatus(void)* reads the status register for the TX portion of the UART.
- *void UART_PutChar(uint8 txDataByte)* places a byte of data into the transmit buffer to be sent when the bus is available. This is a blocking API that waits until the TX buffer has room to hold the data.
- *void UART_PutString(char* string)* sends a NULL terminated string to the TX buffer for transmission.
- *void UART_PutArray(uint8* string, uint8/uint16 byteCount)* places N bytes of data from a memory array into the TX buffer for transmission.
- *void UART_PutCRLF(uint8 txDataByte)* writes a byte of data followed by a carriage return (0x0D) and line feed (0x0A) to the transmit buffer.
- *uint8/uint16 UART_ GetTxBufferSize(void)* determines the number of bytes used in the TX buffer. An empty buffer returns 0.
- *void UART_ ClearTxBuffer(void)* clears all data from the TX buffer and hardware TX FIFO.
- *void UART_SendBreak(uint8 retMode)* transmits a break signal on the bus.
- *void UART_SetTxAddressMode(uint8 addressMode)* configures the transmitter to signal the next bytes is address, or data.
- *void UART_LoadRxConfig(void)*loads the receiver configuration in half duplex mode. After calling this function, the UART is ready to receive data.
- *void UART_LoadTxConfig(void)* loads the transmitter configuration in half duplex mode. After calling this function, the UART is ready to transmit data.
- *void UART_Sleep(void)* is the preferred API to prepare the component for sleep. The *UART_Sleep()* API saves the current component state. Then it calls the *UART_Stop()* function and calls *UART_ SaveConfig()* to save the hardware configuration. Call the *UART_Sleep()* function before calling the *CyPmSleep()* or the *CyPmHibernate()*function.
- *void UART_ Wakeup(void)* is the preferred API to restore the component to the state when *UART_Sleep()*was called. The *UART_ Wakeup()* function calls the *UART_RestoreConfig()* function to restore the configuration. If the component was enabled before the *UART_ Sleep()* function was called, the *UART_ Wakeup()* function will also re-enable the component.
- *void UART_Init(void)*initializes, or restores, the component according to the customizer *Configure* dialog settings. It is not necessary to call *UART_Init()* because the *UART_Start()* API calls this function and is the preferred method to begin component operation.
- *void UART_Enable(void)* activates the hardware and begins component operation. It is not necessary to call *UART_Enable()* because the *UART_Start()* API calls this function, which is the preferred method to begin component operation.
- *void UART_SaveConfig(void)* saves the component configuration and nonretention registers. It also saves the current component parameter values, as defined in the *Configure* dialog or as modified by appropriate APIs. This function is called by the UART_Sleep() function.
- *void UART_RestoreConfig(void)* restores the user configuration of nonretention registers.

4.3.2 Interrupts

The *Interrupt On* parameters allow the interrupt sources to be configured. These values are ORed with any of the other *Interrupt On* parameter to give a final group of events that can trigger an interrupt. The software can reconfigure these modes at any time, and these parameters define an initial configuration.

©Copyright 2012 Cypress Semiconductor

- RX - On Byte Received
  (UART_RX_STS_FIFO_NOTEMPTY)
- TX - On TX Complete
  (UART_TX_STS_COMPLETE)
- RX - On Parity Error
  (UART_RX_STS_PAR_ERROR)
- TX - On FIFO Empty
  (UART_TX_STS_FIFO_EMPTY)
- RX - On Stop Error
  (UART_RX_STS_STOP_ERROR)
- TX - On FIFO Full
  (UART_TX_STS_FIFO_FULL)
- RX - On Break
  (UART_RX_STS_BREAK)
- TX - On FIFO Not Full
  (UART_TX_STS_FIFO_NOT_FULL)
- RX - On Overrun Error
  (UART_RX_STS_OVERRUN)
- RX - On Address Match
  (UART_RX_STS_ADDR_MATCH)
- RX - On Address Detect
  (UART_RX_STS_MRKSPC)

An ISR can be handled by an external interrupt component connected to the *tx_interrupt* or *rx_interrupt* output. The interrupt output pin is visible depending on the selected *Mode* parameter. It outputs the same signal to the internal interrupt based on the selected status interrupts.

```
#include <device.h>

#define START_CHAR_VALUE    0x20
#define END_CHAR_VALUE      0x7E

uint8 trigger = 0;

void main()
{
    uint8 ch;           /* Data sent on the serial port */
    uint8 count = 0;    /* Initializing the count value */
    uint8 pos = 0;

    CyGlobalIntEnable;

    isr_1_Start();      /* Initializing the ISR */
    UART_1_Start();     /* Enabling the UART */

    for(ch = START_CHAR_VALUE; ch <= END_CHAR_VALUE; ch++)
```

©Copyright 2012 Cypress Semiconductor

```
    {
        UART_1_WriteTxData(ch); /* Sending the data */
        CyDelay(200);
    }

    for(;;) {}
}

void main()
{
    char8 ch;       /* Data received from the Serial port */

    CyGlobalIntEnable; /* Enable all interrupts by the processor. */

        UART_1_Start();

    while(1)
    {
        /* Check the UART status */
        ch = UART_1_GetChar();

        /* If byte received */
        if(ch > 0)
        {
           // Place character
           //handling code here
        }
    }
```

4.3.3 UART Config Tab

The Mode dialog box determines the mode of operation of the UART, e.g., as a bidirectional *Full UART (TX + RX)*[47], *Half Duplex UART*, requiring half of the resources, *RX Only* (RS232 Receiver) or *TX Only* (Transmitter). The *Bits Per Second* parameter determines the baud-rate, or bit-width, configuration of the hardware for clock generation.[48] The *Data bits* parameter determines the number of data bits transmitted between the start and stop of a single UART transaction.[49]

4.3.4 Parity

Parity refers to the appending of an extra bit to each byte for the purpose of detecting an error that occurs during serial byte transmission. The parity bit is set to one, if the number of 1 bits[50] is either even or odd, depending on the parity mode selected. Parity can be set as *Even*, *Odd, None* or *Mark/Space*. *None* implies that the ninth bit is not to be employed, i.e. parity is not used, Even/Odd implies that the number of 1s, exclusive of the parity bit, in the byte is

[47]This is the default mode.

[48]The default setting for bits per second is 57,600. If the internal clock is used, by setting the Clock Selection parameter, PSoC Creator generates the necessary clock for 57,600 bps.

[49]Options are 5,6,7,8, or 9 data bits. The default setting of 8 bits, results in a transmission of a single byte per transmission. The 9 bit setting utilizes a ninth bit as a parity bit as an indication of either even or odd parity of the eight bits.

[50]Exclusive of the parity bit setting.

©Copyright 2012 Cypress Semiconductor

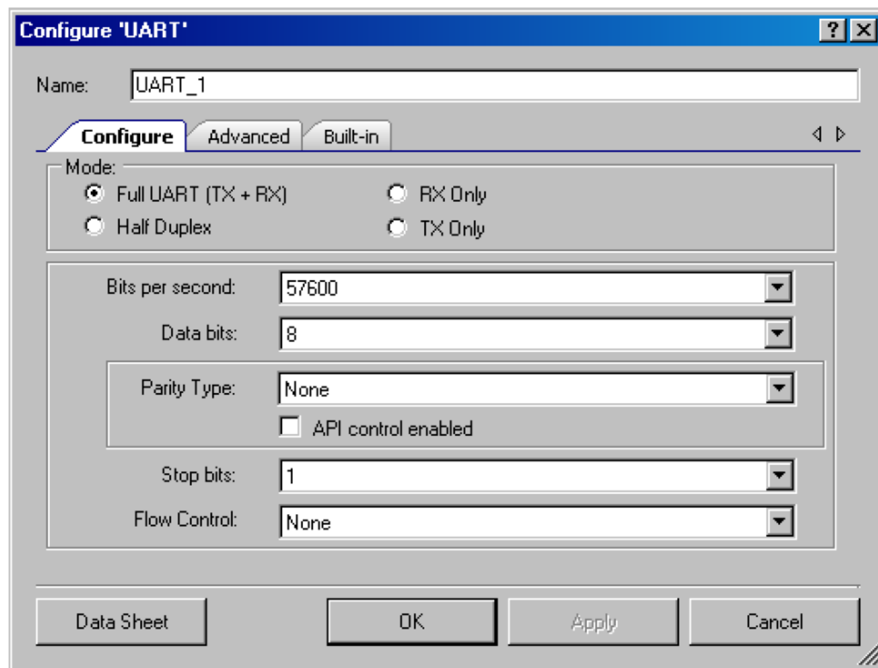


Figure 4.10: PSoC Creator's UART configuration tab.

even/odd. Upon receipt of each byte, the parity bit can be checked to determine if a change has occurred.[51]

4.3.5 Simplex, Half and Full Duplex

Serial transmission[52] can occur in various modes including simplex, half duplex and full duplex as illustrated in Figure 4.11. A duplex communications system allows communications between two points, in either direction, contemporaneously. Half-duplex systems also allow communications between two points, in both directions, but only in only direction at a time. Simplex systems allow communication between any two points in one direction only.

4.3.6 RS232, RS422 and RS485 Protocols

The serial communication protocols that has been most prevalent for applications employing UARTs have historically been RS232, RS422[53] and RS485[54]. The RS232 protocol is a point to point bidirectional communications link as opposed to RS485, a single channel bus. The RS232 signal path employs a single wire and symmetric voltages about a common ground The RS485

[51] Single bit errors are the most common type or errors that occur during byte transmission.
[52] Serial communication refers to the transfer of information, in a sequential fashion, from location to another.
[53] RS422 is a communication protocol based on differential data transmission which was originally intended to support higher data rates than RS232 and over longer distances. However, this protocol is not a true multidrop protocol in that it will only support one driver and a maximum of ten receivers. There is a four wire implementation of RS422 that supports multiple drivers but typically in a half duplex mode.
[54] RS485 is a true *multidrop system* in that it will support multiple drivers and receivers .

©Copyright 2012 Cypress Semiconductor

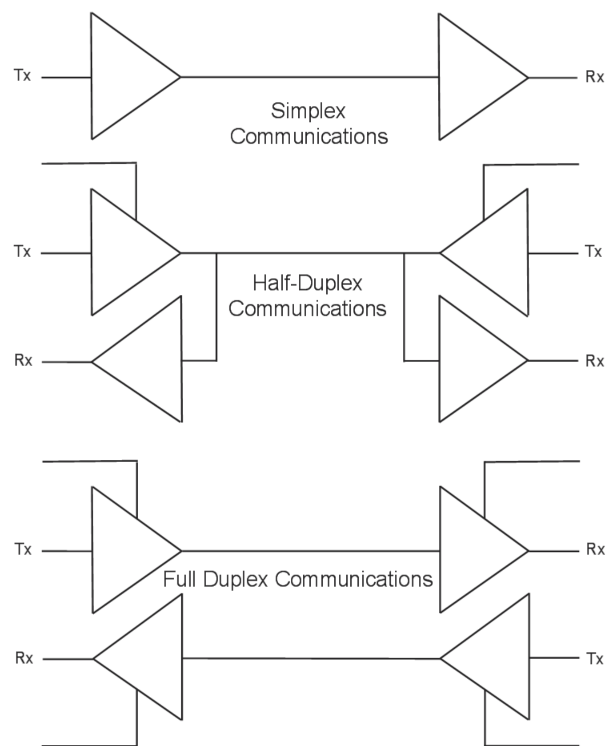


Figure 4.11: Simplex, Half and Full Duplex.

©Copyright 2012 Cypress Semiconductor

protocol is an EIA[55] standard interface that employs a balanced transmission path[56] and is able to communicate with multiple nodes. It is particularly useful when the communications is to occur over relatively long distances and can be employed at distances of up to 1200 meters and at rates up to 100 kbit/sec.

Table 4.5: Comparison of RS232 and RS485 protocols.

Protocol	RS232	RS485
Max Drivers/Receivers	1/1	32/32
Load Impedance	3k-7k	54
Mode	Single-ended	Differential
Max Data rate	115.200 kbaud	100 -3500 kbaud
Max cable length	15 meters	1200 meters
Max slew rate	30 V/sec	N/A
Max Driver Current	+/- 6 ma @	+/-100a
Output signal levels	+/- 5 to +/- 15 v	+/- 1.5 v
Supported Duplex Modes	Full and Half	Full and Half
Communication Type	Peer	Multi-point
Sync/Async	Async	Async

The RS232 and RS485 protocols are similar but their are some significant differences as shown in Table 4.5.

4.4 Serial Peripheral Interface (SPI)

The serial peripheral interface bus, or SPI, was developed by Motorola for intercommunications with relatively slow peripheral devices on an intermittent basis, e.g., transfer of data to to a microcontroller from an analog to digital converter. Although comparable to I2C in many respects, SPI is capable of higher data rates and its ability to operate in a full duplex mode.

PSoC3/5's Serial Peripheral Interface (SPI) component features include:

- 3- to 16-bit data width
- Four SPI operating modes
- Bit rates up to 9 Mbps[57]

4.4.1 SPI Device Configurations

PSoC Creator supports a number of configurations of SPI masters and slaves as shown in Figure 4.12.

[55]Electronic Industries Alliance.

[56]The balance path affords noise immunity making it possible for the receiver to reject common mode signals and shifts in the ground pathway.

[57]This value is valid only for MOSI+MISO (Full Duplex) interfacing mode and is restricted up to 1 Mbps in the bidirectional mode because of internal bidirectional pin constraints.

©Copyright 2012 Cypress Semiconductor

4.4.2 SPI Master

The *SPI Master* component provides an industry-standard, 4-wire, master SPI interface. It can also provide a 3-wire (bidirectional) SPI interface. Both interfaces support all four SPI operating modes, allowing communication with any SPI slave device. In addition to the standard 8-bit word length, the *SPI Master* supports a configurable 3- to 16-bit word length for communicating with nonstandard SPI word lengths. SPI signals include the standard *Serial Clock* (SCLK), *Master In Slave Out* (MISO), *Master Out Slave In* (MOSI), bidirectional *Serial Data* (SDAT), and *Slave Select* (SS). The *SPI Master* component can be used when the PSoC device must interface with one, or more, SPI slave devices. In addition to SPI slave labeled devices, the *SPI Master* can be used with many devices implementing a shift-register-type serial interface. The *SPI Slave* component should be used in instances in which the PSoC device must communicate with an SPI master device. The *Shift Register* component can be used in situations for which its low-level flexibility provides hardware capabilities not available in the *SPI Master* component.

4.4.3 SPI I/O

PSoC Creator's SPI Master component can be configured using the following:

- *void SPIM_Start(void)* calls both *SPIM_Init()* and *SPIM_Enable()*.[58]
- *void SPIM_Stop(void)* disables *SPI Master* operation by disabling the internal clock and internal interrupts, if the *SPI Master* is configured that way.
- *void SPIM_Start(void)* calls both *SPIM_Init()*and *SPIM_Enable()* and should be called the first time the component is started.
- *void SPIM_Stop(void)* disables *SPI Master* operation by disabling the internal clock and internal interrupts.
- *void SPIM_EnableTxInt(void)* enables the internal Tx interrupt irq.
- *void SPIM_EnableRxInt(void)* enables the internal Rx interrupt irq.
- *void SPIM_DisableTxInt(void)* disables the internal Tx interrupt irq.
- *void SPIM_DisableRxInt(void)* disables the internal Rx interrupt irq.
- *void SPIM_SetTxInterruptMode(uint8 intSrc)* configures which status bits trigger an interrupt event.
- *void SPIM_SetRxInterruptMode(uint8 intSrc)* configures which status bits trigger an interrupt event.
- *uint8 SPIM_ReadTxStatus(void)* returns the current state of the Tx status register.
- *uint8 SPIM_ReadRxStatus(void)* returns the current state of the Rx status register.
- *void SPIM_WriteTxData(uint8/uint16 txData)* places a byte/word in the transmit buffer to be sent at the next available SPI bus time. Data may be placed in the memory buffer and will not be transmitted until all other previous data has been transmitted. This function is blocked until there is space in the output memory buffer. It also clears the Tx status register of the component.
- *uint8/uint16 SPIM_ReadRxData(void)*[59] returns the next byte/word of received data available in the receive buffer. It returns invalid data if the FIFO is empty.

  Call *SPIM_GetRxBufferSize()*, and if it returns a nonzero value then it is safe to call the *SPIM_ReadRxData()* function.

[58]This should be called the first time the component is started.
[59]This function returns invalid data if the FIFO is empty.

©Copyright 2012 Cypress Semiconductor

- *uint8 SPIM_ GetRxBufferSize(void)* returns the number of bytes/words of received data currently held in the Rx buffer.
  - If the Rx software buffer is disabled, this function returns 0 = FIFO empty or 1 = FIFO not empty.
  - If the Rx software buffer is enabled, this function returns the size of data in the Rx software buffer. FIFO data not included in this count .
- *uint8 SPIM_ GetTxBufferSize(void)* returns the number of bytes/words of data ready to transmit currently held in the Tx buffer.
  - If Tx software buffer is disabled, this function returns 0 = FIFO empty, 1 = FIFO not full, or 4 = FIFO full.
  - If the Tx software buffer is enabled, this function returns the size of data in the Tx software buffer.[60]
- *void SPIM_ ClearRxBuffer(void)* clears the Rx buffer memory array and Rx hardware FIFO of all received data. It clears the Rx RAM buffer by setting both the read and write pointers to zero. Setting the pointers to zero indicates that there is no data to read. Thus, writing resumes at address 0, overwriting any data that may have remained in the RAM.
- *void SPIM_ ClearTxBuffer(void)* clears the Tx buffer memory array of data waiting to transmit. It clears the Tx RAM buffer by setting both the read and write pointers to zero. Setting the pointers to zero indicates that there is no data to transmit. Thus, writing resumes at address 0, overwriting any data that may have remained in the RAM.
- *void SPIM_ TxEnable(void)* sets the bidirectional pin to transmit, if the SPI Master is configured to use a single bidirectional pin.
- *void SPIM_ TxDisable(void)* sets the bidirectional pin to receive, if the SPI master is configured to use a single bidirectional pin.
- void SPIM_PutArray(uint8/uint16 * buffer, uint8/uint16 byteCount) places an array of data into the transmit buffer.
- *void SPIM_ ClearFIFO(void)* clears any received data from the Tx and Rx FIFOs.
- *void SPIM_ Sleep(void)* prepares the SPI Master for low-power modes by calling the *SPIM_ SaveConfig()*and *SPIM_ Stop()* functions.
- *void SPIM_ Wakeup(void)* prepares the SPI Master to wake up from a low-power mode and calls the SPIM_RestoreConfig() and SPIM_Enable() functions. Also clears all data from the Rx buffer, Tx buffer, and hardware FIFOs.
- *void SPIM_ Init(void)* initializes, or restores, the component according to the customizer *Configure* dialog settings. It is not necessary to call *SPIM_ Init()* because the *SPIM_ Start()* routine calls this function and is the preferred method to begin component operation.
- *void SPIM_ Enable(void)* enables the SPI Master for operation. Starts the internal clock, if the SPI Master is configured that way. If it is configured for an external clock, it must be started separately before calling this function. The *SPIM_ Enable()* function should be called before SPI Master interrupts are enabled. This is because this function configures the interrupt sources and clears any pending interrupts from device configuration, and then enables the internal interrupts if there are any. A *SPIM_ Init()* function must have been previously called.
- *void SPIM_ SaveConfig(void)* saves the SPI Master hardware configuration before entering a low-power mode.
- *void SPIM_ RestoreConfig(void)* restores the SPI Master hardware configuration saved by the *SPIM_ SaveConfig()* function after waking from a lower-power mode.

[60]The FIFO data not included in this count.

©Copyright 2012 Cypress Semiconductor

4.4.4 Tx Status Register

The Tx status register is a read-only register that contains the various transmit status bits defined for a given instance of the SPI Master component. Assuming that an instance of the SPI Master is named *SPIM*, the value of this register can be obtained by using the *SPIM_ReadTxStatus()* function. The interrupt output signal is generated by ORing the masked bit fields within the Tx status register. The mask can be set by using the *SPIM_SetTxInterruptMode()* function. Upon receiving an interrupt, the interrupt source can be retrieved by reading the Tx status register with the*SPIM_ReadTxStatus()*function. Sticky bits in the Tx status register are cleared on reading, so the interrupt source is held until the *SPIM_ReadTxStatus()* function is called.

All operations on the Tx status register must use the following defines for the bit fields, because these bit fields may be moved within the Tx status register at build time. Sticky bits used to generate an interrupt or DMA transaction must be cleared with either a CPU or DMA read to avoid continuously generating the interrupt or DMA. There are several bit fields defined for the Tx status registers. Any combination of these bit fields may be included as an interrupt source. The bit fields indicated with an asterisk (*) in the following list are configured as sticky bits in the Tx status register. All other bits are configured as real-time indicators of status. Sticky bits latch a momentary state so that they may be read at a later time and cleared on read.

The following # defines are available in the generated header file (for example, SPIM.h):

- *SPIM_STS_SPI_DONE* * Set high as the data-latching edge of SCLK (edge is mode dependent) is output. This happens after the last bit of the configured number of bits in a single SPI word is output onto the MOSI line and the transmit FIFO is empty. Cleared when the SPI Master is transmitting data or the transmit FIFO has pending data. Tells you when the SPI Master is complete with a multi-word transaction.
- *SPIM_STS_TX_FIFO_EMPTY* reads high while the transmit FIFO contains no data pending transmission. Reads low if data is waiting for transmission.
- *SPIM_STS_TX_FIFO_NOT_FULL* reads high while the transmit FIFO is not full and has room to write more data. Reads low if the FIFO is full of data pending transmit and there is no room for more writes at this time. Tells you when it is safe to pend more data into the transmit FIFO.
- SPIM_STS_BYTE_COMPLETE * set high as the last bit of the configured number of bits in a single SPI word is output onto the MOSI line. Cleared* as the data latching edge of SCLK (edge is mode dependent) is output.
- SPIM_STS_SPI_IDLE * is set high as long as the component state machine is in the SPI IDLE state (component is waiting for Tx data and is not transmitting any data).

4.4.5 RX Status Register

The Rx status register is a read-only register that contains the various receive status bits defined for the SPI Master. The value of this register can be obtained by using the *SPIM_ReadRxStatus()* function. The interrupt output signal is generated by ORing the masked bit fields within the Rx status register. The mask can be set by using the *SPIM_SetRxInterruptMode()* function. Upon receiving an interrupt, the interrupt source can be retrieved by reading the Rx status register with the *SPIM_ReadRxStatus()* function. Sticky bits in the Rx status register are cleared on reading, so the interrupt source is held until the *SPIM_ReadRxStatus()* function is called. All operations on the Rx status register must use the following defines for the bit fields, because

©Copyright 2012 Cypress Semiconductor

these bit fields may be moved within the Rx status register at build time. Sticky bits used to generate an interrupt or DMA transaction must be cleared with either a CPU or DMA read to avoid continuously generating the interrupt or DMA. There are several bit fields defined for the Rx status register. Any combination of these bit fields can be included as an interrupt source. The bit fields indicated with an asterisk (*) in the following list are configured as sticky bits in the Rx status register. All other bits are configured as real-time indicators of status. Sticky bits latch a momentary state so that they may be read at a later time and cleared when read. The following #defines are available in the generated header file (for example, SPIM.h):

- *SPIM_STS_SPI_DONE* * set high as the data-latching edge of SCLK (edge is mode dependent) is output. This happens after the last bit of the configured number of bits in a single SPI word is output onto the MOSI line and the transmit FIFO is empty. Cleared when the SPI Master is transmitting data or the transmit FIFO has pending data. Tells you when the SPI Master is complete with a multi-word transaction.
- *SPIM_STS_TX_FIFO_EMPTY* reads high while the transmit FIFO contains no data pending transmission. Reads low if data is waiting for transmission.
- *SPIM_STS_TX_FIFO_NOT_FULL* reads high while the transmit FIFO is not full and has room to write more data. Reads low if the FIFO is full of data pending transmit and there is no room for more writes at this time. Indicates when it is safe to send more data to the transmit FIFO.
- *SPIM_STS_BYTE_COMPLETE* * set high as the last bit of the configured number of bits in a single SPI word is output onto the MOSI line. Cleared* as the data latching edge of SCLK (edge is mode dependent) is output.
- *SPIM_STS_SPI_IDLE* * this bit is set high as long as the component state machine is in the SPI IDLE state (component is waiting for Tx data and is not transmitting any data).

4.4.6 Tx Data Register

The Tx data register contains the transmit data value to send and is implemented as a FIFO in the SPI Master. There is an optional higher-level software state machine that controls data from the transmit memory buffer. It handles large amounts of data to be sent that exceed the FIFO's capacity. All APIs that involve transmitting data must go through this register to place the data onto the bus. If there is data in this register and the control state machine indicates that data can be sent, then the data is transmitted on the bus. As soon as this register (FIFO) is empty, no more data will be transmitted on the bus until it is added to the FIFO. DMA can be set up to fill this FIFO when empty, using the *TXDATA_REG* address defined in the header file.

4.4.7 Rx Data Register

The Rx data register contains the received data and is implemented as a FIFO in the SPI Master. There is an optional higher-level software state machine that controls data movement from this receive FIFO into the memory buffer. Typically, the Rx interrupt indicates that data has been received. At that time, that data has several routes to the firmware. DMA can be set up from this register to the memory array, or the firmware can simply call the *SPIM_ReadRxData()* function. DMA must use the *RXDATA_REG* address defined in the header file.

4.4.8 Conditional Compilation Information

The SPI Master requires only one conditional compile definition to handle the 8- or 16-bit datapath configuration necessary to implement the configured *NumberOfDataBits*. The API must

©Copyright 2012 Cypress Semiconductor

conditionally compile for the data width defined. APIs should never use these parameters directly but should use the following define:

- *SPIM_DATAWIDTH* defines how many data bits will make up a single-byte transfer. Valid range is 3 to 16 bits.

4.5 Serial Peripheral Interface Slave

PSoC Creator's SPI Slave provides an industry-standard, 4-wire slave SPI interface capable of providing a 3-wire, bidirectional, SPI interface. Both interfaces support all four SPI operating modes, allowing communication with any SPI master device. In addition to the standard 8-bit word length, the SPI Slave supports a configurable 3- to 16-bit word length for communicating with nonstandard SPI word lengths. SPI signals include the standard *Serial Clock* (SCLK), *Master In Slave Out* (MISO), *Master Out Slave In* (MOSI), *bidirectional Serial Data* (SDAT), and *Slave Select* (SS). The SPI Slave component can be used any time a PSoC device is required to interface with an SPI Master device. In addition to use with SPI Master devices, the SPI Slave can be used with devices implementing a shift register interface. The SPI Master component can be employed in applications requiring that a PSoC device to communicate with an SPI Slave device.

By default, the *PSoC Creator Component Catalog* contains *Schematic Macro* implementations for the SPI Slave component. These macros contain already connected and adjusted input and output pins and clock source. Schematic Macros are available for 3-wire (Bidirectional), 4-wire (Full Duplex) and Full Duplex Multislave SPI interfacing as shown in Figures 4.12 e), f) and g), respectively.

4.5.1 Slave I/O Connections

The slave I/O connections supported by PSoC Creator are as follows:[61]

- *mosi - Input** The *Master Output Slave Input* (MOSI) signal from a master device is applied to the *mosi input*. This input is visible when the *Data Lines* parameter is set to *MOSI + MISO*. If visible, this input must be connected.
- *sdat - Inout** The *Serial Data* (SDAT) signal is applied to the *sdat inout* input which is used when the Data Lines parameter is set to Bidirectional. For both PSoC 3 and PSoC 5 silicon, an *asynchronous clock crossing* warning will be reported between the component clock and the SCLK signal when timing analysis is performed. The following is an example of such a message: Path(s) exist between clocks *IntClock* and *SCLK(0)_PAD*, but the clocks are not synchronous to each other. This message applies to a path from the register that controls the direction and the sampling of data by SCLK. SCLK should not be running when the direction is being changed. As long as this rule is followed, there is no problem and warning message can be ignored.
- *sclk Input* - The *Serial Clock* (SCLK) signal is applied to the sclk input which provides the slave synchronization clock input to the device. This input is always visible and must be connected.[62]

[61] An asterisk (*) in the list of I/Os indicates that the I/O may be hidden for the component symbol under the conditions listed in the description of that I/O.

[62] Some SPI Master devices, e.g., the TotalPhase Aardvark I2C/SPI host adapter, drive the *sclk* output in a specific way. For the *SPI Slave* component to function properly with such devices in modes 1 and 3, when CPOL = 1), the sclk pin should be set to *resistive pull-up drive mode*. Otherwise, corrupted data is output.

©Copyright 2012 Cypress Semiconductor

- *ss Input* - The Slave Select (SS) signal to the device is applied to the *ss input*. This input is always visible and must be connected. The following diagrams show the timing correlation between SCLK and SS signals Generally, 0.5 of the SCLK period is enough delay between the SS negative edge and the first SCLK edge for the SPI Slave to work correctly in all supported bit-rate ranges.
- *reset Input* - resets the SPI Slave and deletes any data that was currently being transmitted, or received. However, it does not clear data from the FIFO that has already been received or is ready to be transmitted. PSoC3/5 ES2 silicon does not support this reset functionality, so this input is ignored when used with those devices. Use of the reset input results in an asynchronous clock crossing warning being reported between the clock that generates the *Reset input* and the SCLK signal when timing analysis is performed. The following is an example of such a message: Path(s) exist between clocks *BUS_ CLK* and *SCLK(0)_ PAD*, but the clocks are not synchronous to each other.This message applies to a path from the Reset signal to the operation of the SPI component clocked by SCLK. SCLK should not be running when the Reset signal is changed. As long as this rule is followed, there is no problem and you can ignore this message. The reset input may be left floating with no external connection. If nothing is connected to the reset line the component will assign it a constant logic 0.
- *clock - Input** defines the sampling rate of the status register. All data clocking happens on the *sclk* input, so the clock input does not handle the bit-rate of the SPI Slave. The clock input is visible when the Clock Selection parameter is set to External. If visible, this input must be connected.
- *miso - Output** transmits the *Master In Slave Out* (MISO) signal to the master device on the bus. This output is visible when the Data Lines parameter is set to *MOSI + MISO*.
- *interrupt - Output* is the logical OR of the group of possible interrupt sources. This signal goes high while any of the enabled interrupt sources are true.

The PSoC Creator Component Catalog contains *Schematic Macro* implementations for the SPI Slave component that have connected and adjusted input pins, output pins and a clock source. As shown in Figure 4.12 d) e), f) and g), Schematic Macros are available for 4-wire (Full Duplex), 3-wire (Bidirectional), and Full Duplex Multislave SPI interfacing.[63]

[63]If schematic macros are not used the *Pins* component should be configured to deselect the *Input Sychronized* parameter for each of the assigned input pins, i.e., MOSI, SCLK and SS. This parameter is located beneath the Pins>Input tab of the applicable *Pins Config* dialog.

©Copyright 2012 Cypress Semiconductor

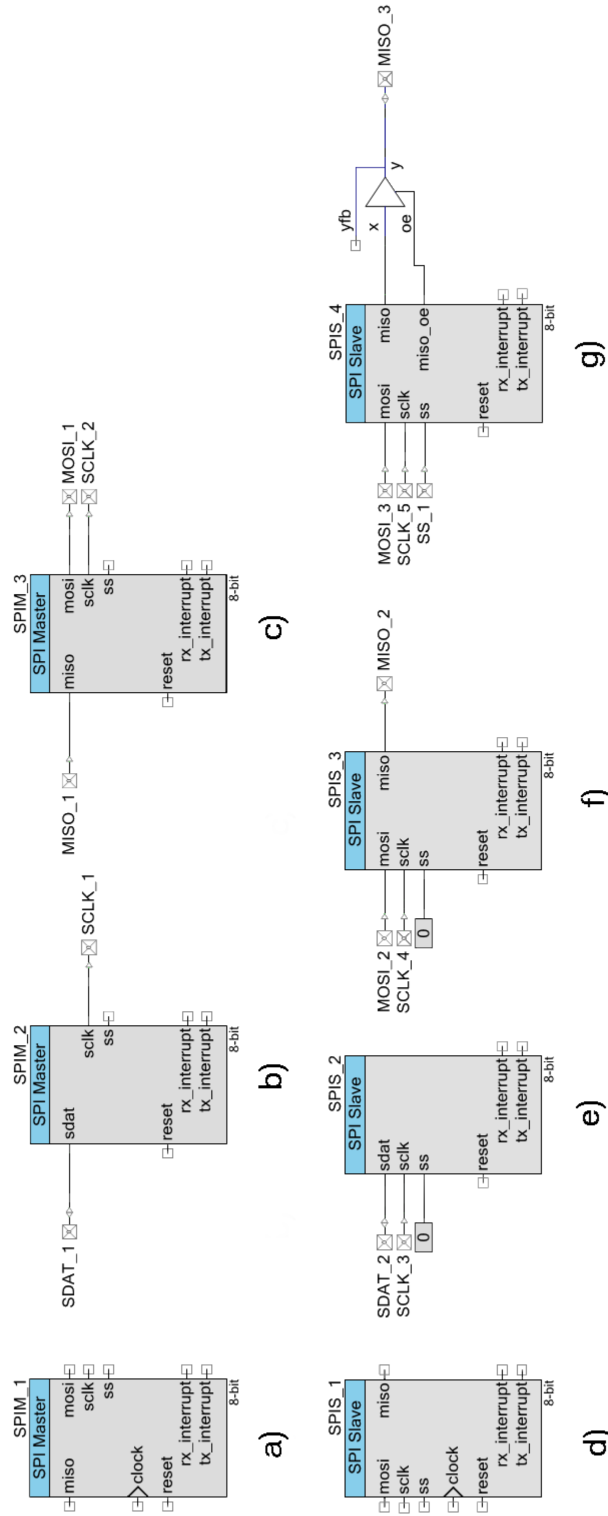


Figure 4.12: SPI master and slave Schematic Macros supported by PSoC Creator.

©Copyright 2012 Cypress Semiconductor

4.6 Universal Serial Bus (USB) Basics

The Universal Serial Bus (USB) is an industry standard[64], serial communications protocol originally designed for communications between computers and peripheral devices such as mice, keyboards modems, external hard drives, etc., as an alternative to larger, and slower, connections that employ serial and parallel ports.[50] In addition to providing faster transfer rates, the intent was to eliminate the various connector configurations used by the different protocols and standardize on a single, physical configuration, connection device. Originally, version 1.0 supported two configurations referred to as low-speed (LS) and full-speed (FS) at 1.5 and 12 Mbits/second. The LS configuration, while significantly slower than it FS counterpart, is much less susceptible to electromagnetic interference. Version 2.0 introduced a higher speed (HS) configuration as part of the specification that supported transmission rates of 480 Mbits/second.

A typical USB application includes a personal computer which serves as a host an several peripheral devices employed as part of a tiered, star topology which can include hubs that provide multiple connection points. The host utilizes at least one host controller and a root hub. Each host controller can support up to 127 connections, inclusive, when used with external USB hubs. The internal root hub is connected to the host controller(s) and provides the first interface layer to the USB. Most PCs are provided with multiple USB ports that are part of the root hub in the PC.

The host controller consists of a hardware chipset and software driver layer that

- detects the attachment/removal of USB devices
- manages data flow between the host an such devices
- supplies power to the USB connected devices

and,

- monitors the USB bus activity.

Each USB device is assigned an address by the host, a connection pathway referred to as a *pipe* that connects the host and an addressable buffer known as an *endpoint.* The endpoint serves as an addressable buffer that holds data to be transmitted to the host, or that has been received from the host. A USB device can have multiple endpoints each of which has an associated pipe, as illustrated in Figure 4.13.

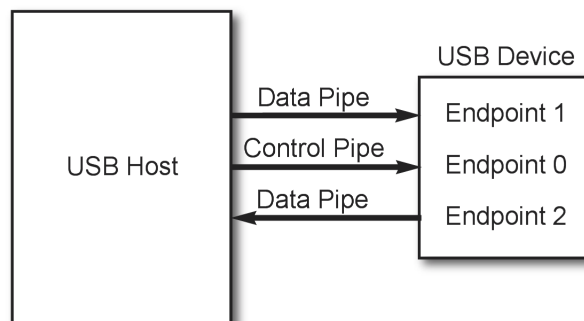


Figure 4.13: USB pipe model.

The USB specification defines four types of of data transfer categories:

[64] Compaq, DEC, IBM, Intel, Motorola NEC and Nortel collaborated on the development of the specification for the universal serial bus

©Copyright 2012 Cypress Semiconductor

- *control transfers* are used for sending commands to a device, to make inquiries and configure a device via the control pipe. Bulk transfers for large data transmissions that exploit all of the available USB bandwidth using a data pipe.[65]
- *interrupt transfers* are used for sending small amounts of *bursty* data and to provide a guaranteed minimum latency.
- *isochronous transfers* are used for data that must be transferred at a guaranteed data rate which is based on a fixed bus bandwidth, fixed latency and no error correction.[66]

Every device has a control pipe through which transfers to send, and receive, messages are transmitted. Optionally, a device may have data pipes for transferring data through interrupt, bulk, or isochronous transfers, but he control pipe is the only bidirectional pipe in the USB system. All the data pipes are unidirectional. Each endpoint is accessed with a device address, assigned by the host, and an endpoint number, assigned by the device. When information is sent to the device the device address and endpoint number are identified with a token packet. The host initiates this token packet before a data transaction. When a USB device is first connected to a host, the USB enumeration process is initiated.

Two files, on the host side, are affiliated with enumeration and the loading of a driver:

- .INF is text file that contains all the information necessary to install a device, e.g., driver names and locations, windows registry, and driver version information.
- .SYS is the driver needed to communicate effectively with the USB device.

Enumeration is the process of exchanging information between the device and the host that includes learning about the device. Additionally, enumeration includes assigning an address to the device, reading descriptors[67], and assigning and loading a device driver a process that can occur in seconds. Once this process is complete, the device is ready to transfer data to the host.

The flow chart of the general enumeration process as shown is Figure 4.14, are:

1. the device is connected to host,
2. the host resets the device and requests a device descriptor,
3. the device responds to the request and the host sets a new address,
4. the host requests a device descriptor using the new address,
5. the host locates and reads the INF file,
6. the INF file specifies the device driver,
7. the driver is loaded on the host,

and

8. the device is configured and ready to use.

After a device has been enumerated, the host directs all traffic flow, to the devices, on the bus and therefore no device can transfer data without a request from the host controller.

[65]Bulk transfers cannot be relied on top take place with a specific speed or latency.
[66]Error correction can introduce variable delays caused by having to delay transmission while compromised packets are resent.
[67]Descriptors are data structures that provide information about the device.

©Copyright 2012 Cypress Semiconductor

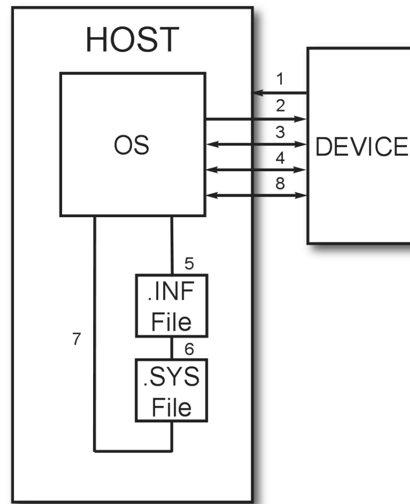


Figure 4.14: Sequence of enumeration events.

4.6.1 USB Architecture

Only one host can exist in the system and communication with devices is from the host's perspective. A host is an*upstream component*, while a device is a *downstream component*. Data moved from the host to the peripheral is an *OUT transfer*. Data moved to the host from the peripheral is an *IN transfer*. The host, specifically the host controller, controls all traffic and issues commands to devices.

There are three common types of USB host controllers:

- *Universal Host Controller Interface* (UHCI) (UHCI): Produced by Intel for USB 1.0 and USB 1.1. Using UHCI requires a license from Intel. This controller supports both low-speed and full-speed.
- *Open Host Controller Interface* (OHCI): Produced for USB 1.0 and 1.1 by Compaq, Microsoft, and National Semiconductor. Supports low-speed and full-speed and tends to be more efficient then UHCI by performing more functionality in hardware.
- *Extended Host Controller Interface (EHCI)*: Created for USB 2.0 after USB-IF requested that a single host controller specification be created. EHCI is used for high-speed transactions and delegates low-speed and full-speed transactions to an OHCI or UHCI sister controller.

One or more devices are attached to a host. Each device has a unique address and responds only to host commands that are addressed to it. Every is expected to have some form of functionality and not simply be passive. Devices contain one upstream port which serves as the physical USB connection point on the device. A hub is a specialized device that allows the host to communicate with multiple peripheral devices on the bus. Unlike USB peripheral devices, such as a mouse that has actual functionality, a hub device is transparent and is intended to act as a pass-through. A hub also acts as a channel between the host and the device. Hubs have additional attachment points to allow the connection of multiple devices to a single host. A hub repeats traffic to and from downstream devices through one upstream port and up to seven downstream ports. The hub, however, does not have any host capabilities.

As discussed previously, up to 127 devices can be connected to the host controller with the

©Copyright 2012 Cypress Semiconductor

use of hubs. This limitation is based on the USB protocol, which limits the device address to 7 bits. Additionally, a maximum of 5 hubs can be chained together, a limitation imposed by timing considerations. The USB interface can be viewed as being divided into different layers. The *Bus Interface Layer* provides the physical connection, electrical signaling, and packet connectivity. This is the layer that is handled by the hardware in a device. This is accomplished by a physical interface external to the device. The *Device Layer* is used by the USB system software for performing USB operations such as sending and receiving information. This is accomplished with a *Serial Interface Engine*, which is also internal to the device. Finally, the *Function Layer* is the software portion of a USB device that handles the information it receives and gathers data to transfer to the host.

4.6.2 USB Signal Paths

All signals involve a return path, often referred to as the *ground return*.[68] Although ground is typically assumed to be at a potential of zero volts, it is really a reference that may deviate from zero volts as a result of electromagnetic interference, the impedance of the return path, i.e., the ground path, and other phenomenon. In the case of long signal paths, there can be a significant difference between the ground at the transmitter (source) and the receiver (sink).

A USB cable consists of multiple conductors that are protected by an insulating jacket. Within this jacket is an outer shield consisting of copper braid. Inside this copper shield are multiple wires: a copper drain wire, a VBUS wire. (red) and a ground wire (black). An inner shield made of aluminum contains a twisted pair of data wires as seen in Figure 7. There is a D+ wire (green) and a D- wire (white). In full-speed and high-speed devices, the maximum cable length is 5-meters. To increase the distance between the host and a device, a series of hubs and 5-meter cables must be used. While USB extension cables are available, using them to exceed 5 meters is not in compliance with the USB protocol. Low-speed devices have slightly different specifications, e.g., their cable length is limited to 3 meters and low-speed cables are not required to be a twisted pair, an example of which is shown in Figure 4.15.

The VBUS wire provides a constant 4.40 - 5.25 V supply to all attached devices. While USB supplies up to 5.25 V to devices, the data lines (D+ and D-) function at 3.3 V. The USB interface uses a differential transmission that is non-return-to-zero inverted (NRZI) encoded with bit stuffing across a twisted pair of conductors.

Figure 4.15: An example of a twisted pair cable.

NRZI encoding is a method for mapping a binary transmission signal in which a logic 1 is represented by *no change* in voltage level and a logic 0 is represented by a *change* in voltage level as Figure 4.16 shows. The data that will be transmitted over USB is shown at the top of the figure. The encoded NRZI data is shown in the lower portion of the figure. The bit stuffing occurs by inserting a logic 0 into the data stream, following seven consecutive logic 1s. The purpose of the bit stuffing is for synchronization of the USB hardware by using a phase-locked loop (PLL). If there are too many logic 1s in the data, then there may not be enough transitions in the NRZI encoded stream to support synchronization. The USB receiver hardware automatically detects this extra bit and disregards it. However, this extra bit stuffing contributes additional USB overhead. Figure 4.16 shows an example of NRZI data with bit stuffing. Although there

[68] In some cases the return path is simply a ground plane.

©Copyright 2012 Cypress Semiconductor

are eight 1s in the *Data to Send* stream, in the encoded data a logic 0 is inserted after the sixth logic 1. The seventh and eighth logic 1 then follow after the 0 logic bit.

The hardware in USB devices handles all the encoding and bit stuffing upon receiving any data and prior to transmitting any data. The use of differential D+ and D- signals rejects common-mode noise. If noise becomes coupled into the cable, it will normally be present on all wires in the cable. With the use of a differential amplifier in the USB hardware internal to the host and device, the common-mode noise can be rejected, as illustrated in Figures 4.17 and 4.18. It should be noted that in the *Data to Send* stream, shown in Figure 4.16, there are eight 1s.

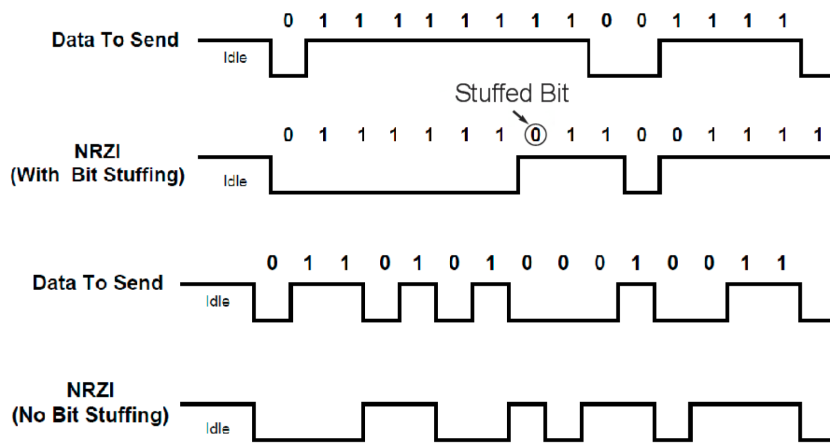


Figure 4.16: A bit stuffing example.

In the encoded data, after the sixth logic 1, a logic 0 is inserted. The seventh and eighth logic 1 then follow after this logic 0. The hardware in USB devices handles all of the encoding and bit stuffing upon receiving any data, and prior to transmitting any data. The reason for using the differential D+ and D- signal is for rejecting common-mode noise. If noise becomes coupled into the cable, it will normally be present on all wires in the cable. With the use of a differential amplifier in the USB hardware internal to the host and device, the common-mode noise can be rejected.

The hardware in USB devices handle all of the encoding and bit stuffing upon receiving any data and prior to transmitting any data. The reason for using the differential D+ and D- signal is for rejecting common-mode noise. If noise becomes coupled into the cable, it will normally be present on all wires in the cable. With the use of a differential amplifier in the USB hardware internal to the host and device, the common-mode noise can be rejected as shown in Figure 4.18.

USB communication occurs through many different signaling states on the D+ and D- lines. Some of these states transmit the data while others are used as specific signaling conditions. These states are described below with a quick reference list located in Table 4.6 .

Differential 0 and Differential 1: These two states are used in the general data communication across a USB communication path. Differential 1 is when the D+ line is high and the D- line is low. Differential 0 occurs when the D+ line is low, and the D- line is high.

J-State and K-State: In addition to the differential signals, the USB specification defines two additional differential states: J-States and K-States. Their definitions depend on the device speed. On a full-speed and high-speed device, a J-State is a Differential 1 and a K-State is a Differential 0. The opposite is true for a low-speed device.

©Copyright 2012 Cypress Semiconductor

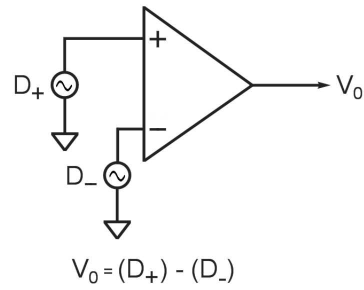


Figure 4.17: An ideal differential amplifier configuration

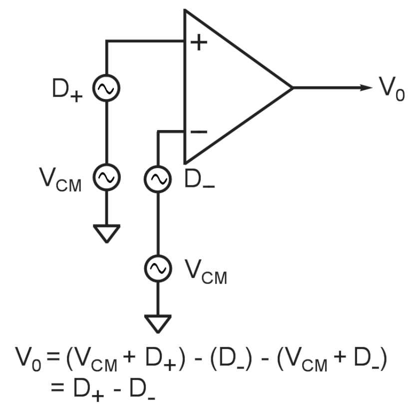


Figure 4.18: An example of USB common mode rejection.

©Copyright 2012 Cypress Semiconductor

Single Ended Zero (SE0) is a condition that occurs when both D+ and D- are driven low, indicating a reset, disconnect, or End of Packet.

Single Ended One (SE1): Condition that occurs when D+ and D- are both driven high. This condition does not ever occur intentionally and should never occur in a USB design.

Idle is a condition that occurs before and after a packet is sent. An Idle condition is signified by one of the data lines being low and the other line being high. The definition of high vs. low depends on device speed for a full-speed device, an idle condition consists of D+ being high and D- being low. The opposite is true for a low-speed device.

Resume is used to wake a device from a suspend state, by issuing a K-State.

Start of Packet (SOP) occurs before the start of any low-speed or full-speed packet when the D+ and D- lines transition from an idle state to a K-State.

End of Packet (EOP) occurs at the end of any low-speed or full-speed packet. An EOP occurs when an SE0 state occurs for 2 bit times, followed by a J-State for 1 bit time.

Reset occurs when an SE0 state lasts for 10 ms. The device can recognize the reset and begin to enter a reset after a SE0 has occurred for at least 2.5 ms, .

Keep Alive is a signal used in low-speed devices that lack a Start-of-Frame packet that is required to prevent suspend and use an EOP every millisecond to keep the device from entering suspend.

Table 4.6: USB Communications States

Bus State	Indication
Differential 1	D+ High, D- Low
Differential 0	D+ High, D- Low
Single Ended 0 (SE0)	D+ High, D- Low
Single Ended 1 (SE1)	D+ High, D- Low
J-State: Low Speed Full Speed High Speed	 Differential 0 Differential 1 Differential 1
K-State: Low Speed Full Speed High Speed	 Differential 1 Differential 0 Differential 0
Resume State	K-State
Start of Packet (SOP)	Data lines switch from idle to K-state
End of Packet (EOP)	SE0 for 2 bit times followed by J-statefor 1 bit time

©Copyright 2012 Cypress Semiconductor

4.6.3 USB Endpoints

In the USB specification, a *device endpoint* is a uniquely addressable portion of a USB device that is the source, or sink, of information in a communication flow between the host and device. The USB enumeration section describes a step in which the device responds to the default address. This occurs before other descriptor information such as the endpoint descriptors are read by the host, later in the enumeration process. During the enumeration sequence, a special set of endpoints are used for communication with the device. These special endpoints, collectively known as the *Control Endpoint* or *Endpoint 0*, are defined as *Endpoint 0 IN* and *Endpoint 0 OUT*. Even though *Endpoint 0 IN* and *Endpoint 0 OUT* are two endpoints, they look and act like one endpoint to the developer. Every USB device must support *Endpoint 0*. For this reason, *Endpoint 0* does not require a separate descriptor.

In addition to *Endpoint 0*, the number of endpoints supported in any particular device is based on its design requirements. A fairly simple design, such as a mouse, may need only a single IN endpoint. More complex designs may need several data endpoints. The USB specification sets a limit on the number of endpoints to 16 for each direction (16 IN/16 OUT = 32 Total) for high and full-speed devices, which does not include the control endpoints *0 IN* and *0 OUT*. Low-speed devices are limited to two endpoints. USB Class devices may set a greater limit on the number of endpoints, e.g., a low-speed HID design may have no more than two data endpoints, typically one IN endpoint and one OUT endpoint. Data endpoints are bidirectional by nature, but it is not until they are configured that they become unidirectional. Endpoint 1, for example, can be either an IN or OUT endpoint. It is in the device descriptors that Endpoint 1 becomes a IN endpoint.

Endpoints use cyclic redundancy checks (CRCs) to detect errors in transactions.[69] The handling of these calculations is taken care of by the USB hardware so that the proper response can be issued. The recipient of a transaction checks the transmitted CRC value against the CRC calculated by the receiver based on the received data. If the two match, then the receiver issues an ACK. If the data and the CRC do not match, then no handshake is sent. This absence of a handshake tells the transmitter to try again.

The USB specification further defines four types of endpoints and sets the maximum packet size, based on both the type and the supported device speed. The endpoint descriptor should be used to identify the type of endpoint requirements.

The four types of *endpoints* and characteristics are:

- *Control Endpoints* support control transfers, which all devices must support. Control transfers send, and receive, device information across the bus. The primary advantages of control transfers are guaranteed accuracy, proper detection of Errors and assurance that the data is resent. Control transfers have a 10% reserved bandwidth on the bus in low and full-speed devices (20% at high-speed) and give the USB system level control.
- *Interrupt Endpoints* support interrupt transfers which are used on devices that require a highly reliable method to communicate a small amount of data.[70] However, the name of this transfer can be misleading because it is not truly an interrupt based system, but instead employs a polling method. However, is does guarantee that the host check for data at a predictable interval. Interrupt transfers give guaranteed accuracy because errors are properly detected and transactions are retried at the next transaction. Interrupt transfers

[69] The CRC is a calculated value used for error checking. The CRC calculation is based on an equation defined in the USB specification.
[70] This is commonly used in *Human Interface Device* (HID) designs.

©Copyright 2012 Cypress Semiconductor

have a guaranteed bandwidth of 90% on low- and full-speed devices and 80% on high-speed devices. This bandwidth is shared with *isochronous endpoints*. The maximum packet size when employing interrupt endpoints is a function of device speed. High-speed capable devices support a maximum packet size of 1024 byes. Devices ca[able of operating at full speed support a maximum packet size of 64 bytes. Low-speed devices support a maximum packet size of 8 bytes.

- *Bulk Endpoints* support bulk transfers, which are commonly used on devices that move relatively large amounts of data at highly variable times where the transfers can use any available bandwidth space.[71] Delivery time for a bulk transfer is variable because there is no predefined bandwidth for the transfer, but instead, varies depending on how much bandwidth on the bus is available, which makes the actual delivery time unpredictable. Bulk transfers give guaranteed accuracy because errors are properly detected, and transactions are resent. Bulk transfers are useful in moving large amounts of data that are not time sensitive. A bulk endpoint maximum packet size is a function of device speed.[72] Devices that support full speed transfer have a maximum packet size of 64-bytes. Low-speed devices do not support bulk transfer types.

- *Isochronous Endpoints* support isochronous transfers, which are continuous, real-time transfers that have a pre-negotiated bandwidth. Isochronous transfers must support streams of error tolerant data because they do not have an error recovery mechanism, or handshaking. Errors are detected through the CRC field, but not corrected. With isochronous endpoints, a tradeoff must be made between guaranteed delivery and guaranteed accuracy. Streaming music, or video, are examples of an application that uses isochronous endpoints because the occasional missed data is ignored by human ears and eyes. Isochronous transfers have a guaranteed bandwidth of 90% on low and full-speed devices (80% on high-speed devices) that is shared with interrupt endpoints.

High-speed capable devices support a maximum packet size of 1024 bytes, full-speed devices 1023 bytes.[73] There are special considerations with isochronous transfers, e.g., 3x buffering is preferable to ensure data is ready to go by having one actively transmitting buffer, another buffer loaded and ready to transfer, and a third buffer being actively loaded.

4.6.4 USB Transfer Structure

During the enumeration process, the host requests the device descriptor. The transfer process consists of making the request for the device descriptor, receiving the device descriptor information, and the host acknowledging the successful reception of the data. However, the transfer consists of multiple stages called *transactions*. Each transfer consists of one or more transactions and in the case of the device descriptor request, there are three transactions. The first is the *Setup transaction*, the second is the*Data transaction*, where the descriptor information is sent to the host. The third transaction is the *handshake transaction* where the host acknowledges receiving the packet. Each transaction is made up of multiple packets and contains a token packet at minimum. Inclusion of a data packet and handshake packet can vary depending on the transfer type.

Each transfer contains one or more transactions, each of which always contains a *token packet.* A *data packet,* and *handshake packet,* may be included depending on the transaction type. Interrupt, bulk, and control transfers always include a token, data, and handshake packet with each

[71]They are the most common transfer type for USB devices.

[72]High-speed capable devices support a maximum BULK packet size of 512 bytes. Low-speed devices do not support bulk transfer.

[73]Low-speed devices do not support isochronous transfer types.

©Copyright 2012 Cypress Semiconductor

Table 4.7: Endpoint Transfer Type Features

Transfer	Control	Interrupt	Bulk	Isochronous
Typical User	Device Initialization and Management	Mouse and Keyboard	Printer and Mass Storage	Streaming Audio and Video
Low-speed Support	Yes	Yes	No	No
Error Correction	Yes	Yes	Yes	No
Guaranteed Delivery Rate	No	No	No	Yes
Guaranteed Bandwidth	Yes (10%)	Yes (90%)*	No	Yes (90%)*
Guaranteed Latency	No	Yes	No	Yes

* Shared bandwidth between isochronous and interrupt..

transaction. Control transfers have three stages: *Setup*, *Data*, and *Status*, and each one of these stages contains a token, data, and handshake packet. Therefore, while an Interrupt and Bulk transfer have a minimum of three packets, a control transfer has nine, or more, with a data stage and six or more without a data stage.

4.6.5 Transfer Composition

A USB packet has the structure as shown in Figure 4.19. A total of five fields can be populated, four of which are optional, and one is required.

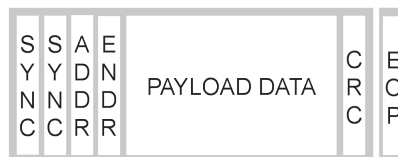


Figure 4.19: USB Packet contents.

Packet ID (PID) (8 bits: 4 type bits and 4 check bits)
Optional Device Address (7 bits: Max of 127 devices)
Optional Endpoint Address (4 bits: Max of 16 endpoints)
Optional Payload Data (0 to 1023 bytes)
Optional CRC (5 or 16 bits)

The *Packet ID* is the only required field in a packet. The *Device Address*, *Endpoint Address*, *Payload Data*, and *CRC* are filled depending on which packet type is sent. *Packet IDs* (PID)

©Copyright 2012 Cypress Semiconductor

are the heart of a USB packet. There are different PIDs depending on which packet is sent (see Table 1).

4.6.6 Packet Types

There are four different packet types, as shown in Figure 24. that can potentially represent.

- *Token packets*
  - Initiate a transaction.
  - Identify the device involved in transaction.
  - Are always sourced by the host.
- *Data packets*
  - Delivers payload data.
  - Are sourced by host or device.
- *Handshake packets*
  - Acknowledge error-free data receipt.
  - Are sourced by receiver of data.
- *Special packets*
  - Facilitates speed differentials.
  - Are sourced by host-to-hub devices.

Although everything in the packet, with the exception of the PID is optional, token, data, and handshake packets have different combinations of the packet information.

Token packets always come from the host, and are used to direct traffic on the bus. The function of the token packet depends on the activity performed, e.g., *IN tokens* are used to request that devices send data to the host and *OUT tokens* are used to precede data from the host. *SETUP tokens* are used to precede commands from the host and *SOF tokens* are used to mark time frames. With an *IN*, *OUT*, and *SETUP* token packet, there is a 7-bit device address, 4-bit endpoint ID, and 5-bit CRC.

The *SOF* gives a way for devices to identify the beginning of a frame and synchronize with the host. They are also used to prevent a device from entering suspend mode, which it must do if 3 milliseconds pass without an SOF. SOF packets are only found on full and high speed devices and are sent every millisecond. The SOF packet contains an 8-bit SOF PID, 11-bit frame count value (which rolls over when it reaches maximum value), and a 5-bit CRC. The CRC is the only error check used. A handshake packet does not occur for a SOF packet. High-speed communication goes a step further with microframes. With a high-speed device, a SOF is sent out every 125μs and frame count is only incremented every 1 ms.

Data packets follow *IN*, *OUT*, and *SETUP* token packets. The size of the payload data ranges from 0 to 1024 bytes, depending on the transfer type. The packet ID toggles between *DATA0* and *DATA1* for each successful data packet transfer, and the packet closes with a 16-bit CRC. The *data toggle* is updated at the host, and the device for each successful data packet transfer. One advantage to the *data toggle* is that it acts as additional error detection method. If a different packet ID is received than what is expected, the device will be able to know there was an error in the transfer and it can be handled appropriately. If an ACK is sent, but not received, the sender updates the data toggle from 1 to 0, but the receiver does not, and the data toggle remains at 1.

©Copyright 2012 Cypress Semiconductor

Handshake packets conclude each transaction. Each handshake includes an 8-bit packet ID and is sent by the receiver of the transaction. Each USB speed has several options for a handshake response.

The handshakes supported depend on the USB Speed:

- *ACK* is an acknowledgement of successful completion. (LS/FS/HS)
- *NAK* is a negative acknowledgement. (LS/FS/HS)
- *STALL* is an error indication sent by a device. (LS/FS/HS)
- *NYET* indicates the device is not ready to receive another data packet. (HS Only)

4.6.7 Transaction Types

Data from the host, and the device, are transferred from point A to point B, via *transactions*.. IN/Read/Upstream Transactions are terms that refer to a transaction that is sent from the device to the host. These transactions are initiated when the host by sends an *IN token packet*. The targeted device responds by sending one or more data packets, and the host responds with a *handshake packet*.

IN/Read/Upstream Special Packets are defined by the USB specification:

- *PRE* is issued to hubs by the host to indicate that the next packet is low speed.
- *SPLIT* precedes a token packet to indicate a split transaction. (HS Only)
- *ERR* is returned by a hub to report an error in a split transaction. (HS Only)
- *PING* checks the status for a Bulk OUT or Control Write after receiving a NYET handshake. (HS Only)

4.6.8 USB Descriptors

As described earlier, when a device is connected to a USB host, the device gives information to the host about its capabilities and power requirements. The device typically gives this information via a *descriptor table* that is part of its firmware. A descriptor table is a structured sequence of values that describe the device, and whose values are defined by the developer.

All descriptor tables contain a standard set of information that describes the device attributes and power requirements. If a design conforms to the requirement of a particular USB device class, additional descriptor information that the class must have is included in the device descriptor structure. When reading or creating descriptors, it is important to assure that the data fields are transmitted with the least significant bit first. Many parameters are 2 bytes long with the low byte occurring first and followed by the high byte.

Device descriptors provide the host with USB specification to which the device conforms, the number of device configurations, and protocols supported by the device, *Vendor Identification*[74], *Product Identification* (also known as a PID, different from a packet ID), and a *serial number*, if the device has one. The *Device Descriptor* contains the crucial information about the USB device.

Table 4.8 shows the structure for a device descriptor given that:

bLength is the total length, in bytes, of the device descriptor,

[74](also known as a VID, which is something that each company gets uniquely from the USB Implementers Forum)

©Copyright 2012 Cypress Semiconductor

Table 4.8: Device Descriptor Table.

Offset	Field	Size (Bytes)	Description
0	bLength	1	Descriptor Length = 18 bytes
1	bDescriptior Type	1	Descriptor type = DEVICE (01h)
2	bcdUSB	2	USB Spec Version (BCD)
4	bDeviceClass	1	Device Class
5	bDeviceSubClass	1	Device subclass
6	bDeviceProtocol	1	Device Protocol
7	bMaxPacketSize0	1	Max Packet Size for Endpoint 0
8	idVendor	2	Vendor ID (VID) (Assigned by USB-IF)
10	idProduct	2	Product ID (PID) (Assigned by the manufacturer)
12	bcdDevice	2	Device Release Number (BCD)
14	iManufacturer	1	Manufacturing String Index
15	idProduct	1	Product String Index
16	iSerialNumber	1	Serial Number String Index
17	bNumConfigurations	1	# 0f Supported Configurations

bcdUSB reports the USB revision that the device supports, which should be latest supported revision. This is a binary-coded decimal value that uses a format of 0xAABC, where A is the major version number, B is the minor version number, and C is the sub-minor version number. For example, a USB 2.0 device would have a value of 0x0200 and USB 1.1 would have a value of 0x0110. This is normally used by the host in determining which driver to load,

bDeviceClass,*bDeviceSubClass*, and*bDeviceProtocol* are used by the operating system to identify a driver for a USB device during the enumeration process. Filling in this field in the device descriptor prevents different interfaces from functioning independently, such as a composite device. Most USB devices define their class(es) in the interface descriptor, and leave these fields as 00h,

bMaxPacketSize reports the maximum number of packets supported by *Endpoint zero*. Depending on the device, the possible sizes are 8 bytes, 16 bytes, 32 bytes, and 64 bytes,

iManufacturer, *iProduct*, and *iSerialNumber* are indexes to string descriptors. String descriptors give details about the manufacturer, product, and serial number. If string descriptors exist, these variables should point to their index location. If no string exists, then the respective field should be assigned a value of zero,

and,

©Copyright 2012 Cypress Semiconductor

bNumConfigurations defines the total number of configurations the device can support. Multiple configurations allow the device to be configured differently depending on certain conditions, such as being bus-powered, or self-powered.

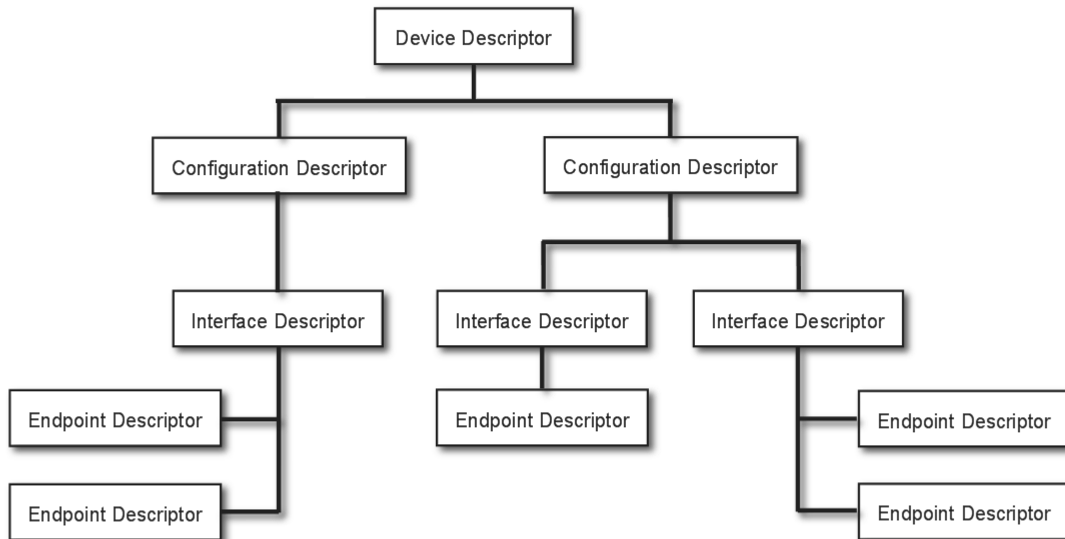


Figure 4.20: USB Descriptor Tree.

4.6.9 Configuration Descriptor

This descriptor gives information about a specific device configuration, e.g., the number of interfaces, whether the device is bus-powered or self-powered, if the device can start a remote wake-up, and how much power the device needs. Table 4.9 shows the structure for a configuration descriptor.

Table 4.9: Configuration Descriptor Type.

Offset	Field	Size (Bytes)	Description
0	bLength	1	Descriptor length = 9 bytes
1	bDescription Type	1	Descriptor Type = COMFIGURATION (02h)
2	wTotalLength	2	Total length including interface and endpoint descriptors
4	bNumInterface	1	Number of interfaces in the configuration
5	bConfiguration Value	1	Configuration values used by SET_CONFIGURATION to select the configuration
6	iConfiguration	1	String index that decribes the configuration
7	bmAttributes	1	Bit 7: Reserved (set to 1) Bit 6: Self-powered Bit 5: remote wakeup
8	bMaxPower	1	Max power required for the configuration (in 2 ma units)

©Copyright 2012 Cypress Semiconductor

wTotalLength is the length of the entire hierarchy of this configuration. This value reports the total number of bytes of the configuration, interface, and endpoint descriptors for one configuration.

bNumInterfaces defines the total number of possible interfaces in this particular configuration. This field has a minimum value of 1.

bConfigurationValue defines a value to use as an argument to the *SET_ CONFIGURATION* request to select this configuration.

bmAttributes defines parameters for the USB device. If the device is bus-powered, bit 6 is set to 0, if the device is self-powered, then bit 6 is set to 1. If the USB device supports remote wakeup, bit 5 is set to 1. If remote wakeup is not supported, bit 5 is set to 0.

bMaxPower defines the maximum power consumption drawn from the bus when the device is fully operational, expressed in 2 mA units. If a self-powered device becomes detached from its external power source, it may not draw more than the value indicated in this field.

4.6.10 Device Descriptor

Device descriptors give the host information, such as the USB specification to which the device conforms, the number of device configurations, and protocols supported by the device, *Vendor Identification*[75], *Product Identification*[76], and a serial number, if the device has one. The *Device Descriptor* contains of the most crucial information about the USB device. Table 4.9 shows the structure for a device descriptor.

4.7 Full Speed USB (USBFS)

PSoC Creator's *USBFS component* provides a USB, full-speed, Chapter 9 compliant device, framework.[77] It provides a low-level driver for the control endpoint that decodes and dispatches requests from the USB host. Additionally, this component provides a USBFS customizer to make it easy to construct the appropriate descriptor. The option of constructing a HID-based device or a generic USB Device is also provided. In PSoC Creator, HID can be selected by setting the *Configuration/Interface* descriptors. The USBFS component can be used to provide an interface that is USB 2.0 compliant.

USB transmissions are based on one of several types of transfer, viz., bulk, control, interrupt and isochronous, depending on the application. While the formal USB specification defines specific commands that may be required for a USB device to receive and respond to USB transmission, it is also possible for the designer to introduce custom commands[78]. Reliable data transmission schemes often rely on data integrity algorithms to detect, and perhaps correct, errors and/or generate an error signal. Handshaking schemes provide feedback to the transmitter to indicate whether or not data integrity has been preserved, and thereby allow retransmission of data to be employed in the event of errors in transmission. The *start-of-frame* (sof) output for the component allows endpoints to identify the start of the frame and synchronize internal endpoint clocks to the host.

[75]Also known as a VID, which is something that each company gets uniquely from the USB Implementers Forum)

[76]Referred to as as a PID, and it is different from a packet ID.

[77]SuiteUSB, a set of USB development tools, is available free of charge when used with Cypress silicon. http://www.cypress.com.

[78]Such custom commands, e.g., introduced to provide control of a specific type of device, are often referred to as *vendor commands.*

©Copyright 2012 Cypress Semiconductor

4.7.1 Endpoint Memory Management

The USBFS block contains 512 bytes of target memory for the data endpoints to use. However, the architecture supports a *cut-through mode* of operation, referred to as *DMA w/Automatic Memory Management*, that reduces the memory requirement, based on system performance. Some applications can benefit from using Direct Memory Access (DMA) to move data into and out of the endpoint memory buffers.

- Manual (default) Select this option to use *LoadInEP/ReadOutEP* to load and unload the endpoint buffers.
  - *Static Allocation* - The memory for the endpoints is allocated immediately after a *SET_ CONFIGURATION* request. This takes longest when multiple *Alternate* settings use the same endpoint (EP) number.
  - *Dynamic Allocation* - The memory for the endpoints is allocated dynamically after each *SET_ CONFIGURATION* and *SET_ INTERFACE* request. This option is useful when multiple alternate settings are used with mutually exclusive EP settings.
  - *DMA w/Manual Memory Management*[79] - Select this option for manual DMA transactions. The*LoadInEP/ReadOutEP* functions fully support this mode and initialize the DMA automatically.[80]
  - *DMA w/Automatic Memory Management* - Select this option for automatic DMA transactions. This is the only configuration that supports combined data endpoint use of more than 512 bytes. *LoadInEP/ReadOutEP* functions should be used for initial DMA configuration.

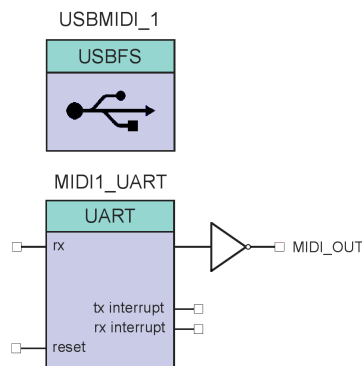


Figure 4.21: PSoC3/5's Midi component

4.7.2 Enabling VBUS Monitoring

USB signals are transmitted via a USB cable consisting of a twisted pair that has a characteristic impedance of 90 ohms, a shield that functions as a ground return and power connections D+ and D−. The protocol assumes that there are no more than 127 devices[81] connected at any one time,

[79] PSoC3 does not support DMA transactions directly between USB endpoints and other peripherals. All DMA transactions involving USB endpoints, both in and out, must terminate, or originate, with main system memory. Applications requiring DMA transactions directly between USB endpoints, and other peripherals, must use two DMA transactions to move data to main system memory as an intermediate step between the USB endpoint and the other peripheral.

[80] This option is supported for PSoC 3 Production silicon only.

[81] This limitation is a result, in part, of the fact that the address field is 7 bits and that address zero is reserved.

©Copyright 2012 Cypress Semiconductor

in a *tiered-star* topology. The maximum allowable cable length between hubs is 5 meters, and no more than six hubs are supported, for a maximum of thirty meters. The USB specification requires that no device supplies current on VBUS at its upstream facing port at any time. To meet this requirement, the device must monitor for the presence, or absence, of VBUS and remove power from the D+/D- pull-up resistor, if VBUS is absent. For bus-powered designs, power will obviously be removed when the USB cable is removed from a host but, for self-powered designs, it is imperative for proper operation, and USB certification, that the device comply with this requirement.

4.7.2.1 USBFS MIDI

The USBFS MIDI component, shown in Figure 4.21, provides support for communicating with external MIDI equipment and for the USB device class definition for MIDI devices. This component can be used to add MIDI I/O capability to a standalone device, or to implement MIDI capability for a host computer, or mobile device, through a computer's, or mobile device's, USB port. In such cases, it appears to the host computer, or mobile device, as a class-compliant USB MIDI device, and it uses the native MIDI drivers in the host.

The supported features include:

- USB MIDI Class Compliant MIDI input and output.
- Hardware interfacing to external MIDI equipment using UART.
- Adjustable transmit and receive buffers managed using interrupts.
- MIDI running status for both receive and transmit functions.
- Up to 16 input, and output, ports using only two USB endpoints by using virtual cables.

The *PSoC Creator Component* catalog contains a *Schematic Macro* implementation of a MIDI[82] interface. The macro consists of instances of the UART component with the hardware MIDI interface configuration (31.25 kbps, 8 data bits) and a USBFS component with the descriptors configured to support MIDI devices. This allows the user to employ a MIDI-enabled, USBFS component with minimal configuration changes. A *USBMIDI Schematic Macro* labeled *USBMIDI* is available in PSoC Creator that has been previously configured to function as an external mode MIDI device with 1 input and 1 output.

4.7.3 USB Function Calls

PSoC Creator provides an extensive list of USB function calls and, by default, assigns the instance name *USBFS_1* to the first instance of a component in a given design. However, such instance names can be renamed to any unique value that follows the syntactic rules for identifiers. In any event, the instance name becomes the prefix of every global function name, variable, and constant symbol.

For readability, the instance name used in the following is *USBFS*.

- *void USBFS_Start(uint8 device, uint8 mode)* performs all required initialization for the USBFS Component.

[82]The musical instrument digital interface (MIDI), defined by the MIDI Manufacturing Association in 1982, is an industry standard protocol for intercommunication between a wide variety of music related devices. It serves as a software, hardware, communication and instrument categorization standard and is often employed to allow one instrument to control an arbitrary number of other musical instruments, or music-related equipment.

©Copyright 2012 Cypress Semiconductor

- *void USBFS_Init(void)* initializes, or restores, the component according to the customizer *Configure* dialog settings.[83]
- *void USBFS_InitComponent(uint8 device, uint8 mode)* initializes the component's global variables and initiates communication with the host by pulling up the D+ line.
- *void USBFS_Stop(void)* performs all necessary shutdown tasks required for the USBFS component.
- *uint8 USBFS_ GetConfiguration(void)* gets the current configuration of the USB device.
- *uint8 USBFS_ IsConfigurationChanged(void)* returns the *clear-on-read* configuration state. It is useful when the PC sends double *SET_ CONFIGURATION* requests with the same configuration number.
- *uint8 USBFS_ GetInterfacuint8 USBFS_ GetEPState(uint8 epNumber)* returns the state of the requested endpoint.
- *uint8 USBFS_ GetInterfaceSetting(uint8 interfaceNumber)* gets the current alternate setting for the specified interface.
- *uint8 USBFS_ GetEPState(uint8 epNumber)* returns the state of the requested endpoint.
- *uint8 USBFS_ GetEPAckState(uint8 epNumber)* determines whether or not an ACK transaction occurred on this endpoint by reading the ACK bit in the control register of the endpoint.[84]
- *uint16 USBFS_ GetEPCount(uint8 epNumber)* returns the transfer count for the requested endpoint. The value from the count registers includes two counts for the two-byte checksum of the packet. This function subtracts the two counts.
- *void USBFS_InitEP_DMA(uint8 epNumber, uint8 *pData)*[85] allocates and initializes a DMA channel to be used by the *USBFS_LoadInEP()* or *USBFS_ReadOutEP()* APIs for data transfer. It is available when the Endpoint Memory Management parameter is set to DMA.
- void USBFS_LoadInEP(uint8 epNumber, uint8 *pData, uint16 length) in manual mode: loads and enables the specified USB data endpoint for an IN data transfer.

  Manual DMA:

  - Configures DMA for a transfer data from data RAM to endpoint RAM.
  - Generates request for a transfer.

  Automatic DMA:

  - Configures DMA. This is required only once, therefore it is done only when parameter Data is not NULL. When pData pointer is NULL, the function skips this task.
  - Sets Data ready status: This generates the first DMA transfer and prepares data in endpoint RAM memory.
- *uint16 USBFS_ ReadOutEP(uint8 epNumber, uint8 *pData, uint16 length)* in manual mode moves the specified number of bytes from endpoint RAM to data RAM. The number of bytes actually transferred from endpoint RAM, to data RAM, is the lesser of the actual number of bytes sent by the host, or the number of bytes requested by the wCount parameter.

  Manual DMA:

  - Configures DMA for a transfer data from endpoint RAM to data RAM.

[83] It is not necessary to *call USBFS_Init()* because the *USBFS_ Start()* routine calls this function and is the preferred method to begin component operation.
[84] This function does not clear the ACK bit.
[85] This function is automatically called from the *USBFS_ LoadInEP()* and *USBFS_ ReadOutEP()* APIs.

©Copyright 2012 Cypress Semiconductor

- Generates request for a transfer.
- After *USB_ReadOutEP()*API and before expected data usage it is required to wait on DMA transfer complete. For example by checking EPstate:
  while *(USBFS_GetEPState(OUT_EP) == USB_OUT_BUFFER_FULL);*

Automatic DMA:

- Configures DMA.[86]

- *void USBFS_EnableOutEP(uint8 epNumber)* enables the specified endpoint for OUT transfers.
- *void USBFS_DisableOutEP(uint8 epNumber)* disables the specified USBFS OUT endpoint.[87]
- *void USBFS_SetPowerStatus(uint8 powerStatus)* sets the current power status. The device replies to *USB GET_STATUS* requests based on this value. This allows the device to properly report its status for *USB Chapter 9* compliance. Devices can change their power source from self-powered to bus-powered, at any time, and report their current power source as part of the device status. This function can be called any time the device changes from self-powered to bus-powered, or vice versa, and set the status appropriately.
- *void USBFS_Force(uint8 state)* forces a USB J, K, or SE0 state on the D+/D- lines. This function provides the necessary mechanism for a USB device application to perform a USB Remote Wakeup.[88]
- *void USBFS_SerialNumString(uint8 *snString)* is available only when the *User Call Back* option in the *Serial Number String* descriptor properties is selected. Application firmware can provide the source of the USB device serial number string descriptor during runtime. The default string is used if the application firmware does not use this function or sets the wrong string descriptor.
- *void USBFS_TerminateEP(uint8 epNumber)*[89] terminates the specified USBFS endpoint.
- *uint8 USBFS_UpdateHIDTimer(uint8 interface)* updates the HID Report idle timer and returns the status and reloads the timer, if it expires.
- *uint8 USBFS_GetProtocol(uint8 interface)* returns the HID protocol value for the selected interface.

4.8 Controller Area Network (CAN)

The Controller Area Network (CAN) controller implements the CAN2.0A and CAN2.0B specifications as defined in the Bosch specification and conforms to the ISO-11898-1 standard. The CAN protocol was originally designed for automotive applications with a focus on a high level of fault detection thereby ensuring high communication reliability at a low cost. Because of its success in automotive applications, CAN is used as a standard communication protocol for motion-oriented, machine-control networks (*CANOpen*) and factory automation applications (*DeviceNet*). The CAN controller features make it possible to efficiently implement higher-level protocols, without adversely affecting the performance of the microcontroller CPU.

CAN is an *arbitration-free* system in that the highest priority message is always transmitted first. The transmit buffer arbitration scheme employed can be either *round-robin*, the default mode, or *fixed priority*. In the round-robin mode, buffers are served in the following order:

[86]This is required only once.
[87]Do not call this function for IN endpoints.
[88]For more information, refer to the USB 2.0 Specification for details on *Suspend* and *Resume*.
[89]This function should be used before endpoint reconfiguration.

©Copyright 2012 Cypress Semiconductor

$0 - 1 - 2 \cdots 7 - 0 - 1$.[90] In the fixed priority mode, buffer zero is assigned the highest priority which allows it to be the error message buffer thereby assuring that error messages are transmitted first.

4.8.1 PSoC Creator's CAN Component

This component has three standard I/O connections, and a fourth, optional, *interrupt* connection[91], as shown in Figure 4.22.

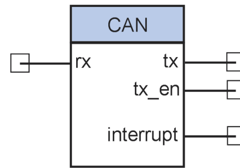


Figure 4.22: PSoC Creator's CAN component.

- *rx* is the CAN bus receive (input) signal and is connected to the CAN Rx bus which is external to the transceiver.
- *tx* is the CAN bus transmit signal and is connected to the CAN Tx bus of the external transceiver.
- *tx_ en* is the external transceiver enable signal.

The default CAN configuration in the *Component Catalog* is a schematic macro using a CAN component with default settings, and is connected to an *Input* and an *Output Pins* component. The Pins components are also configured with default settings, except that *Input Synchronized* is set to false in the *Input Pin* component.

4.8.2 Interrupt Service Routines

There are several CAN component interrupt sources, all of which have entry points (functions) that allow user code to be placed in them.[92]

- *Acknowledge Error* - The CAN controller detected a CAN message acknowledge error.
- *Arbitration Lost Detection* - The arbitration was lost while sending a message.
- *Bit Error* - The CAN controller detected a bit error.
- *Bit Stuff Error* - The CAN controller detected a bit stuffing[93] error.
- *Bus Off* - The CAN controller has reached the bus-off state
- *CRC Error* - The CAN controller detected a CAN CRC error.
- *Form Error* - The CAN controller detected a CAN message format error.
- *Message Lost* - A new message arrived, but there was nowhere to put it.
- *Transmit Message* - The queued message was sent.
- *Receive Message* - A message was received.[94]

[90] This mode assures that all buffers have the same probability of sending a message.
[91] This output is displayed in PSoC Creator only when the *Add Transceiver Enable Signal* option has been selected in the *Config* dialog.
[92] These functions are conditionally compiled, depending on the customizer.
[93] *Bit stuffing* refers to the introduction of of "non-information" bits into frames, buffers, etc., for the purpose of filling them.
[94] The Receive Message interrupt has a special handler that calls appropriate functions for Full and Basic mailboxes.

©Copyright 2012 Cypress Semiconductor

4.8.3 Hardware Control of Logic on Interrupt Events

The hardware interrupt input can be used to perform simple tasks such as estimating the CAN bus load. By enabling the *Message Transmitted* and *Message Received* interrupts in the CAN component customizer, and connecting the interrupt line to a counter, the number of messages that are on the bus during a specific time interval can be evaluated. Actions can be taken directly in hardware if the message rate is above a certain value.

4.8.4 Interrupt Output Interaction with DMA

PSoC Creator's CAN component does not support DMA operation internally, but the DMA component can be connected to the external interrupt line, if it is enabled and provided that the designer assumes responsibility for the DMA configuration and operation. However, it is necessary to manage some housekeeping tasks, e.g., acknowledging the message and clearing the interrupt flags, in code to handle CAN interrupts properly. With a hardware DMA trigger, registers and data transfers can be handled when a *Message Received* interrupt occurs, without any firmware executing in the CPU. [95] The Message Transmitted interrupt can be used to trigger a DMA transfer to reload the message buffer with new data, without CPU intervention.

4.8.5 Custom External Interrupt Service Routine

Custom external ISRs can be used in addition to, or as a replacement for, the internal ISR. When both external and internal ISRs nare used, the Interrupt priority can be set to determine which ISR should execute first, i.e., internal or external, thus forcing actions before, or after, those coded in the internal ISR. When the external ISR is used, as replacement for the internal ISR, the designer is responsible for proper handling of CAN registers and events.

The external interrupt line is visible only if it is enabled in the customizer. If an external Interrupt component is connected, the external Interrupt component is not started as part of the *CAN_Start()* API, and must be started outside that routine. If an external Interrupt component is connected and the internal ISR is not disabled or bypassed, two Interrupt components are connected to the same line. In this case, there will be two separate Interrupt components that will handle the same interrupt events which in most cases is undesirable.

If the internal ISR is disabled, or bypassed using a customizer option, the internal Interrupt component will be removed during the build process. If individual interrupt function call is disabled in the internal interrupt routine, for an enabled interrupt event by using a customizer option, the CAN block interrupt triggers, when the relevant event occurs, but no internal function call is executed in the internal *CAN_ISR* routine. If a specific event needs to be handled, e.g., message received, through a different path, other than the standard user function call, through DMA. If the internal ISR is to be customized, using customizer options, the *CAN_ISR*function will not contain any function call other than the optional PSoC 3 ES1/ES2 ISR patch.

There are several important references that should be consulted when designing systems that involving controller area networks, viz.,

- ISO-11898: Road vehicles – Controller area network (CAN):
  - Part 1: Data link layer and physical signaling
  - Part 2: High-speed medium access unit Controller Area Network (CAN)
  - Part 3: Low-speed, fault-tolerant, medium-dependent interface
  - Part 4: Time-triggered communication

[95] This is also useful when handling RTR messages.

©Copyright 2012 Cypress Semiconductor

  - Part 5: High-speed medium access unit with low-power mode
- CAN Specification Version 2 BOSCH
- Inicore CANmodule-III-AHB Datasheet

4.8.6 Interrupt Output Interaction with the Interrupt Subsystem

The CAN component Interrupt Output settings allow:

- Enabling or disabling of an external interrupt line (customizer option)
- Disabling or bypassing the internal ISR (customizer option)
- Full customization of the internal ISR (customizer option)
- Enabling, or disabling, of specific interrupts handling function calls in the internal ISR, when the relevant event interrupts are enabled using the customizer option. Individual interrupts, e.g., message transmitted, message received, receive buffer full, bus off state,etc., can be enabled, or disabled, in the CAN component customizer. Once enabled, the relevant function call is executed in the internal *CAN_ISR*. This allows disabling, i.e., removing, of such function calls.

The external interrupt line is visible only if it is enabled in the customizer.

- *uint8 CAN_Start(void)* sets the *initVar* variable, calls the *CAN_Init()* function, and then calls the *CAN_Enable()* function. This function sets the CAN component into run mode and starts the counter, if polling mailboxes available.
- *uint8 CAN_Stop(void)* sets the CAN component into Stop mode and stops the counter, if polling mailboxes available.
- *uint8 CAN_GlobalIntEnable(void)* enables global interrupts from the CAN component.
- *uint8 CAN_GlobalIntDisable(void)* disables global interrupts from the CAN component.
- *uint8 CAN_SetPreScaler(uint16 bitrate)* sets the *prescaler* for generation of the time quanta from the *BUS_CLK*. Values between 0x0, and 0x7FFF, are valid.
- *uint8 CAN_SetArbiter(uint8 arbiter)* sets the arbitration type for transmit buffers. Types of arbiters are Round Robin and Fixed priority. Values 0 and 1 are valid.
- *uint8 CAN_SetTsegSample(uint8 cfgTseg1, uint8 cfgTseg2, uint8 sjw, uint8 sm)* this function configures: *Time segment 1*, *Time segment 2*, *Synchronization Jump Width*, and Sampling Mode.
- *uint8 CAN_SetRestartType(uint8 reset)* sets the reset type. Types of reset are *Automatic* and *Manual*. *Manual* reset is the recommended setting. Values 0 and 1 are valid.
- uint8 CAN_SetEdgeMode(uint8 edge) sets *Edge Mode*. Modes are 'R' to 'D' (Recessive to Dominant) and Both edges are used. Values 0 and 1 are valid.
- *uint8 CAN_RXRegisterInit(uint32 *regAddr, uint32 config)* writes CAN receive registers only.
- *uint8 CAN_SetOpMode(uint8 opMode)* sets *Operation Mode*. Operation modes are *Active* or *Listen Only*. Values 0 and 1 are valid.
- uint8 CAN_GetTXErrorflag(void) returns the flag that indicates whether or not the number of transmit errors exceeds 0x60.
- *uint8 CAN_GetRXErrorflag(void)* returns the flag that indicates whether or not the number of receive errors has exceeded 0x60.

©Copyright 2012 Cypress Semiconductor

- *uint8 CAN_ GetTXErrorCount(void)* returns the number of transmit errors.
- *uint8 CAN_ GetRXErrorCount(void)* returns the number of receive errors.
- *uint8 CAN_ GetRXErrorCount(void)* returns the number of receive errors.
- *uint8 CAN_ GetErrorState(void)* returns the error status of the CAN component.
- *uint8 CAN_ SetIrqMask(uint16 mask)* enables, or disables, particular interrupt sources. *Interrupt Mask* directly writes to the CAN Interrupt Enable register.
- *void CAN_ ArbLostIsr(void)* is the entry point to the *Arbitration Lost Interrupt*. It clears the *Arbitration Lost*interrupt flag. It is only generated, if the *Arbitration Lost Interrupt* parameter is enabled.
- *void CAN_ OvrLdErrrorIsr(void)* is the entry point to the *Overload Error Interrupt*. It clears the *Overload Error* interrupt flag. It is only generated, if the *Overload Error Interrupt* parameter is enabled.
- *void CAN_ BitErrorIsr(void)* is the entry point to the *Bit Error Interrupt*. It clears *Bit Error Interrupt* flag. It is only generated, if the *Bit Error Interrupt* parameter is enabled.
- *void CAN_ BitStuffErrorIsr(void)* is the entry point to the *Bit Stuff Error Interrupt*. It clears the *Bit Stuff Error Interrupt* flag. It is only generated, if the *Bit Stuff Error Interrupt* parameter is enabled.
- *void CAN_ AckErrorIsr(void)* is the entry point to the *Acknowledge Error Interrupt*. It clears the *Acknowledge Error* interrupt flag and is only generated, if the *Acknowledge Error Interrupt* parameter is enabled.
- *void CAN_ MsgErrorIsr(void)* is the entry point to the *Form Error Interrupt*. It clears the *Form Error* interrupt flag. It is only generated, if the *Form Error Interrupt* parameter is enabled.
- *void CAN_ CrcErrorIsr(void)* is the entry point to the *CRC Error Interrupt*. It clears the *CRC Error* interrupt flag. It is only generated, if the *CRC Error Interrupt* parameter is enabled.
- *void CAN_ BusOffIsr(void)* is the entry point to the *Bus Off Interrupt*. It puts the CAN component in Stop mode. It is only generated, if the *Bus Off Interrupt* parameter is enabled. Enabling this interrupt is recommended.
- *void CAN_ MsgLostIsr(void)* is the entry point to the *Message Lost Interrupt*. It clears the *Message Lost Interrupt* flag. It is only generated, if the *Message Lost Interrupt* parameter is enabled.
- *void CAN_ MsgTXIsr(void)* is the entry point to the *Transmit Message Interrupt*. It clears the *Transmit Message Interrupt* flag. It is only generated, if the *Transmit Message Interrupt* parameter is enabled.
- *void CAN_ MsgRXIsr(void)* is the entry point to the *Receive Message Interrupt*. It clears the *Receive Message Interrupt* flag and calls the appropriate handlers for *Basic* and *Full* interrupt-based mailboxes. It is only generated, if the *Receive Message Interrupt* parameter is enabled. Enabling this interrupt is recommended.
- *uint8 CAN_ RxBufConfig(CAN_ RX_ CFG *rxConfig)* function configures all receive registers for a particular mailbox. The mailbox number contains *CAN_ RX_ CFG* structure.
- *uint8 CAN_ TxBufConfig(CAN_ TX_ CFG *txConfig)* configures all transmit registers for a particular mailbox. The mailbox number contains *CAN_ TX_ CFG* structure.
- *uint8 CAN_ SendMsg(CANTXMsg *message)* sends a message from one of the *Basic* mailboxes. The function loops through the transmit message buffer designed as *Basic* CAN mailboxes. It looks for the first free available mailbox and sends from it. There can only be three retries.

©Copyright 2012 Cypress Semiconductor

- *uint8 CAN_SendMsg0-7(void)* are the entry point to *Transmit Message* 0-7. This function checks if mailbox 0-7 already has untransmitted messages waiting for arbitration. If so, it initiates transmission of the message. It is only generated for Transmit mailboxes designed as *Full*.
- void CAN_TxCancel(uint8 bufferld) cancels transmission of a message that has been queued for transmission. Values between 0 and 15 are valid.
- *void CAN_ReceiveMsg0-15(void)* are the entry point to the *Receive Message* 0-15 Interrupt. They clear Receive Message 0 - 15 interrupt flags. They are only generated for *Receive* mailboxes designed as *Full* interrupt based.
- *void CAN_ReceiveMsg(uint8 rxMailbox)* is the entry point to the *Receive Message Interrupt* for Basic mailboxes. It clears the *Receive* particular Message interrupt flag. It is only generated, if one of the Receive mailboxes is designed as *Basic.*
- *void CAN_Sleep(void)* is the preferred routine to prepare the component for sleep. The *CAN_Sleep()* routine saves the current component state. Then it calls the *CAN_Stop()* function and calls CAN_SaveConfig() to save the hardware configuration. The

  CAN_Sleep() function must be called before calling the *CyPmSleep()* or the *CyPmHibernate()* function.
- *void CAN_Wakeup(void)* is the preferred routine to restore the component to the state when *CAN_Sleep()* was called. The *CAN_Wakeup()* function calls the *CAN_RestoreConfig()* function to restore the configuration. If the component was enabled before the *CAN_Sleep()* function was called, the *CAN_Wakeup()* function will also re-enable the component.
- *uint8 CAN_Init(void)* initializes, or restores, the component according to the customizer *Configure* dialog settings. It is not necessary to call *CAN_Init()* because the *CAN_Start()* routine calls this function and is the preferred method to begin component operation.
- *uint8 CAN_Enable(void)* activates the hardware and begins component operation. It is not necessary to call CAN_Enable() because the *CAN_Start()* routine calls this function, which is the preferred method to begin component operation.
- *void CAN_SaveConfig(void)* saves the component configuration and nonretention registers. This function also saves the current component parameter values, as defined in the *Configure* dialog or as modified by appropriate APIs. This function is called by the *CAN_Sleep()* function.
- *void CAN_RestoreConfig(void)* restores the component configuration and nonretention registers. This function also restores the component parameter values to what they were prior to calling the *CAN_Sleep()* function.

4.9 S/PDIF Transmitter (SPDIF_Tx)

PSoC3/5's *SPDIF_Tx* component[96] provides a simple way to add digital audio output to any design[97]. It formats incoming audio- and meta-data to create a *S/PDIF* bit stream appropriate for optical, or coaxial, digital audio. This component, shown in Figure 4.23, supports interleaved and separated audio. The *SPDIF_Tx* component receives audio data from DMA, as well as, channel status information. Although the channel status DMA will be managed by the component, alteratively, this data can be handled separately to better control a given system.

[96]This component can be used in conjunction with an I2S component and external ADC to convert from analog audio to digital audio.

[97]S/PDIF refers to the Sony Philips digital interface data link layer protocol and an associated physical layer specification. This protocol is often used to transfer compressed digital audio and has no defined data data rate.

©Copyright 2012 Cypress Semiconductor

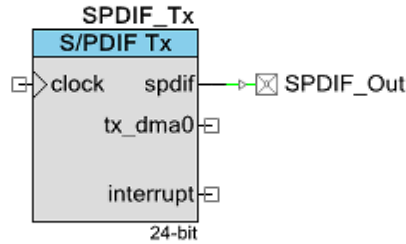


Figure 4.23: PSoC3/5's S/PDIF transmitter component.

SPDIF_Tx provides a fast solution whenever an *S/PDIF* transmitter is essential, including, e.g., digital audio players, computer audio interfaces and audio mastering equipment.

The supported features of the SPDIF_Tx include:

- conforming to IEC-60958, AES/EBU, AES3 standards for Linear PCM Audio Transmission,
- configurable audio sample lengths (8/16/24),
- or channel status bits generator for consumer applications,
- DMA support,
- sample rate support for clock/128 (up to 192 kHz),

and,

- independent left and right channel FIFOs, or interleaved stereo FIFOs.

4.9.1 SPDIF_Tx component I/O Connections[98]

The following are the available I/O connections for PSoC3/5's SPDIF_Tx component:

clock - The clock rate must be two times the desired data rate for the *spdif* output, e.g., production of 48-kHz audio, would require a clock frequency given by:

$$(2)(48kHz)(64) = 6.144MHz \tag{4.1}$$

spdif - Serial data output.

sck - Serial clock output.

interrupt - Interrupt output.

tx_DMA0 - DMA request output for audio FIFO 0 (Channel 0 or Interleaved).

tx_DMA1 - DMA request for audio FIFO 1 (Channel 1) output. Displays, if *Separated* under the *Audio Mode* parameter is selected.

*cst_DMA0** - Request for channel status FIFO 0 (Channel 0) output. Displays, if the *checkbox* under the *Managed DMA* parameter is deselected.

*cst_DMA1** - Request for channel status FIFO 1 (Channel 1) output. Displays, if the checkbox under the Managed DMA parameter is deselected.

[98] An asterisk (*) in the list of indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

©Copyright 2012 Cypress Semiconductor

4.9.2 SPDIF_Tx API

The *SPDIF_Tx API* supports the following functions:

void SPDIF_Start(void) starts the *S/PDIF* interface, and the channel status DMA, if the component is configured to handle the channel status DMA. It also enables the Active mode power template bits, or clock gating as appropriate, starts the generation of the *S/PDIF* output with channel status, but the audio data is set to all 0s. It also allows the *S/PDIF* receiver to lock on to the component's clock.

void *SPDIF_Stop(void)* disables the *S/PDIF* interface and the active mode power template bits or clock gating, as appropriate. The *S/PDIF* output is set to 0. The audio data and channel data FIFOs are cleared. The *SPDIF_Stop()* function calls *SPDIF_DisableTx()* and stops the managed channel status DMA.

void SPDIF_Sleep(void) is the preferred routine to prepare the component for sleep.[99] The *SPDIF_Sleep()* routine saves the current component state and then calls *SPDIF_Stop()* and *SPDIF_SaveConfig()* saves the hardware configuration, disables the active mode power template bits, or clock gating, as appropriate, sets the spdif output to 0. *SPDIF_Sleep()* should be called *CyPmSleep()* or *CyPmHibernate()* are called.

void SPDIF_Wakeup(void) restores the *SPDIF* configuration and nonretention register values. The component is stopped, regardless of its state before sleep. The *SPDIF_Start()* function must be called explicitly to start the component again.[100]

void SPDIF_EnableTx(void) enables the audio data output in the S/PDIF bit stream. Transmission will begin at the next X, or Z, frame.

void SPDIF_DisableTx(void) disables the audio output in the S/PDIF bit stream. Transmission of data will stop at the next rising edge of clock and a constant 0 value will be transmitted.

void SPDIF_WriteTxByte(uint8 wrData, uint8 channelSelect) writes a single byte into the audio data FIFO. The component status should be checked before this call to confirm that the audio data FIFO is not full. *uint8 wrData* contains the audio data to transmit. *uint8 channelSelect* contains the constant for *Channel* to write. See channel status macros below. In the interleaved mode this parameter is ignored.

void SPDIF_WriteCstByte(uint8 wrData, uint8 channelSelect) writes a single byte into the specified channel status FIFO. The component status should be checked before this call to confirm that the channel status FIFO is not full. *uint8 wrData* contains the status data to transmit and *uint8 channelSelect* the constant for the Channel to be written to.

void SPDIF_SetInterruptMode(uint8 interruptSource) sets the interrupt source for the S/PDIF interrupt. Multiple sources may be ORed.

The *SPDIF* component formats incoming audio data and metadata to create the *S/PDIF* bit stream. This component receives audio data from DMA, as well as, channel status information. Most of the time, the channel status DMA is managed by the component. However, there is an option that allows the data to be specified separately, to better control a system.

[99] *SPDIF_Sleep()* should be called before calling *CyPmSleep()* or *CyPmHibernate()*.
[100] Calling *SPDIF_Wakeup()* without first calling *SPDIF_Sleep()*or *SPDIF_SaveConfig()* may produce unexpected behavior.

©Copyright 2012 Cypress Semiconductor

SPDIF Tx Interrupt Source	Value
AUDIO_FIFO_UNDERFLOW	0x01
AUDIO_0_FIFO_NOT_FULL	0x02
AUDIO_1_FIFO_NOT_FULL	0x04
CHST_FIFO_UNDERFLOW	0x08
CHST_0_FIFO_NOT_FULL	0x10
CHST_1_FIFO_NOT_FULL	0x20

Figure 4.24: SPDIF - Interrupt mode values.

SPDIF Status Masks	Value	Type
AUDIO_FIFO_UNDERFLOW	0x01	Clear on read
AUDIO_0_FIFO_NOT_FULL	0x02	Transparent
AUDIO_1_FIFO_NOT_FULL	0x04	Transparent
CHST_FIFO_UNDERFLOW	0x08	Clear on read
CHST_0_FIFO_NOT_FULL	0x10	Transparent
CHST_1_FIFO_NOT_FULL	0x20	Transparent

Figure 4.25: SPDIF - Status mask values.

Name	Description
SPDIF_SPS_22KHZ	Clock rate is set for 22-kHz audio
SPDIF_SPS_44KHZ	Clock rate is set for 44-kHz audio
SPDIF_SPS_88KHZ	Clock rate is set for 88-kHz audio
SPDIF_SPS_24KHZ	Clock rate is set for 24-kHz audio
SPDIF_SPS_48KHZ	Clock rate is set for 48-kHz audio
SPDIF_SPS_96KhZ	Clock rate is set for 96-kHz audio
SPDIF_SPS_32KHZ	Clock rate is set for 32-kHz audio
SPDIF_SPS_64KHZ	Clock rate is set for 64-kHz audio
SPDIF_SPS_192KHZ	Clock rate is set for 192-kHz audio
SPDIF_SPS_UNKNOWN	Clock rate is not specified.

Figure 4.26: SPDIF - Frequency values.

©Copyright 2012 Cypress Semiconductor

4.9.3 S/PDIF Data Stream Format

The audio and channel status data are independent byte streams, packed with the least significant byte and bit first. The number of bytes used for each sample is the minimum number of bytes to hold a sample. Any unused bits will be padded with zeros, starting at the left-most bit. The audio data stream can be a single byte stream, or it can be two byte streams. In the case of a single byte stream, the left and right channels are interleaved with a sample for the left channel first followed by the right channel. In the two stream case, the left and right channel byte streams use separate FIFOs. The status byte stream is always two byte streams.

4.9.4 S/PDIF and DMA Transfers

The S/PDIF interface is a continuous interface that requires an uninterrupted stream of data. For most applications, this requires the use of DMA transfers to prevent the underflow of the audio data or channel status FIFOs. Typically, the Channel Status DMA occurs entirely using two channel status arrays and can be modified using macros. However, data can be provided by an external DMA or CPU to allow flexibility. The S/PDIF can drive up to four DMA components, depending on the component configuration. DMA configuration, using PSoC Creator's DMA Wizard, should be based on Table 4.10.

Table 4.10: SPDIF DMA configuration parameters.

Name of the DMA Source Destination in the DMA Wizard	Direction	DMA Request Signal	DMA Request Type	Description
SPDIF_TX_FIFO_0_PTR	Destination	tx_dma0	Level	Transmit FIFO for Channel 0 or Interleaved Audio Data
SPDIF_TX_FIFO_1_PTR	Destination	tx_dma1	Level	Transmit FIFO for Channel 1 or Interleaved Audio Data
SPDIF_CST_FIFO_0_PTR	Destination	cst_dma0	Level	Transmit FIFO for Channel 0 or Interleaved Status Data
SPDIF_CST_FIFO_1_PTR	Destination	cst_dma1	Level	Transmit FIFO for Channel 1 or Interleaved Status Data

4.9.5 S/PDIF Channel Encoding

S/PDIF is a single-wire serial interface. The bit clock is embedded within the *S/PDIF* data stream. The digital signal is coded using *Biphase Mark Code* (BMC), which is a kind of phase modulation. The frequency of the clock is twice the bit-rate. Every bit of the original data is represented as two logical states, which, together, form a cell. The logical level at the start of a bit is always inverted to the level at the end of the previous bit. To transmit a one in this format, there is a transition in the middle of the data bit boundary. If there is no transition in the middle, the data is considered a zero.

4.9.6 S/PDIF Protocol Hierarchy

The *S/PDIF* signal format is shown in Figure 4.27. Audio data is transmitted in sequential blocks each of which contains 192 frames, each of which consists of two *subframes* that are the basic units into which digital audio data is organized.

©Copyright 2012 Cypress Semiconductor

A subframe, shown in Figure 4.28, contains a preamble pattern, an audio sample that may be up to 24 bits wide, a validity bit that indicates whether the sample is valid, a bit containing user data, a bit containing the channel status, and an even parity bit for this subframe. There are three types of preambles: X, Y and Z. Preamble Z indicates the start of a block and the start of subframe channel 0. Preamble X indicates the start of a channel 0 subframe when not at the start of a block. Preamble Y always indicates the start of a channel 1 subframe.

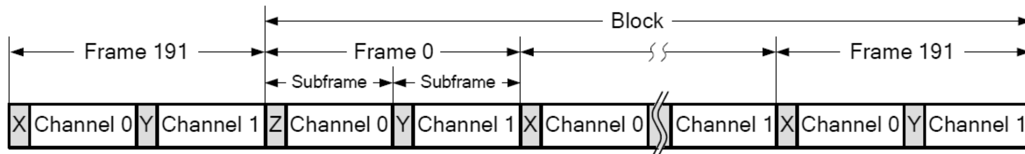


Figure 4.27: S/PDIF block format.

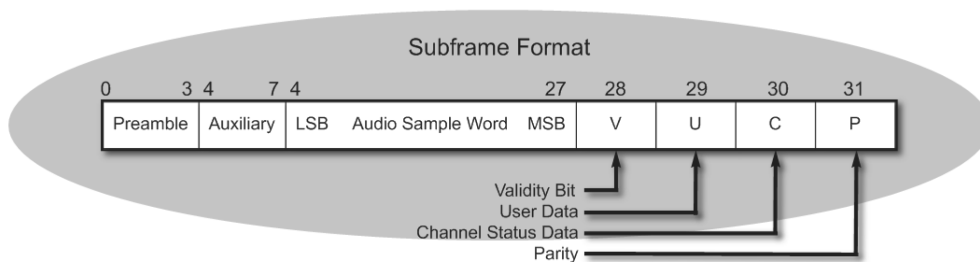


Figure 4.28: *S/PDIF* subframe format.

4.9.7 *S/PDIF* Error Handling

There are two error conditions for the *S/PDIF* component that can occur, if the audio is emptied and a subsequent read occurs (transmit underflow) or the channel status FIFO is emptied and subsequent read occurs (status underflow). If transmit underflow occurs, the component forces the constant transmission of zeros for audio data and continue correct generation of all framing and status data. Before transmission begins again, transmission must be disabled, the FIFOs should be cleared, data for transmit must be buffered, and then transmission re-enabled. This underflow condition can be monitored by the CPU using the component status bit *AUDIO_FIFO_UNDERFLOW*.[101] While the component is started, if the status underflow occurs, the component will send all 0s for channel status with the correct generation of X, Y, Z framing and correct parity. The audio data is continuous, not impacted.

To correct channel status data transmission, the component must be stopped and restarted again. This underflow condition can be monitored by the CPU using the status bit *CHST_FIFO_UNDERFLOW*. An interrupt can also be configured for this error condition. If the component doesn't manage DMA, the status data must be buffered before restarting the component. Enabling Audio data transmission has dedicated enabling. When the component is started, but not enabled, the *S/PDIF* output with channel status is generated, but the audio data is set to all zeros. This allows the *S/PDIF* receiver to lock on the component clock and the transition into the enabled state occurs at the X or Z frame.

[101] An interrupt can also be configured for this error condition.

©Copyright 2012 Cypress Semiconductor

The *SPDIF_Tx* component is implemented as a set of configured UDBs as shown in Figure 4.29. The incoming audio data is received through the system bus interface and can be provided

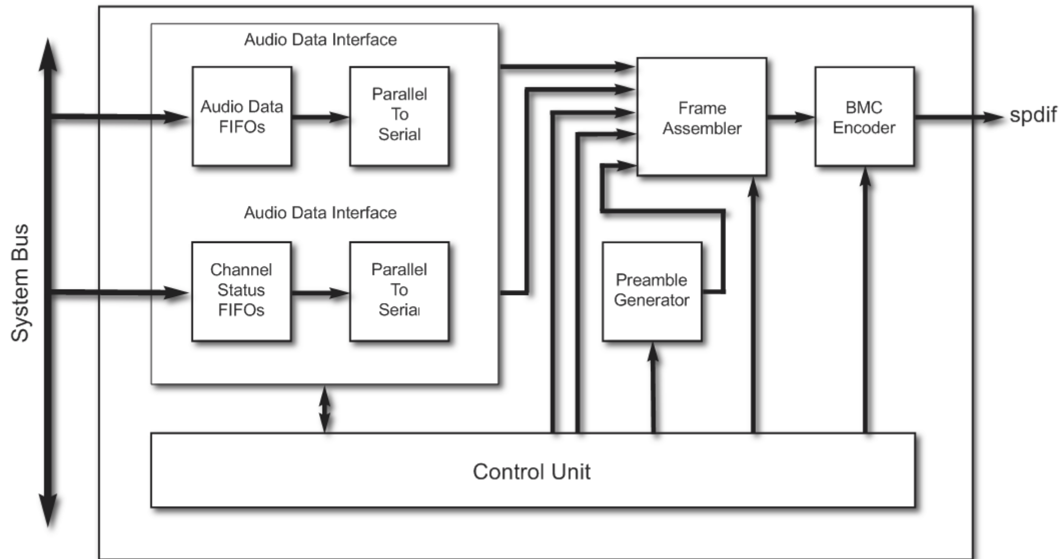


Figure 4.29: A block diagram of the implementation of SPDIF_Tx.

via the CPU, or DMA. The data is byte wide, with the least significant byte first, and is stored in an audio buffer, i.e., one or two FIFOs, depending on the component configuration). The Channel Status stream has its own dedicated interface. As with the audio data, there are two Channel Status FIFOs and the channel status is byte wide data, with the least significant byte occurring first. One byte is consumed from these FIFOs every eight samples. Both audio and status data are converted from parallel to serial form. The User Data are not defined in the S/PDIF standard and may be ignored by some receivers, so they are sent as constant zeros. The validity bit, when low, indicates the audio sample is fit for conversion to analog. This bit is sent as constant zeros. The preamble patterns are generated in the *Preamble Generator* block and are transmitted in serial form. This is all of the data required to form the SPDIF subframe structure, except for the parity bit which is calculated in the *Frame Assembler* block during assembling all the inputs in the subframe structure. The output of the *Frame Assembler* block goes to *BMC Encoder* where the data is encoded in a spdif format. The *Control Unit* block gets the control data from the *System Bus* interface and returns the status of component operation to the bus. It controls all other blocks during data transmission.

4.9.8 S/PDIF Channel Encoding

S/PDIF is a single-wire serial interface and the bit clock is embedded within the S/PDIF data stream. The digital signal is coded using *Biphase Mark Code* (BMC), which is a kind of phase modulation. The frequency of the clock is twice the bit-rate. Every bit of the original data is represented as two logical states which together form a cell. The logical level at the start of a bit is always inverted to the level at the end of the previous bit. To transmit a '1' in this format, there is a transition in the middle of the data bit boundary. If there is no transition in the middle, the data is considered a '0'.

©Copyright 2012 Cypress Semiconductor

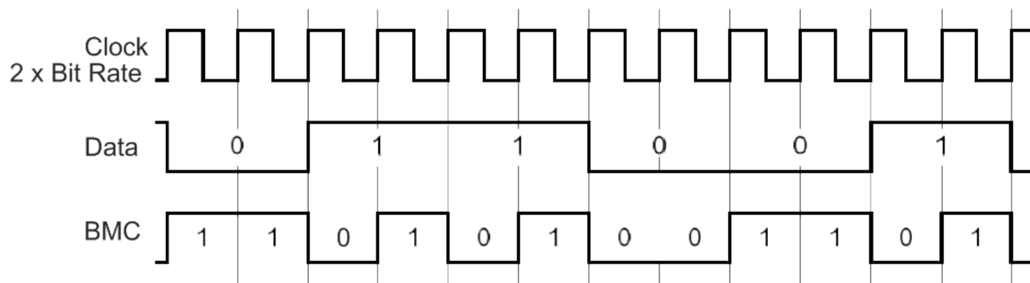


Figure 4.30: S/PDIF channel encoding timing.

4.9.9 SPDIF Registers

The transmit control and status registers, shown in Figures 4.31 and 4.32, for SPDIF are defined as follows:

- Enable/disable SPDIF_Tx component. When not enabled the component is in reset state.
- txenable: Enable/disable audio data output in the S/PDIF bit stream.

SPDIF_Tx_CONTROL_REG

Bits	7	6	5	4	3	2	1	0
value	reserved						enable	txenable

Figure 4.31: SPDIF control register.

SPDIF_Tx_STATUS_REG

Bits	7	6	5	4	3	2	1	0
value	reserved		chst1_fifo_not_full	chst1_fifo_not_full	chst1_fifo_underflow	audio1_fifo_not_full	audio0_fifo_not_full	audio_fifo_underflow

Figure 4.32: SPDIF status register.

- chst1_fifo_not_full: If set channel status FIFO 1 is not full.
- chst1_fifo_not_full: If set channel status FIFO 0 is not full.
- chst_fifo_underflow: If set channel status FIFOs underflow event has occurred.
- audio1_fifo_not_full: If set audio data FIFO 1 is not full.
- audio0_fifo_not_full: If set audio data FIFO 0 is not full.
- audio_fifo_underflow: If set audio data FIFOs underflow event has occurred.

The register value may be read by the *SPDIF_ Tx_ ReadStatus()*. Bit 3 and bit 0 of the status register are configured in Sticky mode, which is a clear-on-read. In this mode, the input status is sampled each cycle of the status register clock. When the input goes high, the register bit is set and stays set regardless of the subsequent state of the input. The register bit is cleared on a subsequent read by the CPU.

©Copyright 2012 Cypress Semiconductor

4.10 Vector CAN (VCAN)

The Vector CANbedded environment[102] consists of a number of adaptive source code components that cover the basic communication and diagnostic requirements in automotive applications, e.g., ECUs[103]. The Vector CANbedded software suite is customer specific and its operation varies according to the application and OEM[104].

This PSoC Creator VCAN component, shown in Figure 4.33, was designed for the Vector CANbedded suite to generically support the CANbedded structure, independent of the application. The PSoC3 Vector CAN component was developed to allow easy integration of the Vector certified CAN driver.[105]

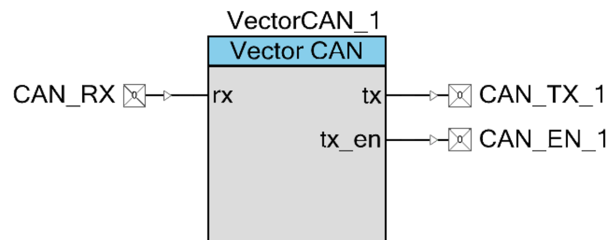


Figure 4.33: PSoC3's Vector CAN component.

PSoC Creator's VCAN component features include:

- CAN2.0 A/B protocol implementation,
- ISO 11898-1 compliant,
- Programmable bit rate up to 1 Mbps @ 8 MHz (BUS_CLK),
- Two or three wire interface to external transceiver (Tx, Rx, and Tx Enable),

and,

- Driver provided and supported by Vector.

The Vector driver uses the CAN interrupt, allowing access. The *Vector_CAN_Init()* function sets up the CAN interrupt with the interrupt service routine *CanIsr_0()* generated by the Vector CAN configuration tool.

4.10.1 Vector CAN I/O Connections

This section describes the various input and output connections for the Vector CAN component. An asterisk (*) in the list of I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

rx - CAN bus receive signal (connected to CAN RX bus of external transceiver).

tx - CAN bus transmit signal, (connected to CAN TX bus of external transceiver).

tx_en - External transceiver enable signal.[106]

[102] Vector Informatik GMBH provides a suite of software components for the automotive industry that serve as defacto standards in the automotive industry worldwide.

[103] Engine control units. (ECUs)

[104] Original equipment manufacturer (OEM).

[105] This component is used in conjunction with a CAN driver for PSoC 3 that is provided by Vector.

[106] This output displays when the Add Transceiver Enable Signal option is selected in the Configure dialog.

©Copyright 2012 Cypress Semiconductor

The Vector CAN component is connected to the *BUS_ CLK* clock signal. A minimum value of 8 MHz is required to support all standard CAN baud rates up to 1 Mbps.[107]

The Vector CAN Driver APIs use function pointers. The Keil compiler for PSoC 3 does function call analysis to determine how it can overlay function variables and arguments. When function pointers are present the compiler cannot adequately analyze the calling structure, so the *NOOVERLAY* option is selected to avoid problems that occur because of the use of function pointers. Further information on the handling of function pointers with the Keil compiler is available in the application note: Function Pointers in C51 (www.keil.com/appnotes/docs/apnt_129.asp).

In main the initialization process requires:

- i ncluding the v_inc.h file for the driver in main.c,
- enabling global interrupts, if required,
- calling the Vector_CAN_Start() function,
- calling the *CanInitPowerOn()* function (generated by the *Vector GENy* tool),

and,

- writing the necessary functionality using an API from Vector CAN and generated by the *Vector GENy* tool.

4.10.2 Vector CAN API

PSoC Creator's Vector CAN component can be configured under software control as summarized in Figure 4.11. By default, PSoC Creator assigns the instance name Vector_CAN_1 to the first

Table 4.11: Vector CAN functions supported by PSoC Creator.

Function	Description
Vector_CAN_Start()	Initializes and enables the Vector CAN component using the Vector_CAN_Init() and Vector_CAN_Enable() functions.
Vector_CAN_Stop()	Disables the Vector CAN component.
Vector_CAN_GlobalIntEnable()	Enables Global Interrupts from CAN Core.
Vector_CAN_GlobalIntDisable()	Disables Global Interrupts from CAN Core.
Vector_CAN_Sleep()	Prepares the component for sleep.
Vector_CAN_Wakeup()	Restores the component to the state when Vector_CAN_Sleep() was called.
Vector_CAN_Init()	Initializes the Vector CAN component based on settings in the component customizer. Sets up the CAN interrupt with the interrupt service routine CanIsr_0() generated by the Vector CAN configuration tool.
Vector_CAN_Enable()	Enables the Vector CAN component.
Vector_CAN_SaveConfig()	Saves the component configuration
Vector_CAN_RestoreConfig()	Restores the component configuration.

[107]The value of the *BUS_ CLK* selected in the PSoC3 project design-wide resources must be the same as the value selected in the Vector CAN driver configuration for bus timing.

©Copyright 2012 Cypress Semiconductor

instance of a component in a given design.[108] The instance name used becomes the prefix of every global function name, variable, and constant symbol. PSoC Creator provides the following application programming interface for the Vector CAN Component:

uint8 Vector_CAN_Start(void) is the preferred method to begin component operation.

uint8 Vector_CAN_Start() sets the *initVar* variable, calls the *Vector_CAN_Init()* function, and then calls the *Vector_CAN_Enable()* function.[109]

*uint8 Vector_CAN_Stop(void)*disables the Vector CAN component. Return a value indicating whether the register is written and verified.

uint8 Vector_CAN_GlobalIntEnable(void) enables global interrupts from the CAN Core.[110]

uint8 Vector_CAN_GlobalIntDisable(void) disables global interrupts from the CAN Core. Return Value: Indication whether register is written and verified.

void Vector_CAN_Sleep(void) is the preferred routine to prepare the component for sleep. *Vector_CAN_Sleep()* saves the current component state, then calls *Vector_CAN_SaveConfig()* and calls *Vector_CAN_Stop()* to save the hardware configuration.[111]

void Vector_CAN_Wakeup(void) is the preferred routine for restoring the component to the state when *Vector_CAN_Sleep()* was called. *Vector_CAN_Wakeup()* calls *Vector_CAN_RestoreConfig()* to restore the configuration. If the component was enabled before *Vector_CAN_Sleep()* was called, *Vector_CAN_Wakeup()* will also re-enable the component. Calling *Vector_CAN_Wakeup()* without first calling *Vector_CAN_Sleep()* , or *Vector_CAN_SaveConfig()* may produce unexpected behavior.

void Vector_CAN_Init (void) initializes, or restores, the component according to the customizer *Configure* dialog settings. It is not necessary to call *Vector_CAN_Init()* because *Vector_CAN_Start()* calls this function and is the preferred method to begin component operation. It sets up the CAN interrupt with the interrupt service routine *CanIsr_0()* generated by the *Vector CAN* configuration tool.

uint8 Vector_CAN_Enable(void) activates the hardware and begins component operation. It is not necessary to call *Vector_CAN_Enable()* because the Vector_CAN_Start() it, which is the preferred method to begin component operation. The return value indicates whether he register is written and verified.

void Vector_CAN_SaveConfig(void) saves the component configuration and nonretention registers, saves the current component parameter values, as defined in the *Configure* dialog or as modified by appropriate APIs. [112]

void Vector_CAN_RestoreConfig(void) restores the component configuration and nonretention registers, restores the component parameters to calling *Vector_CAN_Sleep()*.[113] The global variable, *Vector_CAN_initvar,* is defined in Table 4.12.

[108]The instance can be renamed to any unique value that follows PSoC Creator's syntactic rules for identifiers.
[109]Returns whether the register has been written and verified.
[110]The return value indicates whether or not the register has been written to and verified.
[111] *Vector_CAN_Sleep()* should be called before *CyPmSleep()* or *CyPmHibernate()*.
[112]This function is called by the *Vector_CAN_Sleep()* function.
[113]Calling this function without first calling the *Vector_CAN_Sleep()* or *Vector_CAN_SaveConfig()* may produce unexpected behavior.

©Copyright 2012 Cypress Semiconductor

Table 4.12: The global variable, *Vector_ CAN_ initVar*.

Variable	Description
Vector_CAN_initVar	Vector_CAN_initVar indicates whether the Vector CAN has been initialized. The variable is initialized to 0 and set to 1 the first time Vector_CAN_Start() is called. This allows the component to restart without reinitialization after the first call to the Vector_CAN_Start() routine. If reinitialization of the component is required, then the Vector_CAN_Init() function can be called before the Vector_CAN_Start() or Vector_CAN_Enable() function.

4.11 Inter-IC Sound Bus (I2S)

The Integrated Inter-IC Sound Bus (I2S) is a serial bus interface standard used for connecting digital audio devices together and based on a specification[114] developed by Philips Semiconductor. PSoC Creator's I2S component provides a serial bus interface for stereo audio data, is used primarily by audio ADC and DAC components and operates in master mode only. This component is bidirectional and therefore capable of functioning as a transmitter (Tx) and a receiver (Rx). The number of bytes used for each sample, whether for the right or left channel, is the minimum number of bytes to hold a sample.

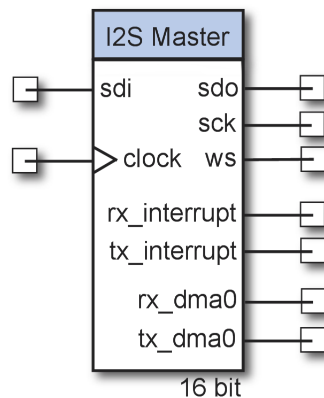


Figure 4.34: The Inter-IC Sound Bus (I2S).

I2C features include:

- 8-32 data bits per sample.
- 16, 32, 48, 64-bit word select period:6.144 MHz.
- Data rates up to 96 kHz.
- DMA support.
- Independent right and left channel FIFOs or interleaved stereo FIFOs.
- Independent enable of Tx and Rx.
- Tx and Rx FIFO interrupts.

4.11.1 Functional Description of the I2S Component

Left/Right and Rx/Tx Configuration - The configuration for the Left and Right channels, viz., the Rx and Tx direction, number of bits, and word-select period, are identical. If the application

[114] I2S bus specification; February 1986, revised June 5, 1996.

©Copyright 2012 Cypress Semiconductor

must have different configurations for Rx and Tx, then two unidirectional component instances should be used.

Data Stream Format

The data for Tx and Rx is independent byte streams that are packed with the most significant byte first and the most significant bit in bit 7 location of the first word. The number of bytes used for each sample, for the right or left channel, is the minimum number of bytes to hold a sample. Any unused bits will be ignored on Tx, and will be 0 on Rx. The data stream for one direction can be a single byte stream, or it can be two byte streams. In the case of a single byte stream, the left and right channels are interleaved with a sample for the left channel first followed by the right channel. In the two-stream case, the left and right channel byte streams use separate FIFOs.

DMA

The I2S has a *continuous interface*, i.e., it requires an uninterrupted stream of data. For most applications, this requires the use of DMA transfers to prevent the underflow of the Tx direction, or the overflow of the Rx direction. The I2S can drive up to two DMA components for each direction. PSoC Creator's DMA Wizard can be used to configure DMA operation as defined in Table 4.13.

Table 4.13: DMA and the I2S Component

Name of DMA Source/Destination in the DMA Wizard	Direction	DMA Request Signal	DMA Request Signal	Description
I2S_RX_FIFO_0_PTR	Source	rx_dma0	Level	Receive FIFO for Left or Interleaved Channel
I2S_RX_FIFO_1_PTR	Source	rx_dma1	Level	Receive FIFO for Right Channel
I2S_TX_FIFO_0_PTR	Destination	rx_dma0	Level	Transmit FIFO for Left of Interleaved Channel
I2S_TX_FIFO_1_PTR	Destination	tx_dma1	Level	Transmit FIFO for Right Channel

4.11.2 Tx and Rx Enabling

The Rx and Tx directions have separate enables. When not enabled, the Tx direction transmits all 0 values, and the Rx direction ignores all received data. The transition into, and out of, the enabled state occurs at a word select boundary such that a left/right sample pair is always transmitted, or received.

©Copyright 2012 Cypress Semiconductor

4.11.3 I2S Input/Output Connections

The I/O connections for the I2S component are:

- *sdi* - Serial data input.[115]
- *clock* - The clock rate must be two times the desired clock rate for the output serial clock (SCK). e.g., to produce 48-kHz audio with a 64-bit word select period, the clock frequency would be: 2 x 48 kHz x 64 = 6.144 MHz.
- *sdo* - Serial data output. Displays if the Tx option is selected for the Direction parameter.
- *sck* - Output serial clock.
- *ws* - Word select output indicates the channel being transmitted.
- *rx_interrupt* - Rx direction interrupt.[116]
- *tx_interrupt* - Tx direction interrupt.[117]
- *rx_DMA0* - Rx direction DMA request for FIFO 0 (Left or Interleaved).[118]
- *rx_DMA1* - Rx direction DMA request for FIFO 1 (Right).[119] Displays if Rx DMA under the DMA Request parameter and Separated L/R under the Data Interleaving parameter for Rx are selected.
- *tx_DMA0* - Tx direction DMA request for FIFO 0 (Left or Interleaved).[120]
- *tx_DMA1* - Tx direction DMA request for FIFO 1 (Right).[121]

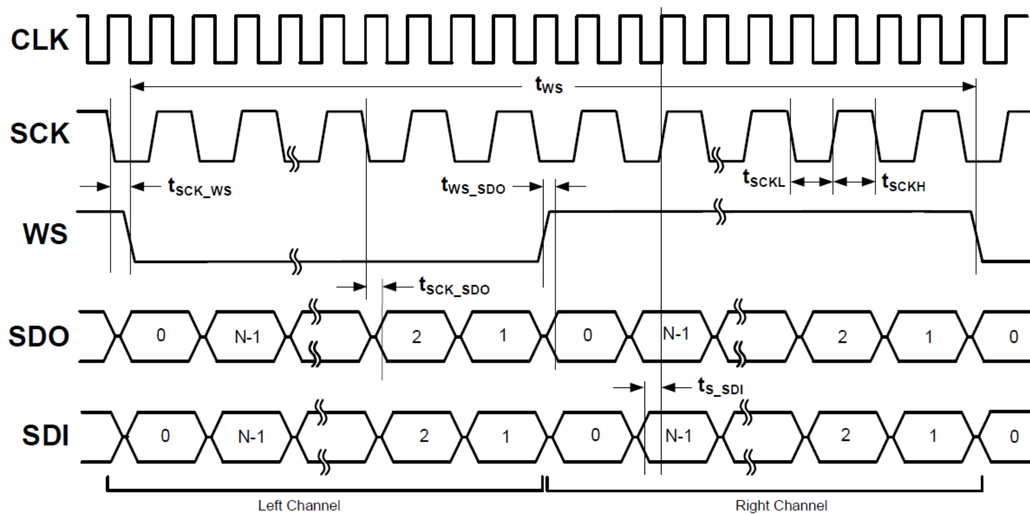


Figure 4.35: I2S data transition timing diagram

[115] If this signal is connected to an input pin, the *Input Synchronized* selection for this pin should be disabled. This signal should already be synchronized to *SCK*, so delaying the signal with the input pin synchronizer could cause the signal to be shifted into the next clock cycle.
[116] Displays if an Rx option for the Direction parameter has been selected.
[117] Displays if a Tx option for the Direction parameter is selection.
[118] Displays if Rx DMA under the DMA Request parameter is selected.
[119] Displays if Rx DMA under the DMA Request parameter and Separated L/R under the Data Interleaving parameter for Rx are selected.
[120] Displays if Tx DMA under the DMA Request parameter is selected.
[121] Displays if Tx DMA under the DMA Request parameter and Separated L/R under the Data Interleaving parameter for Tx are selected.

©Copyright 2012 Cypress Semiconductor

4.11.4 I2S Macros

By default, the *PSoC Creator Component* catalog contains three *Schematic Macro* implementations for the *I2S* component. These macros contain the *I2S* component already connected to digital pin components. The *Input Synchronized* option is unchecked on the SDI pin and the generation of APIs for all of the pins is turned off. The Schematic Macros use the I2S component, configured for Rx only, Tx only, and both Rx and Tx directions, as shown in Figures 4.36 and 4.37.

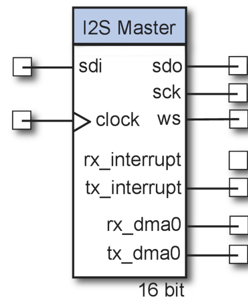


Figure 4.36: I2S Tx and Rx.

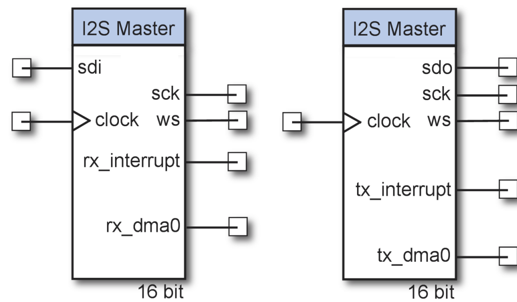


Figure 4.37: I2S Rx only and I2S Tx only.

4.11.5 I2S APIs

- *void I2S_Start(void)* starts the I2S interface, enables the Active mode power template bits, or clock gating, as appropriate. Starts the generation of the *sck* and *ws* outputs. The Tx and Rx directions remain disabled.
- *void I2S_Stop(void)* disables the *I2S* interface and the Active mode power template bits or clock gating as appropriate. sets the *sck* and *ws* outputs to 0. disables the Tx and Rx directions and clears their FIFOs.
- *void I2S_EnableTx(void)* enables the Tx direction of the I2S interface.[122]
- *void I2S_DisableTx(void)* disables the Tx direction of the I2S interface.[123]
- void I2S_EnableRx(void) enables the Rx direction of the I2S interface.[124]

[122] Transmission begins at the next word select falling edge.
[123] Transmission of data stops and a constant 0 value is transmitted at the next word select falling edge.
[124] Data reception begins at the next word select falling edge.

©Copyright 2012 Cypress Semiconductor

- *void I2S_DisableRx(void)* disables the Rx direction of the I2S interface.[125]
- *void I2S_SetRxInterruptMode(uint8 interruptSource)* sets the interrupt source for the I2S Rx direction interrupt. Multiple sources may be ORed.

Table 4.14: I2S Rx Interrupt Source

I2S Rx Interrupt Source	Value
RX_FIFO_OVERFLOW	0x01
RX_FIFO_0_NOT_EMPTY	0x02
RX_FIFO_1_NOT_EMPTY	0x04

4.11.6 I2S Error Handling

Two error conditions can occur if the transmit FIFO is empty, and a subsequent read occurs, i.e., a transmit underflow, or the receive FIFO is full and a subsequent write occurs, i.e., a receive overflow. If the transmit FIFO becomes empty, and data is not available for transmission while transmission is enabled, i.e., a Transmit underflow, the component will force the constant transmission of 0s. Before transmission begins again, transmission must be disabled, the FIFOs should be cleared, data for transmit must be buffered, and then transmission re-enabled. The CPU can monitor this underflow condition using the transmit status bit *I2S_TX_FIFO_UNDERFLOW*. An interrupt can also be configured for this error condition. While reception is enabled, if the receive FIFO becomes full and additional data is received (Receive overflow), the component stops capturing data. Before reception begins again, reception must be disabled, the FIFOs should be cleared, and then reception re-enabled. The CPU can monitor this overflow condition using the receive status bit *I2S_RX_FIFO_OVERFLOW*. An interrupt can also be configured for this error condition.

4.12 Local Interconnect Network (LIN)

The LIN standard was co-developed by a set of companies involved in the automotive industry.[126] It was intended from the outset to serve as a multiplexed communication system that was much simpler than the controller area network (CAN), or the serial peripheral interface(SPI). LIN functions as a subnetwork to CAN and is based on an architecture that supports only a single masterm, and multiple slaves. It is not as robust, has smaller bandwidth/bit rate and offers less functionality than CAN, but it is much more economical. LIN targets low-cost automotive networks as a complement to the existing portfolio of automotive multiplex networks and is typically used for networking sun roof controls, rain detection systems, automatic headlight controls, door locks, interior lighting controls, etc.

The LIN specification consists of an *API specification*, a *configuration/diagnostic specification*, a *physical layer specification.* a *node capability language specification* and *protocol specification.*

[125] At the next word select falling edge, data reception is no longer sent to the receive FIFO.
[126] The original concept for the LIN protocol is attributed to Motorola but they were soon joined in supporting the new standard by Audi, BMW, Daimler Chrysler, Volkswagen and Volvo.The current version is LIN 2.0 and was issued in September 2003.

©Copyright 2012 Cypress Semiconductor

- The *API specification* describes the interface between the application program and the network.
- The *configuration/diagnostic specification* is a description of LIN services available above the data link layer associated with sending configuration and diagnostic messages. The physical layer specification defines clock tolerances, supported bit rates, etc.
- The *node capability language specification* defines the langauge format for certain types of LIN modules employed in plug and play applications.
- The *capability language specification* defines the format of the configuration file used to configure the LIN network.

The LIN system functions as an asynchronous communications system that operates without requiring a clock. Therefore, it functions as a single wire system[127] that does not require arbitration. Baud rates are limited to 20 kbits/second to avoid EMI issues. The master is responsible for determining the priority, and therefore order of message transmission. The master employs a stable clock for reference and monitors data and check bytes, while controlling the error handler. The master controls the bus and transmits *Sync Break, Sync Byte*, and *ID* data fields. Two to sixteen slaves receive/transmit data when there respective IDs are transmitted by the master.[128] Slaves can transmit 1,2,4 or 8 data bytes at a time, together with a check-byte.

The main properties of the LIN bus are:

- Data format similar to the common serial UART format,
- Safe behavior with data checksums,
- Self-synchronization of slaves on master speed,
- Single-master, multiple-slaves (up to 16 slaves),
- Single-wire (max 40 m),

and,

- Speeds up to 19.2 Kbps (choice is 2400, 9600, 19200 bps).

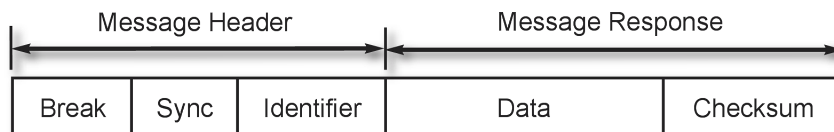


Figure 4.38: The LIN message frame.

The message frame format employed by LIN consists of a *break* containing 13 bits followed by a delimiter of one bit which alerts all of the nodes on the LIN bus and signals the start of a frame. This is immediately followed by a clock synchronization, or *sync* field (x55), that allows the slaves to adjust their respective internal baud rates to that of the bus. An message *identifier* (ID) follows the sync field that consists of a 6-bit message and a 2-bit parity field. IDs 0-59 are assigned to the signal-carrying data frames, 60-61 to the diagnostic data frames, 62 to user-defined extensions and 63 is reserved for future use[129]. The slaves *listen* for IDs and check the respective parities for which they are either a publisher or subscriber. The slave response consists of one-to-eight data bytes followed by an 8-bit checksum[130].

[127]Such systems are often referred to as *one-wire systems,* but in point of fact an additional wire is required to provide to provide a ground return for the system.
[128]It should be noted that a master can also serve as a slave.
[129]ID 63 always employs the *classic* checksum algorithm.
[130]The classic checksum algorithm is used with LIN 1.3 nodes and the enhanced checksum algorithm is used

©Copyright 2012 Cypress Semiconductor

4.12.1 LIN Slave

PSoC Creator's LIN Slave component implements a LIN 2.1 slave node on PSoC 3 and PSoC 5 devices. Options for LIN 2.0, or SAE J2602-1, compliance are also available. This component consists of the hardware blocks necessary to communicate on the LIN bus, and an API to allow the application code to easily interact with the LIN bus communication. The component provides an API that conforms to the API specified by the LIN 2.1 Specification. This component provides a good combination of flexibility and ease of use. A customizer for the component is provided that allows all of the LIN Slave parameters to be easily configured.

Supported features include:

- Automatic baud rate synchronization,
- Automatic configuration services handling,
- Automatic detection of bus inactivity,
- Customizer for fast and easy configuration,
- Editor for *.ncf/*.ldf files with syntax checking,
- Full LIN 2.1 or 2.0 Slave Node implementation,
- Fully implements a Diagnostic Class I Slave Node,
- Full transport layer support,
- Full error detection,
- Import of *.ncf/*.ldf files and *.ncf file export,

and,

- Supports compliance with SAE J2602-1 specification.

The LIN bus is based on a single wire, wired-AND, with a termination resistor placed at each node[131] for each slave, and the supply voltage ranges from 8 to 18 volts, as shown in Figure. The LIN slave component has the following I/O connections:

- RXD - a digital input terminal
- TXD - a digital output terminal which transmits the data sent via the LIN bus by the LIN node.

4.12.2 PSoC and LIN Bus Hardware Interface

A LIN physical layer transceiver device is required when the PSoC LIN slave node is connected directly to a LIN bus. In such cases, the *txd* pin of the LIN component connects to the TXD pin of the transceiver, and the *rxd* pin connects to the RXD pin of the transceiver, as shown in Figure 4.39. The LIN transceiver device is required because the PSoC's electrical signal levels are not compatible with the electrical signals on the LIN bus. Some LIN transceiver devices also have an *enable* or *sleep* input signal that is used to control the operational state of the device. The LIN component does not provide this control signal. Instead, a pin is used to output the desired signal to the LIN transceiver device, if this signal is needed.

with LIN 2.0. The enhanced checksum algorithm requires that the data values be summed and if the sum is greater than, or equal to, 256, 255 is subtracted and the result is appended to the message response.

[131] Typical resistor values are $1k\Omega$ for each each master, $30k\Omega$

©Copyright 2012 Cypress Semiconductor

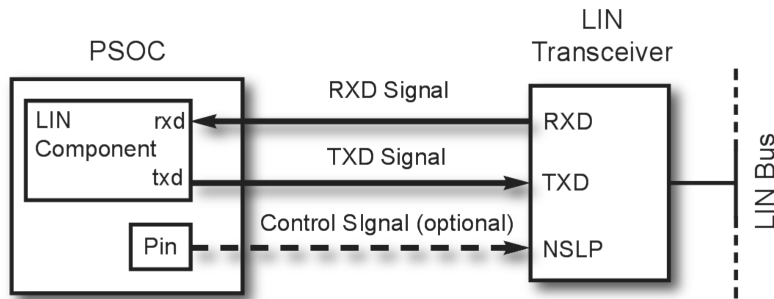


Figure 4.39: The LIN bus physical layer.

4.13 LCD (Visual Communication)

Visual displays are often an important component of an embedded system for displaying important messages, certain parameter values and/or to facilitate debugging. PSoC 3 has as many as 64 built-in segment LCD drivers that be interfaced directly with wide variety of segment, LCD, glass types. This gives it the capability to drive up to 768 pixels (16 commons x 48 segments).

The features supported by the PSoC3 LCD drivers are:

- Adjustable refresh rate from 10 Hz to 150 Hz. Configurable power modes, which allows power optimization,
- Direct drive with internal bias generation no other external hardware is required,
- Maximum 64 in-built LCD drivers (which includes both common and segment pin driver). No CPU intervention in LCD refresh,
- Static, 1/3, 1/4, 1/5 bias ratios. Supports 14-segment and 16-segment alphanumeric display, 7-segment numeric display, dot matrix, and special symbols,
- Support for both Type A and Type B waveforms,
- Support for LCD glass with up to 16 common lines,

and,

- Support for 14-segment and 16-segment alphanumeric display, 7-segment numeric display, dot matrix, and special symbols.

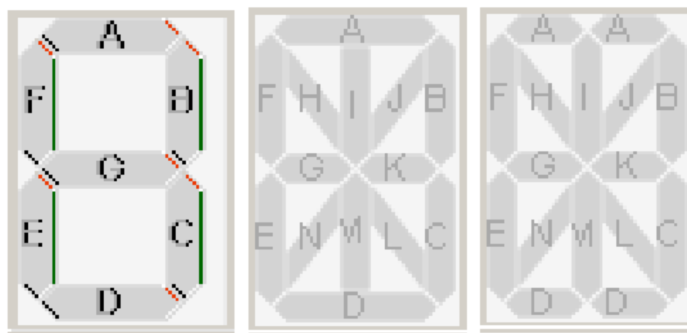


Figure 4.40: Supported LCD segment types.

©Copyright 2012 Cypress Semiconductor

PSoC Creator's LCD component is based on a set of library routines that facilitate the use of one, two or four-line LCD modules that employ the *Hitachi HD44780* LCD display driver, 4-bit protocol. The Hitachi interface has proven to be a widely adopted standard for driving LCD displays of the type shown in Figure 4.41. Each of the 32 segments shown in the figure consist

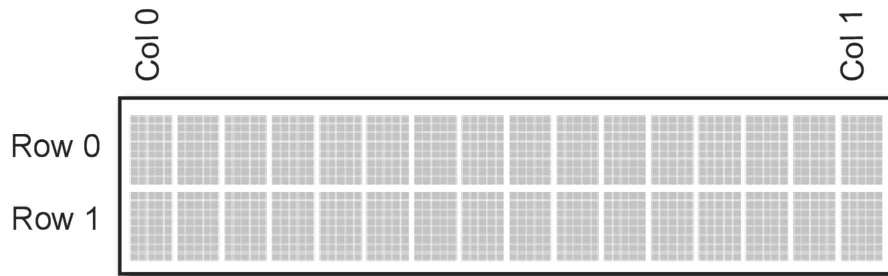


Figure 4.41: Hitachi 2x16 LCD.

of an array of 40 elements (8x5). This particular LCD is capable of displaying two rows of 16 characters[132] each and limited graphic displays. Seven logical port pins are used to transmit data bits 0-3, LCD enable[133], register select[134] and read/not write[135] to the display's integral hardware controller as shown in Table 4.15. The *LCD_Char_Position()* function manages display addressing as follows: row zero, column zero is in the upper left corner with the column number increasing to the right, as shown in Figure 4.41.[136]

Table 4.15: Logical to Physical LCD Port Mapping

Logical Port Pin	LCD Module Pin	Description
LCDPort_0	DB4	Data Bit 0
LCDPort_1	DB5	Data Bit 1
LCDPort_2	DB6	Data Bit 2
LCDPort_3	DB7	Data Bit 3
LCDPort_4	E	LCD Enable
LCDPort_5	RS	Register Select
LCDPort_6	R/!W	Read/not Write

[132]Custom character sets are also supported.
[133]Strobed to confirm new data available.
[134]Select for either data or control input.
[135]Toggle for polling the LCD's ready bit.
[136]In a four-line display, writing beyond column 19 of row 0 can result in row 2 being corrupted because the addressing maps row 0, column 20 to row 2, column 0. This is not an issue in the standard 2x16 Hitachi module.

©Copyright 2012 Cypress Semiconductor

4.13.1 Resistive Touch

PSoC Creator's resistive touchscreen component[137] is used to interface with a 4-wire resistive touch screen. The component provides a method to integrate and configure the resistive touch elements of a touchscreen with the emWin[138] Graphics library.[67] It integrates hardware-dependent functions that are called by the touchscreen driver supplied with emWin, when polling the touch panel.

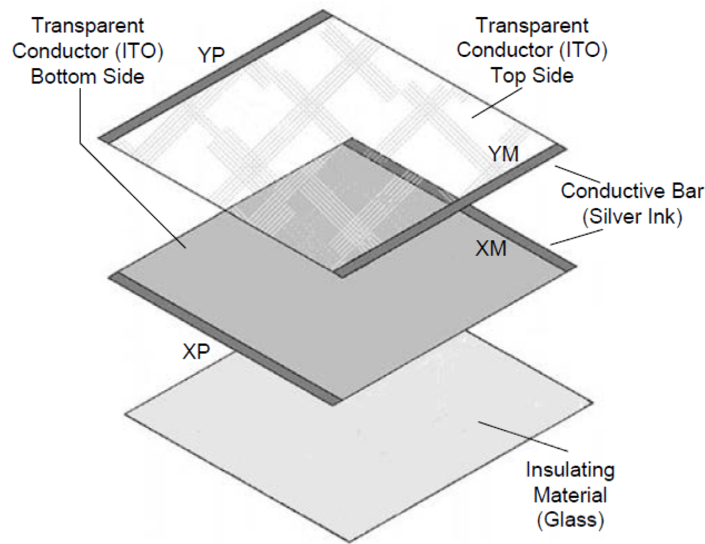


Figure 4.42: Resistive touchscreen construction.

The supported I/O connections are *xm*, *xp*, *ym*, *yp* where

- xm is a digital I/O connection and designated as signal x- with low being active.
- xp is an analog/digital output connection designated as signal x+ from the x axis of the

The point of contact divides each layer in a series resistor network with two resistors, and a connecting resistor between the two layers. By measuring the voltage at this point, information about the position of the contact point orthogonal to the voltage gradient can be obtained. To get a complete set of coordinates, a voltage gradient must be applied once in the vertical and then in the horizontal direction. First, a supply voltage is applied to one layer and a measurement made of the voltage across the other layer; then the supply voltage is applied to the other layer and the opposite layer voltage is measured. When in touch mode, one of the lines is connected to detect touch activity.

4.13.2 Measurement Methods

As shown in Figure 4.43, a touch by a finger, or a stylus, can be uniquely defined by the measurement of three parameters, viz., the x-position, y-position and a third parameter related to the touch pressure. The latter measurement makes it possible to differentiate between finger and

[137]This component provides a 4-wire resistive touch screen interface to read the touchscreen coordinates and measure the screen resistance It provides access to the functionality of the SEGGER emWin graphics library for translation of resistance to screen coordinates.

[138]emWin is a product of SEGGER Microcontroller that was designed to function as an efficient graphical user interface that is processor- and graphical LCD controller-independent. (http://www.segger.com/embedded-software.html)

©Copyright 2012 Cypress Semiconductor

stylus contacts. The conductive bars are located on the opposite edges of the panel, as shown. The voltage applied to the layer produces a linear gradient across this layer. The conducting layers are oriented so that the conducting bars are orthogonal to each other, and voltage gradients in the respective layers are also orthogonal. An equivalent circuit for a resistive touchscreen, can be based on treating the conductive layers as resistors placed between the conductive bars in the corresponding layers. When the touchscreen is touched, a resistive connection is formed between the two layers, as shown in Figure 4.43.

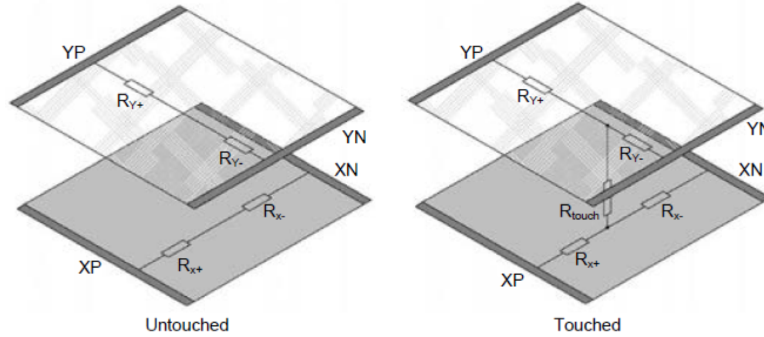


Figure 4.43: Resistive touchscreen equivalent circuit.

To measure a 4-wire touch sensor, a voltage (VCC) is applied to a conductive bar on one of the layers and the other conductive bar on the same layer is grounded, see Figure 4.43. This creates a linear voltage gradient in this layer. One of the conductive bars in the other layer is connected to an ADC through a large impedance. The ADC reference is set to VCC, which makes the ADC range from 0 to the max ADC value. When the screen is touched, the ADC reading corresponds to the position on one of the axes. To obtain the second coordinate, the other layer must be powered and read by the ADC. VCC, GND, Analog hi-Z, and ADC input are switched between the two layers, as shown in the y-position measurement in Figure 4.43. The second ADC reading corresponds to the position on the other axis. Finally, to obtain the touch pressure, two measurements of the cross-layer resistance are required. VCC is applied to a conductive bar on one of the layers while a conductive bar on the other layer is grounded. The voltages on the unconnected bars is then measured, as shown in Figures 4.44 c) and d), respectively.

Examination of Figure 4.44 a) shows that an equivalent circuit for this case is given by

$$\frac{x}{AD_{max}} = \frac{v_{in}}{v_{ref}} = \frac{v_{in}}{v_{cc}} = \frac{iR_{x-}}{i(R_{-x} + R_{x+})} = \frac{R_{x-}}{R_{-plate}} \tag{4.2}$$

where $x = ADC$ value when the ADC input voltage is equal to v_{in}, $AD_{max} = 2^{ADC_resolution}$, v_{ref} is the ADC reference voltage and R_{x_plate} is given by

$$R_{x_plate} = R_{x_} + R_{x+} \tag{4.3}$$

A similar analysis of Figure 4.44 b), c) and d) gives

$$\frac{y}{AD_{max}} = \frac{R_{y-}}{R_{y-} + R_{y+}} = \frac{R_{y-}}{R_{y_plate}} \tag{4.4}$$

$$\frac{z_1}{AD_{max}} = \frac{R_{x-}}{R_{x-} + R_{touch} + R_{y+}} \tag{4.5}$$

©Copyright 2012 Cypress Semiconductor

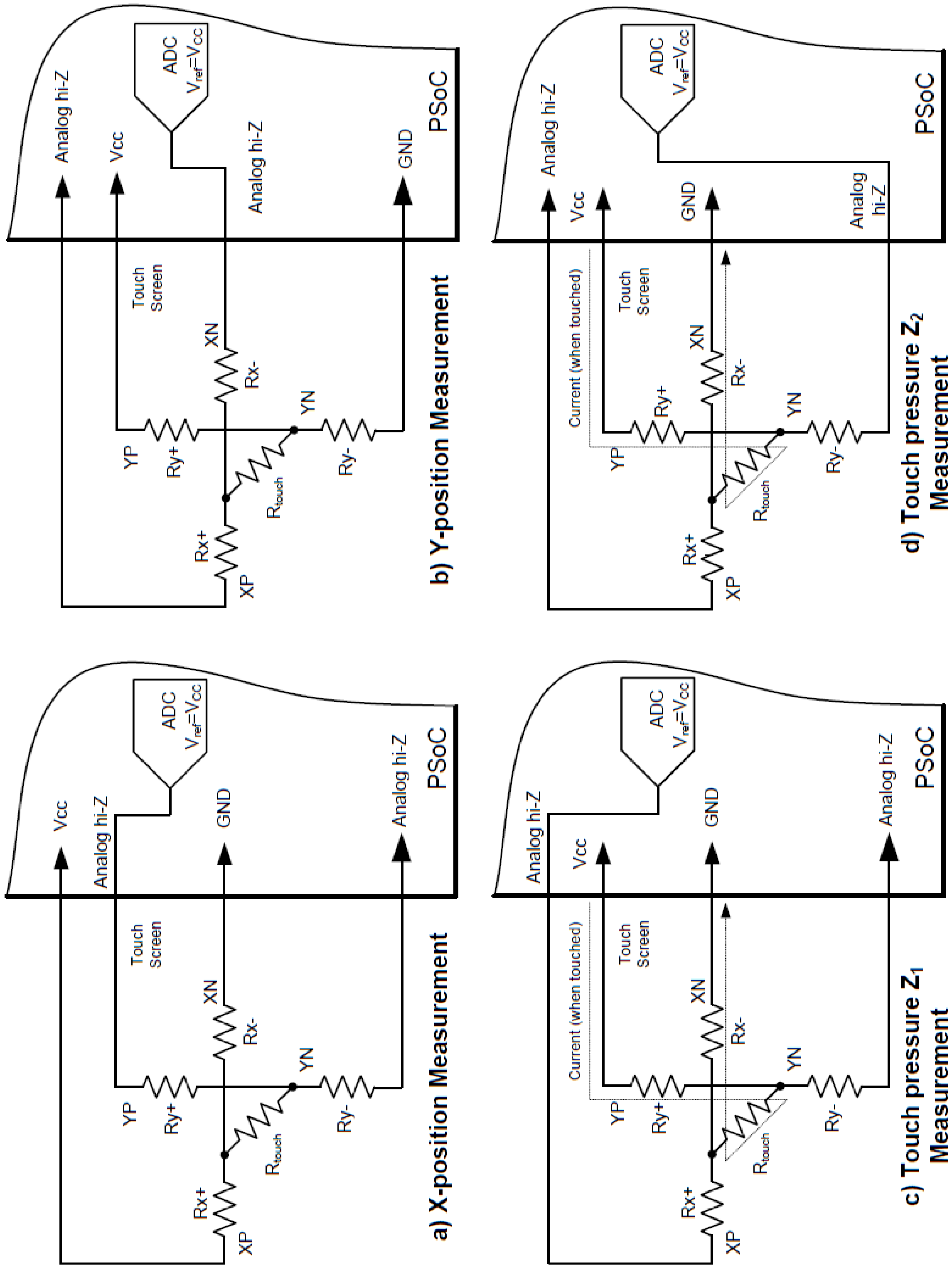


Figure 4.44: Resistive touchscreen equivalent circuit models.

©Copyright 2012 Cypress Semiconductor

and,

$$\frac{z_2}{AD_{max}} = \frac{R_{x-} + R_{touch}}{R_{x-} + R_{touch} + R_{y+}} \tag{4.6}$$

Combining these equations yields

$$R_{touch} = R_{x_plate}\left[\frac{x}{2^{ADC_resolution}}\right]\left[\frac{z_2}{z_1} - 1\right] \tag{4.7}$$

and,

$$R_{touch} = R_{x_plate}\left[\frac{x}{2^{ADC_resolution}}\right]\left[\frac{2^{ADC_resolution}}{z_1} - 1\right] - R_{y_{plate}}\left[1 - \frac{y}{2^{ADC_resolution}}\right] \tag{4.8}$$

Equation (4.7) assumes that x_{plate}, x, z_1 and z_2 are known. R_{touch} can also be determined by evaluation of Equation (4.8), assuming that the values of x_{plate} and y_{plate} are known. A flowchart is shown in Figure 4.45 that represents the steps required to measure the touchscreen parameters.

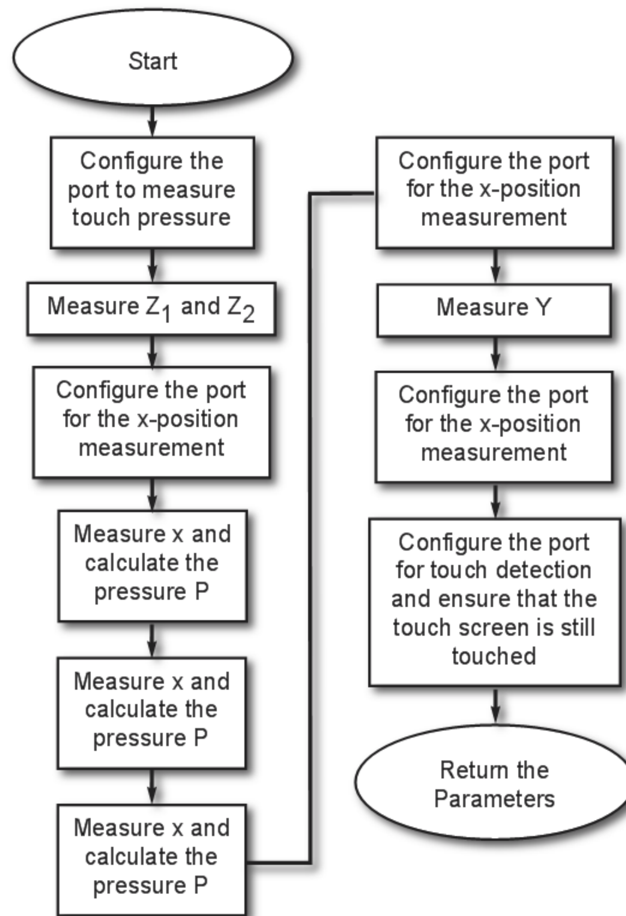


Figure 4.45: Touchscreen flowchart for parameter measurement.

©Copyright 2012 Cypress Semiconductor

4.13.3 Application Programming Interface

PSoC Creator supports the following functions for resistive touchscreens:

- void ResistiveTouch_Start(void) calls the *ResistiveTouch_Init()* and *ResistiveTouch_Enable()* APIs.
- *void ResistiveTouch_Init(void)* calls the *Init* functions of the DelSig ADC or SAR ADC and AMux components.
- *void ResistiveTouch_Enable(void)* enables the DelSig ADC or SAR ADC and the AMux components.
- *void ResistiveTouch_Stop(void)* stops the DelSig ADC or SAR ADC and the AMux components.
- *void ResistiveTouch_ActivateX(void)* configures the pins for measurement of X-axis. void ResistiveTouch_ActivateY(void) configures the pins for measurement of Y-axis.
- *int16 ResistiveTouch_Measure(void)* returns the result of the A/D converter.
- *uint8 ResistiveTouch_TouchDetect(void)* detects a touch on the screen.
- void ResistiveTouch_SaveConfig(void) saves the configuration of the DelSig ADC or SAR ADC.
- *void ResistiveTouch_RestoreConfig(void)* restores the configuration of the DelSig ADC or SAR ADC.
- *void ResistiveTouch_Sleep(void)* prepares the DelSig ADC or SAR ADC for low-power modes by calling *SaveConfig* and *Stop* functions.
- void ResistiveTouch Wakeup(void) restores the DelSig ADC or SAR ADC after waking up from a low-power mode.

	XP	XM	YP	YM
Touch	Res Pullup	Digital Hi-Z	Analog Hi-Z	Strong Drive
x- coordinate	Strong Drive	Strong Drive	Analog Hi-Z	Analog Hi-Z
y- coordinate	Analog Hi-Z	Analog Hi-Z	Strong Drive	Strong Drive

Figure 4.46: Pin configurations for measurement of the touch coordinates.

- void LCD_Char_Start(void) - initializes the LCD hardware module as follows:
  - Enables 4-bit interface
  - Clears the display
  - Enables auto cursor increment
  - Resets the cursor to start position
  - If defined in PSoC Creator's Customizer GUI, a custom LCD character set is also loaded.
- *void LCD_Char_Stop(void)* - turns off the LCD display screen.
- *void LCD_Char_PrintString(char8 * string)* - writes a null-terminated string of characters to the screen beginning at the current cursor location.aaa
- *void LCD_Char_Position(uint8 row, uint8 column)* - moves the cursor to the specified location.

©Copyright 2012 Cypress Semiconductor

- *void LCD_ Char_ WriteData(uint8 dByte)* - writes data to the LCD RAM in the current position. The position is then incremented/decremented depending on the specified entry mode.
- *void LCD_ Char_ WriteControl(uint8 cByte)* - writes a command byte to the LCD module.[139]
- *void LCD_ Char_ ClearDisplay(void)* - clears the contents of the screen, resets the cursor location to be row and column zero and calls *LCD_ Char_ WriteControl()* with the appropriate argument to activate the display.

4.13.4 Capacitive Touchscreens

A capacitive touchscreen[54] can be used, as an alternative to resistive touchscreens , and consists of an insulator, e.g., glass, coated with a transparent conductor such as indium tin oxide (ITO). Because a human body is also an electrical conductor, touching the surface of the screen results in a distortion of the screen's electrostatic field that is measurable as a change in the screen's capacitance. The location that is touched can be determined by a variety of technologies, and subsequently, can then be sent to the controller for processing. Unlike its resistive counterpart, a capacitive touchscreen is not compatible with most types of electrically insulating materials, e.g., gloves. A special capacitive stylus, or glove with finger tips that generate static electricity is required. This disadvantage especially affects a capacitive touchscreen's usability in consumer electronics, such as touch tablet PCs and capacitive smart phones in cold weather.

Surface capacitance applications have only one side of the insulator that is coated with a conductive layer. A small voltage is applied to that layer, resulting in a uniform electrostatic field. When a conductor, such as a human finger, touches an uncoated surface, a capacitor is formed, dynamically. The sensor's controller can determine the location of a touch, indirectly, based on the change in the capacitance, as measured from the four corners of the surface. The controller has a limited resolution, is prone to false signals from parasitic capacitive couplings, and requires calibration during manufacturing. It is therefore most often used in simple applications such as industrial controls and kiosks.

Projected capacitive touch (*PCT*) is a capacitive technology, consisting of an insulator, such as glass or foil, coated with a transparent conductor, e.g., copper, antimony tin oxide (ATO), nanocarbon, or indium tin oxide (ITO), that permits more accurate and flexible operation by etching, rather than coating, a conductive layer. An X-Y grid is formed, either by etching a single layer to form a grid pattern of electrodes, or by etching two separate, perpendicular layers of a conductive material, with parallel lines or tracks to form a grid, comparable to that of the pixel grid found in many LCD displays. A higher resolution PCT allows operation without a direct contact.

PCT is a more robust solution than resistive touch technology because the PCT layers are made from glass. Depending on the implementation, an active, or passive, stylus can be used instead of, or in addition to, a finger. This is common with point-of-sale devices that require a signature capture. Gloved fingers may, or may not, be sensed, depending on the implementation and gain settings. Conductive smudges and similar interference on the panel surface can interfere with the performance. Such conductive smudges come mostly from sticky, or perspiring fingertips, especially in high humidity environments. Collected dust, which adheres to the screen due to the moisture from fingertips, can also be a problem.

There are two types of PCT: *Self Capacitance* and *Mutual Capacitance.* If a finger. which

[139] Different LCD models can have their own commands.

©Copyright 2012 Cypress Semiconductor

is also a conductor, touches the surface of the screen, the local electrostatic field, created by the application of of a voltage to each row and column, distorts the local electrostatic field and hence the effective capacitance, and this distortion can be measured to obtain the finger coordinates. Currently, mutual capacitive technology is more common than PCT technology. In mutual capacitive sensors, there is a capacitor at every intersection of each row and each column, e.g., a 16-by-14 array has 224 independent capacitors.

A voltage is applied to the rows or columns so that a finger, or conductive stylus, close to the surface of the sensor changes the local electrostatic field, thereby reducing the mutual capacitance. The capacitance change at each point on the grid can be measured to accurately determine the touch location by measuring the voltage on the other axis. Mutual capacitance allows multi-touch operation where multiple fingers, palms, or styli, can be accurately tracked at the same time. Self-capacitance sensors can have the same X-Y grid as mutual capacitance sensors, but the columns and rows operate independently. With self-capacitance, the capacitive load of a finger is measured as a current on each column, or row, electrode. This method produces a stronger signal than the mutual capacitive method, but it is unable to detect accurately more than one finger, which results in *ghosting*, or misplaced location sensing.

4.14 Exercises

1. Give examples of when each of the communications protocols discussed in this chapter might be used to provide the most efficient and cost effective transmission channel.

2. Explain why a twisted pair of conductors is used when deploying communications protocols such as USB. What is the significance of the use of 90 ohm impedance wiring in such cases? Can 50 ohm, or 72 ohm, impedance cable be used instead? If not, why not? And if so, what are the constraints on their use, if any?

3. Calculate the CRC for a string of bytes consisting of 01010101, 00000000, 11111111, 00001111, 00000011, 01010101, 11110000 and 10101010.

4. Explain the advantages and disadvantages of using parity checks, versus cyclic redundancy, to insure data integrity.

5. Prepare a table comparing each of the communication protocols discussed in this chapter with respect to parameters such as path differences, transmission speeds, handshaking techniques, multiple master, multiple slave support, error detection methods, etc.

6. When transmitting multiple bits in the form of bytes, are parallel transmission paths always capable of transmitting data faster than serial paths? If not, give an example of a situation for which serial transmission can be faster than parallel transmission.

7. Explain how arbitration works for each of the protocols discussed in this chapter, if applicable. In particular, treat the case of multiple masters, and slaves, operating in the same network.

8. Estimate the propagation delay of individual bits when transmitted in serial fashion over a distance of 5 meters, 30 meters and 1000 meters. State all of your assumptions.

9. What are the advantages of the USB protocol that have have led to its largely replacing the once ubiquitous RS232 protocol?

10. Why do many automotive and other applications often employ multiple communications protocols in the same environment, e.g., why are CAN, LIN and FlexRay sometimes employed in the same vehicle?

©Copyright 2012 Cypress Semiconductor

©Copyright 2012 Cypress Semiconductor

Chapter 5

Programmable Logic

5.1 Programmable Logic Devices

Embedded system have become truly ubiquitous, outnumbering their PC counterparts by at least one to two orders of magnitude. In the most common incarnation, embedded systems perform a function, or typically a limited set of functions, to which they are tightly constrained. They are expected to be fast, inexpensive, highly responsive, require minimal power, etc. In addition many embedded systems are often required to be aware of changes in their operating environment and to make the necessary adjustments, if any, in order to maintain high performance. Computations and decisions made by embedded systems are to occur in real time and not introduce any degradations in overall system performance.

The designer is expected to produce a design capable of meeting, or exceeding, the design specifications while operating well within the constraints imposed by cost effectiveness, small size, low power consumption, etc. This inevitably results in the designer having to optimize multiple facets of the design in order to produce a system that is overall highly optimized with respect to the key design criteria. Typical metrics include, materials cost, size, robustness, power consumption, manufacturing costs, critical component availability, time-to-market, development cost, etc. In addition to creating an optimized design, the designer must also take into account the various design metrics and arrive at a system which represents the best set of tradeoffs with respect to these metrics.

A further complication is introduced by the fact that an embedded system is a synthesis of hardware *and* software. Although an embedded system can be subjected to rigorous testing, the software component is often difficult, if not impossible, to thoroughly test prior to release to the market. An additional complication is that often the designer must have significant expertise in both software and hardware design and implementation in order to effectively optimize the hardware and software aspects of the system. Finally, the designer also needs to have technical competence in a variety of technologies, e.g., optics, analog/digital subsystems, sensors, microcontrollers, ADC/DAC technology, communications protocols, etc.

As discussed briefly in Chapter 1, programmable logic devices allow designers to employ existing generic devices that include the ability to either create internal connections, or destroy existing connections, as may be required, to implement the required functionality. One of the most common of such devices is the field programmable gate array (FPGA). Although PLDs can reduce nonrecurring engineering (NRE[1]) costs and have the additional advantage that custom

[1] Non-recurring engineering costs are one-time costs related to the development, engineering testing and design

devices can be made available almost instantly, they have the disadvantages of often requiring more power, being larger devices, potentially slower than their production counterpart would be, and can be significantly more expensive. While they can be mask-programmed, i.e., factory programmed, it is usually not practical to use this type of technology, except when large volumes of devices are required. Field programmable devices can be rapidly produced in small quantities and allow the designer to make "mid-course" corrections in the field. FPGA-based designs can also be arbitrarily complex and are typically highly scalable.[2]

The hierarchy of programmable, solid state, logic devices is shown in Figure 5.1.

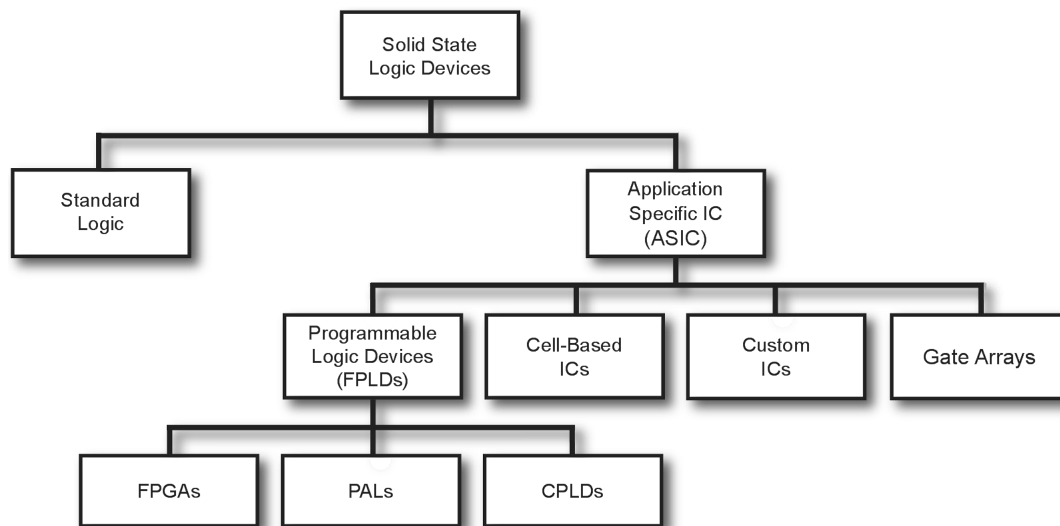


Figure 5.1: Hierarchy of programmable logic devices.

Fortunately, and as a practical matter, it often matters little whether some aspect of an embedded system is implemented in hardware, software or some synthesis of both. As a rule the decision as to what is to be implemented in software and what is to be implemented in hardware is based on any one of a number of tradeoffs in terms of cost, power requirements, size, ability to rapidly adapt to changing market needs, etc. In cases in which the anticipated production may represent relatively low volume in terms of number of units, time-to-prototype, time-to-market, NRE and/or the ability to adapt to changing market needs are major concerns, PLDs can provide an excellent alternative to customized ICs.

PLDs typically consist of large numbers of flip-flops and gates[3] which can be connected under software control in arbitrarily complex configurations to provide specific logic functionality. As discussed briefly in Chapter 1, there are three basic types of PLDs, viz., simple PLDs (SPLDs), complex PLDs (CPLDs) and field programmable PLDs typically referred to as field programmable gate arrays (FPGAs). The programmable logic arrays, which are one type of PLD, employ fuses which can be permanently rendered in an open state by special types of hardware/software programmers. The generic array logic (GAL) is a PLD similar to the PAL except that it is reprogrammable and they are relatively high-speed devices that are compatible with both 3.3

of a system or product.

[2]As an example of complexity and scalability capability of FPGAs, designs exist that have the functionality of 1000 distinct cores for use in extremely high speed image processing.

[3]PLDs typically have hundreds or thousands of AND, OR and NOT gates and in some cases flip-flops that can be programmatically interconnected to provide a wide variety of devices.

and 5 volt logic. In addition to array logic, both GALs and PALs include output logic, e.g., tri-state controls and/or gates that allow the combination of logic arrays and output logic referred to as a "macrocell" to be implemented. PALs and GALS typically have multiple inputs and outputs which further increases their versatility and usefulness.

Programmable logic devices are programmed under software control and typically require that the target device's functionality be provided in the form of state equations, truth tables, Boolean expressions, etc. The programming software can then use these descriptions to produce an industry standard binary file known as a JEDEC file[4] which is subsequently loaded into a hardware programmer capable of erasing, copying, verifying and/or programming PLDs.

5.2 Boundary Scanning

While it is sometimes highly desirable for a designer to be able to incorporate custom devices into a design or application to facilitate its optimization, it is imperative that the designer be assured that in doing so a new level of complexity has not been introduced . An important aspect of incorporating programmable devices particularly into sophisticated designs is the ability to confirm that each such device is in and of itself meeting the relevant specifications and expectations of the designer. Complex systems are generally challenging enough without introducing additional challenges in the form of anomalous or unintended behavior/consequences of a programmable device that is a subcomponent of the system.

Boundary scanning[5] is a technique that allows programmable devices to be tested externally, i.e., without access to the internal logic. Internal registers are provided by the device's manufacturer that allow testing of the internal logic and interconnections. However, the device is not aware that such scanning is taking place and therefore the tests can be carried out while the device under test (DUT) is operating in an unperturbed state, or states. PSoC3/5 include, within their respective architectures, a test controller that can be used to access the device's I/O pins for boundary testing by employing an internal serial shift register routed across all of their pins and hence the name "boundary scan".

The circuitry at each PSoC3/5 pin is supplemented with a multipurpose element called a boundary scan cell and most GPIO and SIO port pins have a boundary scan cell associated with them. The interface used to control the values in the boundary scan cells is called the Test Access Port (TAP) and is commonly known as the JTAG interface. It consists of three signals: (1) Test Data In (TDI), (2) Test Data Out (TDO), and (3) Test Mode Select (TMS). Also included is a clock signal (TCK) that clocks the other signals. TDI, TMS, and TCK are all inputs to the device, and TDO is output from the device as shown in Figure 5.2. This interface enables testing multiple ICs on a circuit board, in a daisy-chain fashion.

The TMS signal controls a state machine in the TAP. The state machine controls which register (including the boundary scan path) is in the TDI-to-TDO shift path, as shown in Figure 5.3 for which:

- *ir* refers to the instruction register,
- *dr* refers to one of the other registers (including the boundary scan path), as determined by the contents of the instruction register,

[4]The Joint Electron Devices Engineering Council (JEDEC) has defined standard object file transfer formats for file transport to PLD programmers, e.g., JESD3-C.

[5]PSoC3/5 Support boundary scanning in accordance with the JTAG IEEE Standard 1149.1-2001 Test Access Port and Boundary-Scan Architecture

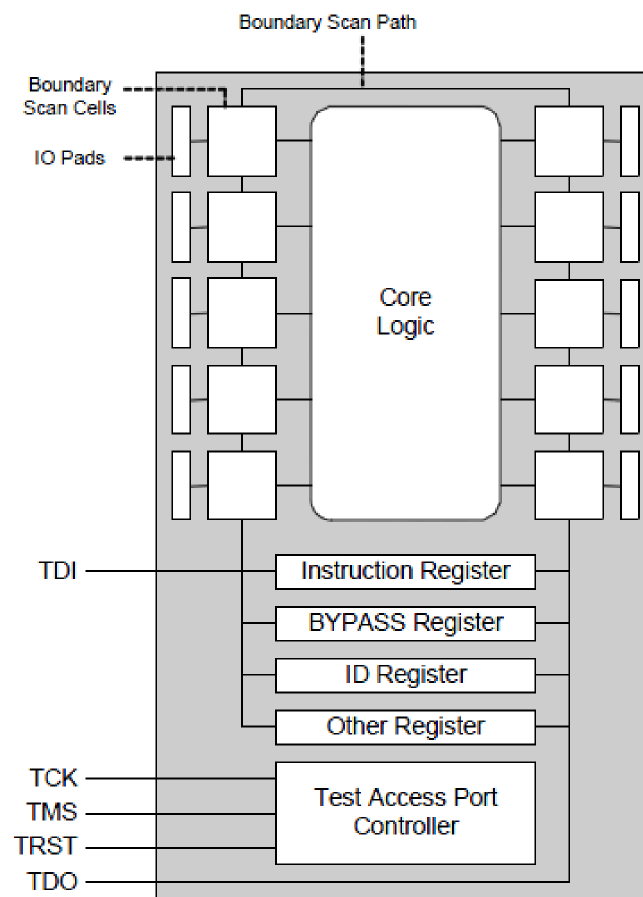


Figure 5.2: PSoC3/5 JTAG interface architecture.

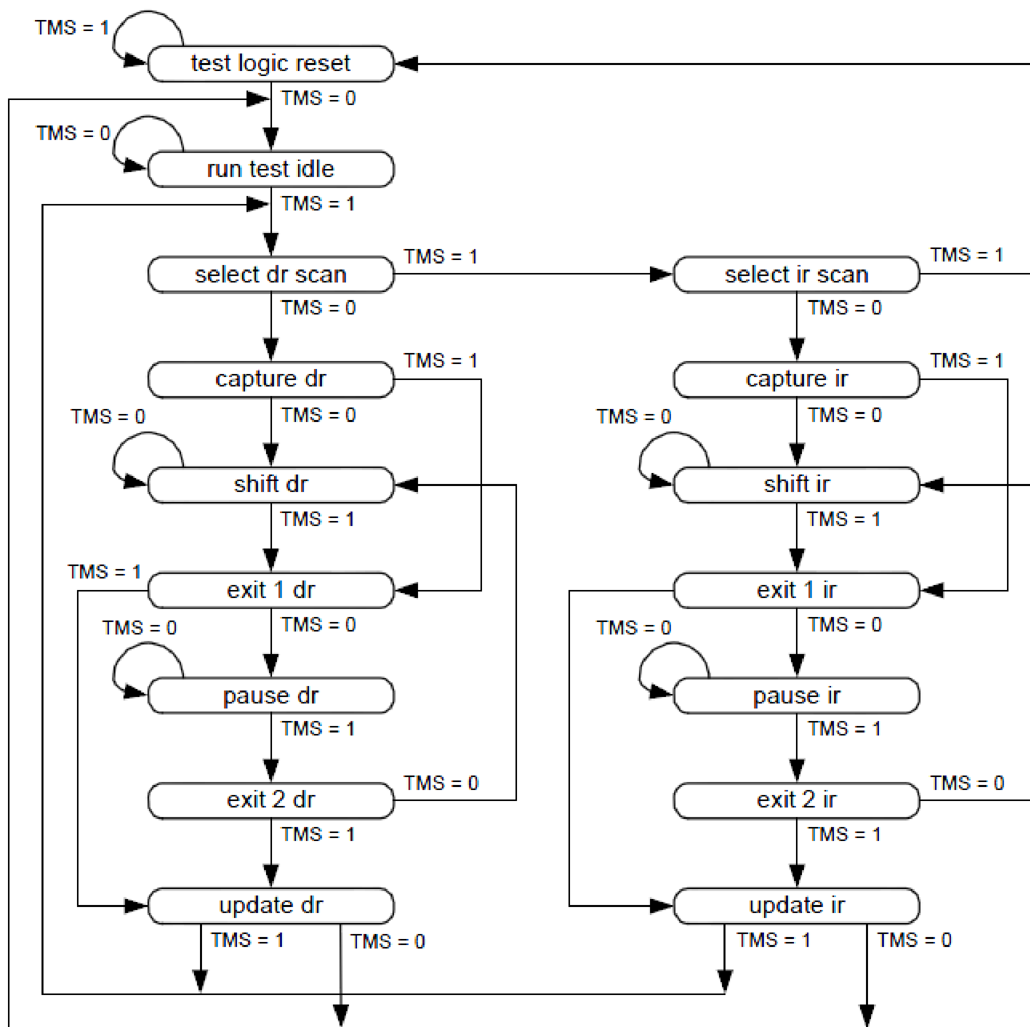


Figure 5.3: Tap state machine.

- *capture* refers to the transfer of the contents of a *dr* to a shift register, to be shifted out on TDO (read the dr)

and,

- *update* refers to the transfer the contents of a shift register, shifted in from TDI, to a dr (write the dr)

The registers in the TAP are:

- *Instruction* - Typically 2 to 4 bits wide and holds the current instruction that defines which data register is placed in the TDI-to-TDO shift path.
- *Bypass* - 1 bit wide, directly connects TDI with TDO, causing the device to be bypassed for JTAG purposes.
- *ID* - 32 bits wide and used to read the JTAG manufacturer/part number ID of the device.
- *Boundary Scan Path (BSR)* - Its width equals the number of I/O pins that have boundary scan cells, used to set or read the states of those I/O pins.

Other registers may be included in accordance with device manufacturer specifications. The standard set of instructions (values that can be shifted into the instruction register), as specified in IEEE 1149, are:

- *EXTEST* - Causes TDI and TDO to be connected to the *boundary scan path* (BSR). The device is changed from its normal operating mode to a test mode. Then, the device's pin states can be sampled using the capture *dr* JTAG state, and new values can be applied to the pins of the device using the update dr state.
- *SAMPLE* - Causes TDI and TDO to be connected to the BSR, but the device is left in its normal operating mode. During this instruction, the BSR can be read by the capture *dr* JTAG state, to take a sample of the functional data entering and leaving the device.
- *PRELOAD* - Causes TDI and TDO to be connected to the BSR, but the device is left in its normal operating mode. The instruction is used to preload test data into the BSR prior to loading an EXTEST instruction. Optional, but commonly available, instructions are:
- *IDCODE* - Causes TDI and TDO to be connected to an IDCODE register.
- *INTEST* - Causes TDI and TDO to be connected to the BSR. While the EXTEST instruction allows access to the device pins, INTEST enables similar access to the core logic signals of a device.

5.3 Macrocells, Logic Arrays and UDBs

Combining gate arrays and macrocells[6] provides a significantly higher level of functionality, particularly when the macrocells include registers, ALUs, flip-flops, etc., than that obtainable through the use of gates alone. One of the simplest configurations of such a combination consists of a *sum-of-products* (SoP) combinatorial logic function and a flip-flop. An example of a device employing multiple such macrocells, as first defined, cells, Logic arrays is shown in Figure 5.4. and serve as fundamental building blocks of PLDs. Combinations of macrocells and gates can be configured in arbitrarily large arrays to provide very complex

[6]In the present discussion, the term macrocell refers solely to a combination of flip-flops and I/O devices exclusive of logic arrays and OR-gates. However, some definitions of the term macrocell represent a broader definition and include all of the logic required to provide the Boolean functionality, flip flops and I/O other than that provided by the logic array. The latter definition is intended to completely encapsulate all of a particular functionality and is often referred to as a "block'. In the case of PSoC Creator, a comprehensive set of such "building blocks" are provided and they are referred to as UDBs, or universal digital blocks.

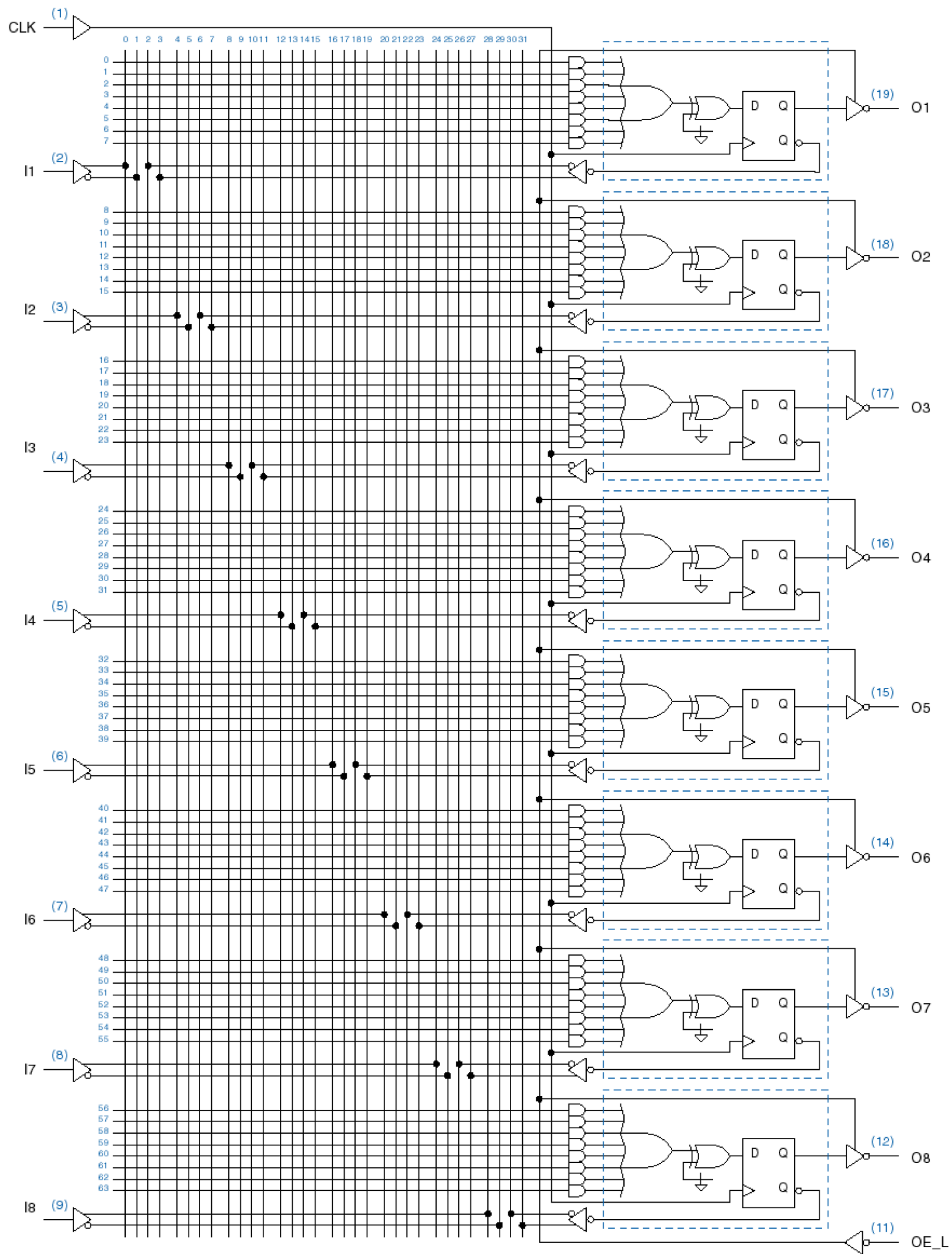


Figure 5.4: A simple device consisting of gate arrays and macrocells.

sequential and combinatorial logic. Some configurations are driven by lookup tables and programmable memory. The information stored in the LUTs can be loaded into memory as required to provide the required logic functions. While in principle the number of inputs to a macrocell are unlimited and therefore should make it possible to have arbitrarily complex functions, a linear increase in fan-in[7] results in a geometric increase in the number of bits to be stored in the LUT.

PSoC Creator allows the designer to employ universal digital blocks based on macrocell-gate array combinations that are that are not only configurable but are specifically designed to serve as customizable blocks within PSoC3/5 for a broad range of embedded system applications that incorporate a microcontroller and associated peripherals. These blocks, referred as UDBs consist of a combination of uncommitted logic similar to programmable logic devices, structured logic (datapaths) and a flexible routing scheme. The UDBs can be further enhanced and supplemented by Boolean elements constructed from basic logic functions supported by PSoC Creator such as AND, NAND, OR, NOR, NOT, XOR, XNOR, D flip-flops, etc.[8] Boolean functions can be created using these basic logic functions to provide the additional functionality that is required for specific applications. Thus designers can create sophisticated systems using the standard set of PSoC3/5 blocks or create combinations of Boolean elements and UDBs to provide the required functionality by employing Verilog/Warp[9].

PSoC3 has 24 UDBs and in the case of pulse width modulators (PWMs), PSoC Creator will allow the creation of as many as 24 PWMs each of which has two independent outputs. Thus it is possible to have 48 PWM outputs.[10] It is also possible to use the 24 UDBs to configure 12 UARTs in a single PSoC3/5 device.

In addition to UDBs, which can be used to provide programmable peripheral functions, PSoC3/5 also includes a suite of user-configurable blocks that provide a wide range of additional capability, e.g., analog, CapSense, communications, digital logic, displays, filters, ports/pins and system blocks as shown in Figure 5.5. All of these blocks, i.e., digital, analog and UDBs, are interoperable and in addition external components such as resistors and capacitors can be used to further extend PSoC3/5's capability as is shown in Chapter 9.

UDB blocks support the following:

- Universal digital block arrays as large as 64 UDBs.
- Portions of UDBs can be either chained, or shared, to enable larger functions.
- Multiple digital functions supported by the UDBs include timers, counters, PWMs (with dead band generator), UART, SPI, and CRC generation/checking.
- Each UDB includes:
  * an ALU-based, 8-bit datapath
  * Two fine-grained PLDs[11]
  * A control and status module
  * A clock and reset module

[7]Fan-in is defined as the number of inputs to a gate, or other device.

[8]Throughout this textbook both upper and lower case will be used when referring to logical operators primarily as a notational convenience.

[9]The discussion of Verilog/Warp begins in section 5.13.

[10]An additional four, single-output PWMs are also available by using PSoC3's fixed function counter/timer/PWMs.

[11]Fine-grained in the present context refers to the implementation of relatively large numbers of simple logic modules as opposed to coarse-grained which implies relatively fewer but larger logic modules often each with two or more sequential logic elements.

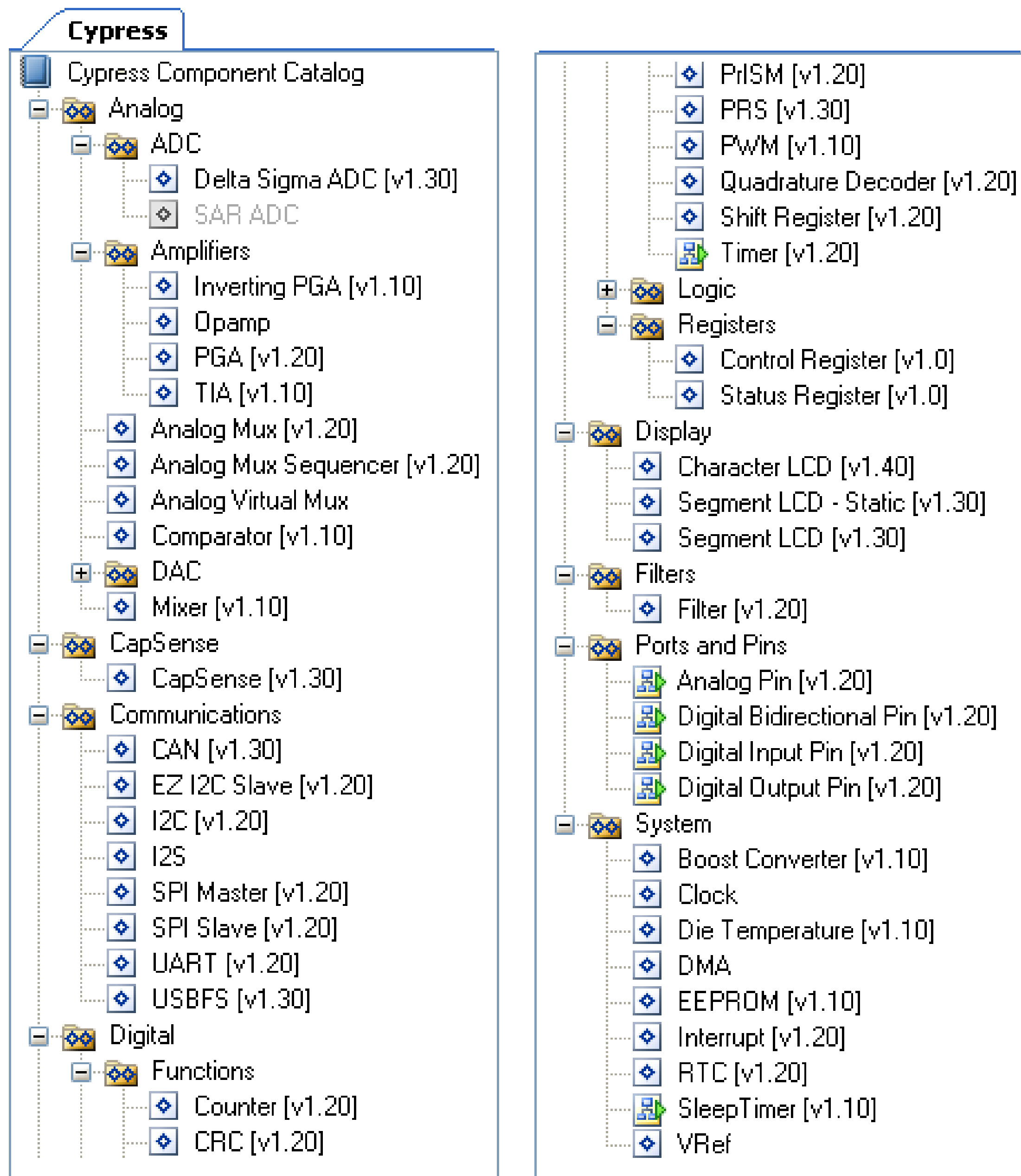


Figure 5.5: Basic building blocks supported by PSoC Creator.

As shown in Figure 5.6, the UDB consists of a pair of PLDs, a datapath and control, status

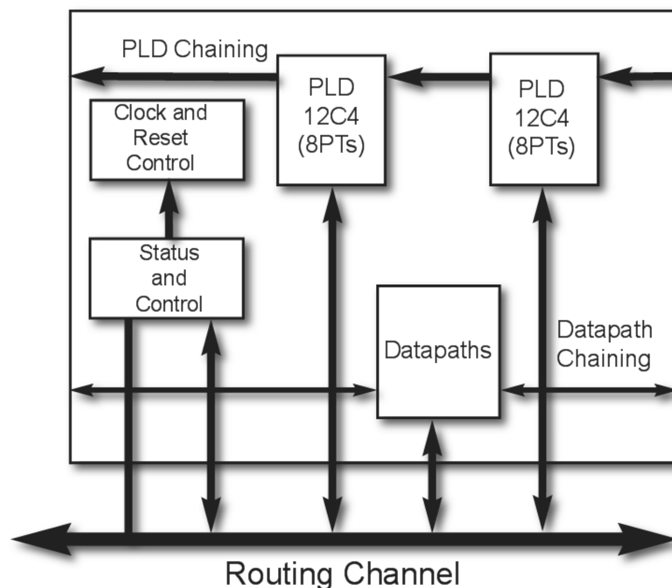


Figure 5.6: UDB Block Diagram.

clock and reset functions. The PLDs accept input from the routing and form registered, or combinational sum-of-products logic, to implement state machines[12], control datapath operations, condition inputs, and drive outputs. The datapath block contains a dynamically programmable ALU, two FIFOs, comparators, and condition generation. The control and status registers provide a way for the CPU firmware to interact, and synchronize, with UDB operations. Control registers drive internal routing, and status registers read internal routing. The reset and clock control block provides clock selection/enabling, and reset selection, for the individual blocks in the UDB. The PLDs and datapath have chaining signals that enable neighboring blocks to be linked to create higher precision functions. UDB I/Os are connected to the routing channel through a programmable switch matrix for connections between blocks in one UDB, and to all other UDBs in the array. All registers and RAM in each UDB are mapped into the system address space and are accessible as both 8- and 16-bit data.

In addition to a UDB's datapath, status register, control register and 2 PLDs, there is also a *count7* down counter available that uses certain resources in the UDB, i.e., the control register, the status register's mask register and if a routed load or enable is used, the status register's inputs. In the latter case if the inputs are not used by the count7, the status register remains available for use.[13] In PSoC Creator *count7* can be implemented as shown in Figure 5.7.

Figure 5.8 shows the internal structure of the PLDs. They can be used to implement state machines, perform input or output data conditioning, and to create lookup tables (LUTs). The PLDs may also be configured to perform arithmetic functions, sequence the datapath, and generate status. General purpose RTL[14] can be synthesized and mapped to the PLD

[12]State machines are discussed in section 5.12
[13]However, the status register's interrupt capability (statusi) not available for use under these circumstances.
[14]RTL refers to register-level-transfer with respect to Verilog code that describes the transformation of data as

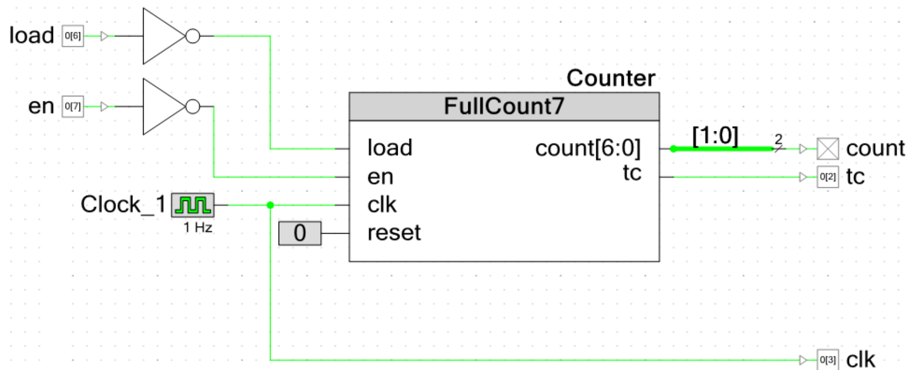


Figure 5.7: Implementation of a *count7* down counter in PSoC Creator.

blocks. Each has 12 inputs which feed across eight product terms (PT) in the AND array. In a given product term, the true (T) or complement (C) of the input can be selected. The output of the PTs are inputs into the OR array. The letter C in 12C4 indicates that the OR terms are constant across all inputs, and each OR input can programmatically access any, or all, of the PTs. This structure gives maximum flexibility and ensures that all inputs and outputs are permutable. Note that there are four outputs OUT0,OUT1, OUT2 and OUT3.

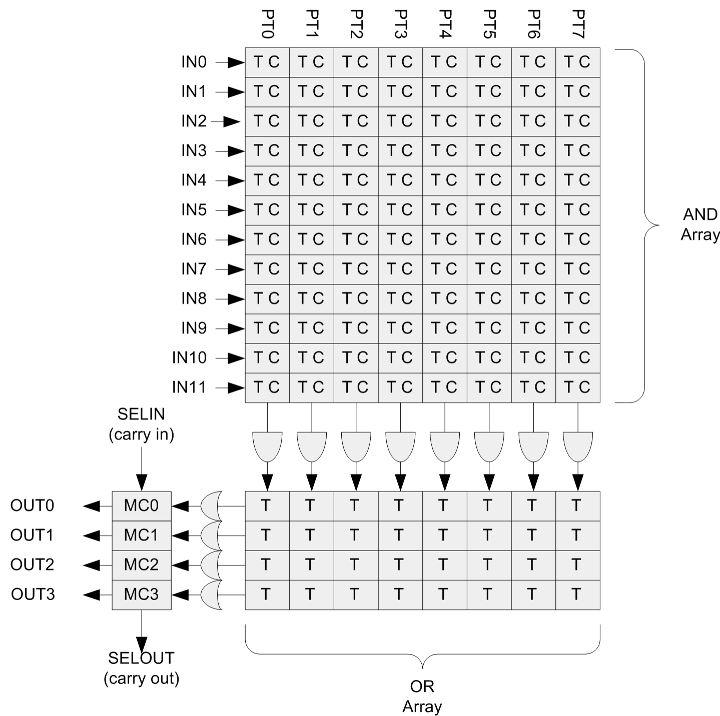


Figure 5.8: PLD 12C4 Structure

PSoC3/5's macrocell architecture is shown in Figure 5.9. The output drives the routing array, and can be registered or combinational. The registered modes are D Flip-Flop with

it is passed from register-to-register.

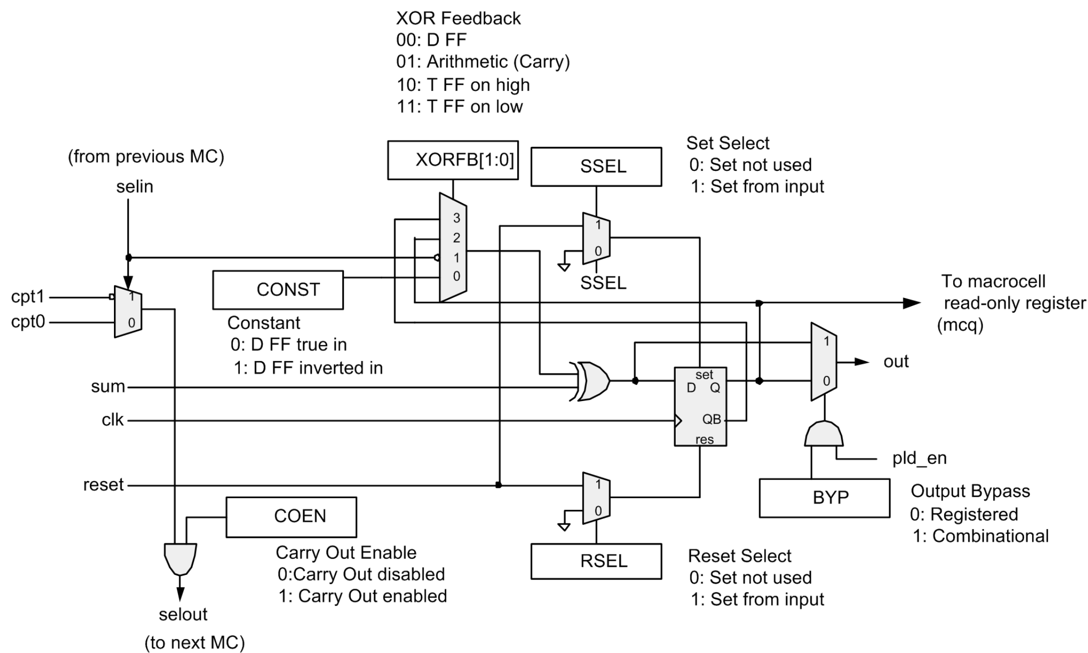


Figure 5.9: PSoC3/5 macrocell architecture.

true or inverted input, and Toggle Flip-Flop on input high or low. The output register can either be set or reset for purposes of initialization, or asynchronously during operation under control of a routed signal. The outputs of the two PLDs are mapped into the address space as an 8-bit, read-only, UDB working register, that is directly addressable by the CPU's firmware, as shown in Figure 5.10.

The PLDs are chained together (the PLD carry chain) in UDB address order. The carry chain input is routed from the previous UDB in the chain, through each macrocell in both of the PLDs, and then to the next UDB as the carry chain out. To support the efficient mapping of arithmetic functions, special product terms are generated and used in the macrocell in conjunction with the carry chain.

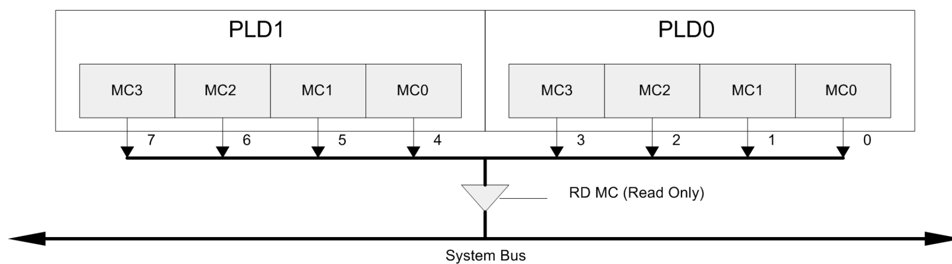


Figure 5.10: Macrocell architecture read-only register.

5.4 The Datapath

The datapath, shown in Figure 5.11, contains an 8-bit single-cycle ALU[15], with associated compare and condition generation circuits. A datapath may be chained with datapaths in neighboring UDBs to achieve higher precision functions. The datapath includes a small, RAM-based control store[16], which can dynamically select the operation and configuration to perform in a given cycle. The datapath is optimized to implement typical embedded functions, such as timers, counters, PWMs, PRS, CRC, shifters and dead band generators. The addition of add and subtract functions allows support for digital Delta Sigma operations.

Dynamic configuration, or perhaps more appropriately "dynamic re-configuration", refers to the ability to change the datapath functions and interconnections, on a cycle-by-cycle basis under sequencer control. This is implemented using the configuration RAM, which stores eight 16-bit wide configurations. The address input to this RAM can be routed from any block connected to the digital peripheral fabric, most typically PLD logic, I/O pins, or other datapaths.

The ALU can perform eight general-purpose functions: increment, decrement, add, subtract, AND, OR, XOR, and PASS. Function selection is controlled by the configuration RAM on a cycle-by-cycle basis. Independent shift (left, right, nibble swap) and masking operations are available at the output of the ALU.

Each datapath has two comparators, with bit-masking options, which can be configured to select a variety of datapath register inputs for comparison. Other detectable conditions include all zeros, all ones, and overflow. These conditions form the primary datapath output selects to be routed to the digital peripheral fabric as outputs, or inputs, to other functions.

The datapath has built-in support for single-cycle Cyclic Redundancy Check (CRC) computation and Pseudo Random Sequence (PRS)[17] generation of arbitrary width and arbitrary polynomial specification. To achieve CRC/PRS widths greater than 8 bits, signals may be chained between datapaths. This feature is controlled dynamically, and therefore can be interleaved with other functions. The most significant bit of an arithmetic and shift function can be programmatically specified (variable MSB). This supports variable width CRC/PRS functions and, in conjunction with ALU output masking, can implement arbitrary width timers, counters, and shift blocks.

5.4.1 Input/Output FIFOs

Each datapath contains two 4-byte FIFOs, that can be individually configured for direction as an input buffer (system bus writes to the FIFO, datapath internals read the FIFO), or an output buffer (datapath internals write to the FIFO, the system bus reads from the FIFO). These FIFOs generate status that can be routed to interact with sequencers, interrupt, or DMA requests.

5.4.2 Chaining

The datapath can be configured to chain conditions and signals with neighboring datapaths. Shift, carry, capture, and other conditional signals can be chained to form higher precision arithmetic, shift, and CRC/PRS functions.

[15]The single-cycle, arithmetic logic units (ALUs) fetch, execute and store results in a single clock cycle.

[16]The control store holds the microinstructions that are used to implement the ALU's instruction set. Some ALUs have been implemented with writable control stores that allow the instruction set to be altered in real time.

[17]Pseudo random refers to the fact that sequence is deterministic and at some point repeat itself. Section 9.15.14 presents a discussion of PRS generation techniques.

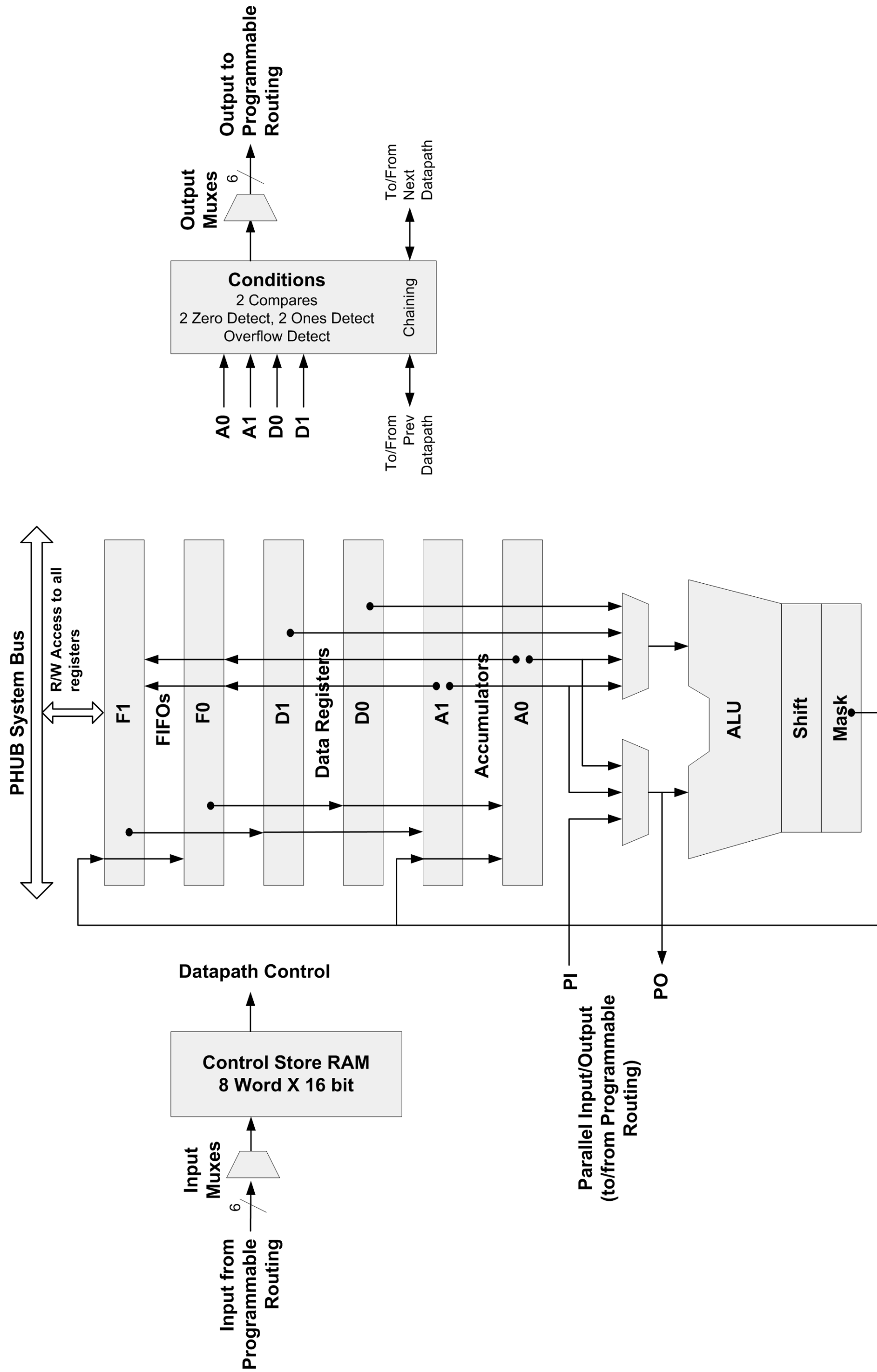


Figure 5.11: Datapath (Top Level).

In applications that are oversampled, or do not need the highest clock rates, the single ALU block in the datapath can be efficiently shared with two sets of registers and condition generators. ALU and shift outputs are registered and can be used as inputs in subsequent cycles. Usage examples include support for 16-bit functions in one 8-bit datapath, or interleaving a CRC generation operation with a data shift operation.

5.4.3 Datapath Inputs and Outputs

The datapath has three types of inputs: configuration, control, and serial/parallel data. The configuration inputs select the control store RAM address. The control inputs load the data registers from the FIFOs and capture accumulator outputs into the FIFOs. Serial data inputs include *shift in* and *carry in*. A parallel data input port allows up to eight bits of data to be brought in from routing.

There are a total of 16 signals generated in the datapath. Some of these signals are conditional signals, e.g., compares, some are status signals, e.g., FIFO status, and the rest are data signals, e.g., shift out. These 16 signals are multiplexed into the six datapath outputs and then driven to the routing matrix. By default the outputs are single-synchronized (pipelined). A combinational output option is also available for these outputs.

5.4.4 Datapath Working Registers

Each datapath module has six, 8-bit working registers all of which are readable and writable by the CPU and DMA:

- **Accumulator (A0,A1)** - The accumulators may be both a source and a destination for the ALU. They may also be loaded from a Data register, or a FIFO. The accumulators typically contain the current value of a function, such as a count, CRC, or shift.
- **Data (D0,D1)** - The Data registers typically contain constant data for a function, such as a PWM compare value, timer period, or CRC polynomial.
- **FIFOs (F0,F1)** - The two 4-byte FIFOs provide both a source and a destination for buffered data. The FIFOs can be configured as one input buffer and one output buffer, two input buffers or two output buffers. Status signals indicate the read and write status of these registers. The FIFOs can be used to buffer TX and RX data in the SPI or UART and PWM compare and timer period data.

Each FIFO has a variety of possible operational modes and configurations:

- **Input/Output** - In input mode, the system bus writes to the FIFO and the data is read and consumed by the datapath internals. In output mode, the FIFO is written to by the datapath internals and is read, and consumed, by the system bus.
- **Single Buffer** - The FIFO operates as a single buffer with no status. Data written to the FIFO is immediately available for reading, and can be overwritten at anytime.
- **Level/Edge** - The control to load the FIFO from the datapath internals can be either level or edge triggered.
- **Normal/Fast** - The control to load the datapath is sampled on the currently selected datapath clock (normal), or the bus clock (fast). This allows captures to occur at the highest rate in the system (bus clock), independent of the datapath clock.
- **Software Capture** - When this mode is enabled, and the FIFO is in output mode, a read by the CPU/DMA of the associated accumulator (A0 for F0, A1 for F1) initiates a synchronous transfer of the accumulator value into the FIFO. The captured value may then

be immediately read from the FIFO by the datapath internals. If chaining is enabled, the operation follows the chain to the MS block for atomic reads by datapaths of multi-byte values.

- **Asynch** - When the datapath is being clocked asynchronously to the system clocks, the FIFO status for use by the datapath state machine (*blk_stat*) is resynchronized to the current DP clock.
- **Independent Clock Polarity** - Each FIFO has a control bit to invert polarity of the FIFO clock with respect to the datapath clock.

The configurations controlled by the FIFO direction bit are shown in Figure 5.12. The TX/RX

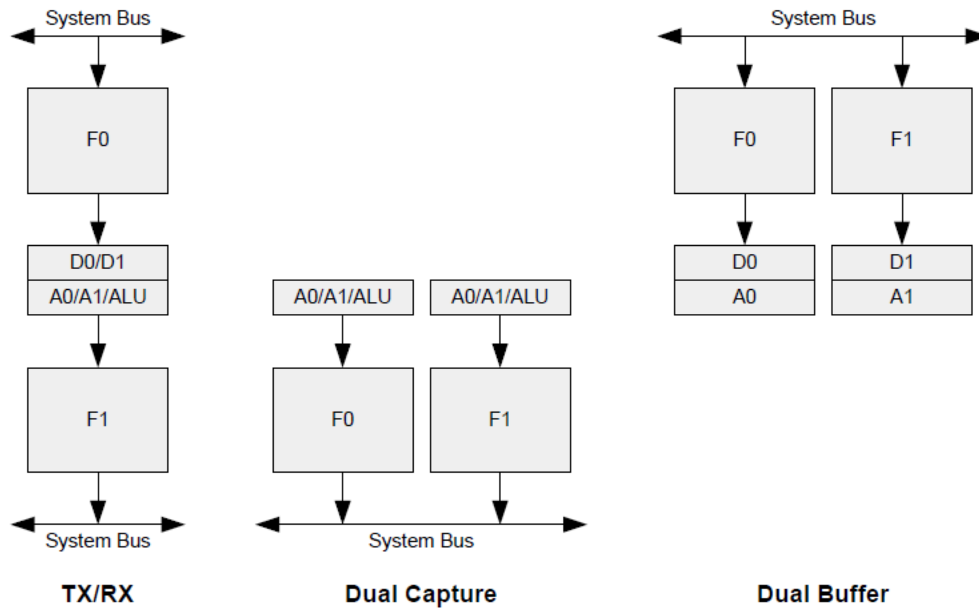


Figure 5.12: FIFO configurations.

mode has one FIFO in input mode and the other in output mode. The primary use for this configuration is serial peripheral interface (SPI) bus communications. The dual capture configuration provides independent capture of A0 and A1, or two separately controlled captures of either A0 or A1. The dual buffer mode provides buffered periods and compares, or two independent periods/compares.

5.5 Datapath ALU

The Datapath block's ALU consists of three, independent, 8-bit, programmable functions that employ an arithmetic/logic, a shifter unit and a mask unit. The ALU functions shown in Table 5.1 are configured dynamically by the RAM control store[18].

5.5.1 Carry Functions

The *carry in* option is used in arithmetic operations. There is a default *carry in* value for each function as shown in Table 5.2. In addition to the default arithmetic mode for carry operation,

[18] "srca" and "srcb" refer to the ALU a and b inputs, respectively.

Table 5.1: ALU Functions

Func[2:0]	Function	Operation
000	PASS	srca
001	INC	++srca
010	DEC	--srca
011	ADD	scra + srcb
100	SUB	srca - srcb
101	XOR	srca ^ srcb
110	AND	srca & srcb
111	OR	srca | srcb

Table 5.2: Carry In Functions

Function	Operation	Default Carry In Implementation
INC	++srca	srca + ooh + ci (ci = 1)
DEC	--srca	srca + ffh + ci (ci=0)
ADD	srca + srcb	srca + srcb + ci (ci=0)
SUB	srca -srcb	srca + -srcb + ci (ci=1)

there are three additional carry options, as shown in Table 5.3. The CI SELA and CI SELB configuration bits determine the *carry in* for a given cycle. Dynamic configuration RAM selects either the A or B configuration on a cycle-by-cycle basis. When a routed carry is used, the

Table 5.3: Carry In Functions

Function	Carry Out Polarity	Carry Out Active	Carry Out Inactive
Inc	True	++srca == 0	srca
Dec	Inverted	--srca == -1	srca
ADD	True	(srca + srcb) > 255	srca+srcb
Sub	Inverted	(srca + srcb) < 0	scra-scrb

meaning with respect to each arithmetic function is shown in Table 5.4. Note that in the case of the decrement and subtract functions, the carry is active low (inverted).

The *carry out* option is a selectable datapath output and is derived from the currently defined MSB position, which is statically programmable. This value is also chained to the next most significant block as an optional *carry in*. Note that in the case of decrement and subtract functions, the carry out is inverted. Options for *carry in*, and for MSB selection for carry out generation, are shown in Figure 5.13. The registered carry out value may be selected as the *carry in* for a subsequent arithmetic operation. This feature can be used to implement higher precision functions in multiple cycles.

Table 5.4: Routed Carry In Functions

Function	Carry In Polarity	Carry In Active	Carry In Inactive
Inc	True	++srca	srca
Dec	Inverted	--srca	srca
ADD	True	(srca + srcb) + 1	srca+srcb
Sub	Inverted	(srca + srcb) - 1	scra-scrb

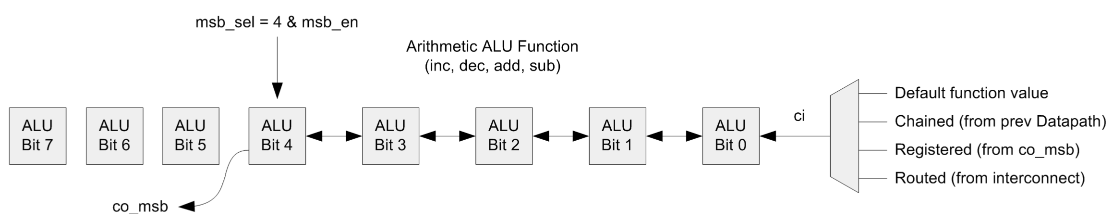


Figure 5.13: Carry operation block diagram.

Additional *carry in* functions are provided by CI SEL A and CI SEL B. A value of 00 imposes the default carry mode. A value of '01' sets the carry mode as *registered* so that add with carry and subtract with borrow operations can be implemented. In this mode, the carry flag represents the result of the previous cycle. A value of '10' sets the *routed carry* mode for cases in which the carry is generated somewhere else and routed to the input allowing controllable counters to be implemented. Finally, the value '11' sets the *chainable* carry mode allowing the carry to be *chained* from the previous datapath and used to implement single-cycle operations of higher precision involving two or more datapaths.

5.5.2 ALU Masking Operations

An 8-bit mask register in the UDB static configuration register space defines the masking operation. In this operation, the output of the ALU is masked (ANDed) with the value in the mask register. A typical use for the ALU mask function is to implement free-running timers and counters in "powers-of-two" resolutions.

5.5.3 All Zeros and Ones Detection

Each accumulator has dedicated *all zeros* and *all ones* detect capability. These conditions are statically chainable as specified in UDB configuration registers. In addition, the requirement to chain, or not chain, these conditions is statically specified in UDB configuration registers. Chaining of zero detect is the same concept as the compare equal. Successive chained data is ANDed, if the chaining is enabled.

5.5.4 Overflow

Overflow is defined as the XOR of the carry into the MSB and the *carry out* of the MSB. The computation is done on the currently defined MSB as specified by the MSB_SEL bits. Although this condition is not chainable, the computation is valid when done in the most significant datapath of a multi-precision function, as long as the carry is chained between blocks.

5.5.5 Shift Operations

Shift operations, shown in Table 5.5, can occur independently from those of the ALU. A *shift out*

Table 5.5: Shift functions.

Shift[1:0]	Function
00	Pass
01	Shift Left
10	Shift Right
11	Nibble Swap

value is available as a datapath output. Both *shift out right* (sor) and *shift out left* (sol_msb) share that output selection. A static configuration bit (SHIFT_OUT in register CFG15) determines which shift output is used as a datapath output. In the absence of a shift, the sor and sol_msb signal is defined as the LSB[19] or MSB of the ALU function, respectively.

The SI SELA and SI SELB configuration bits determine the shift in data for a given operation. Dynamic configuration RAM selects the A or B configuration on a cycle-by-cycle basis. Shift in data is only valid for left and right shift; it is not used for pass and nibble swap. The selections and usage apply to both left and right shift directions, and if for either SI SEL A or SI SEL B the bit values are 00, the shift in source is default/arithmetic, i.e., the default input is the value of the DEF SI configuration bit(fixed 0 or 1). However, if the MSB SI bit is set, then the default input is the currently defined MSB, but for right shift only.

If the bit values are 01, then the shift in source is registered and the shift value is driven by the current registered shift-out value from the previous cycle. The shift-left operation uses the last shift-out left value.The right-shift operation uses the last shift-out right value. If the bit values are 10, then the shift-in source is “routed”. Shift is selected from the routing channel, i.e., the SI input. Finally, if the bit values are 11, the shift-in source is chained and shift-in left is routed from the right datapath neighbor.

The shift-out data comes from the currently defined MSB position and the data that is shifted in from the left (*shift-in right*) goes into the currently defined MSB position. Both shift-out data (left or right) are registered and can be used in a subsequent cycle. This feature can be used to implement a higher precision shift in multiple cycles. The bits that are isolated by the MSB selection are still shifted.

In the example shown in Figure 5.14, bit 7 still shifts in the *sil* value on a right shift and bit 5 shifts in bit 4, on a left shift. The *shift out,* either right or left, from the isolated bits is lost.

5.5.6 Datapath Chaining

As discussed previously, each datapath block contains an 8-bit ALU, which is designed to chain carries, shifted data, capture triggers, and conditional signals to the nearest neighbor datapaths to create higher precision arithmetic functions and shifters. These chaining signals, which are dedicated signals, allow single-cycle 16-, 24- and 32-bit functions to be efficiently implemented without the timing uncertainty of channel routing resources. In addition, the capture chaining makes possible an atomic read of the accumulators in chained blocks. As shown in Figure 5.15, all generated conditional and capture signals chain in the direction of least significant to most

[19]The acronym for least significant byte is LSB.

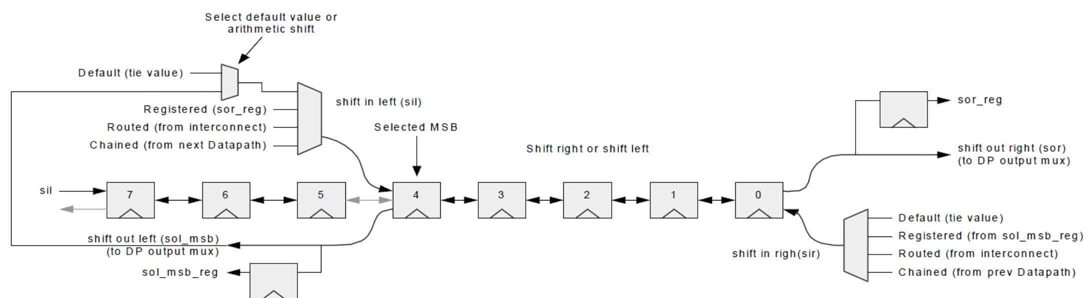


Figure 5.14: Shift Operation.

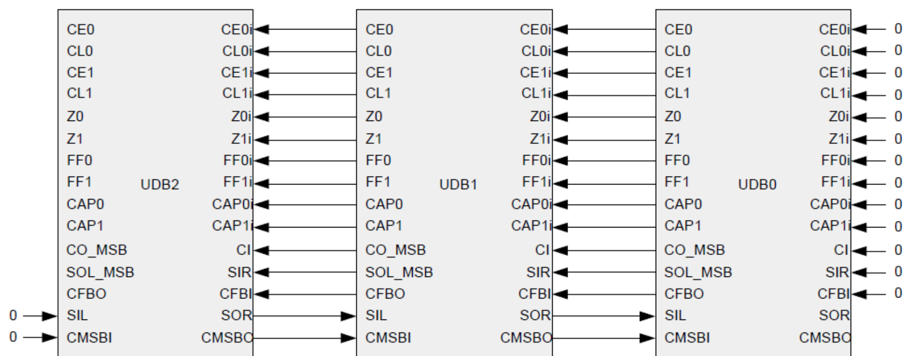


Figure 5.15: Datapath chaining flow.

significant blocks. Shift left also chains from the least-to-most significant block and shift right chains from the most-to-least significant block. The CRC/PRS chaining signal for feedback chains from the least-to-most significant block; the MSB output chains from the most-to-least significant block.

5.5.7 Datapath and CRC/PRS

The datapath has special connectivity to allow cyclic redundancy checking (CRC) and pseudo random sequence (PRS) generation. Chaining signals are routed between datapath blocks to support CRC/PRS bit lengths of more than 8 bits. The most significant bit (MSb) of the most significant block in the CRC/PRS computation is selected and routed, while chained across blocks, to the least significant block. The MSB is then XORed with the data input (SI data) to provide the feedback (FB) signal. The FB signal is then routed and chained across blocks to the most significant block. This feedback value is used in all blocks to gate the XOR of the polynomial from the Data0 or Data1 register with the current accumulator value.

Figure 5.16 shows the structural configuration for the CRC operation. The PRS configuration is identical except that the *shift in* (SI) is tied to '0'. In the PRS configuration, D0 or D1 contain the polynomial value, while A0 or A1 contain the initial or seed[20] value and the CRC residual value at the end of the computation. To enable CRC operation, the CFB_EN bit in the dynamic configuration RAM must be set to '1'. This enables the AND of SRCB ALU input with the CRC feedback signal. When set to zero, the feedback signal is driven to '1', which allows for normal

[20]The phrase "seed value" is used throughout this textbook in various contexts and refers to an initial value with which to begin a process.

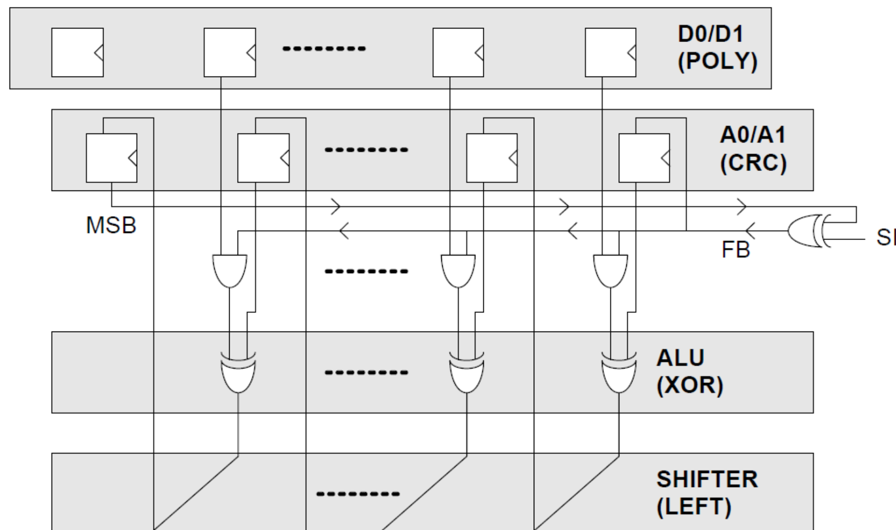


Figure 5.16: CRC Functional Structure.

arithmetic operation. Dynamic control of this bit on a cycle-by-cycle basis gives the capability to interleave a CRC/PRS operation with other arithmetic operations.

5.5.8 CRC/PRS Chaining

Figure 5.17 illustrates an example of CRC/PRS chaining across three UDBs. This arrangement is capable of supporting a 17- to 24-bit operation. The chaining control bits are set according to the position of the datapath in the chain, as shown.

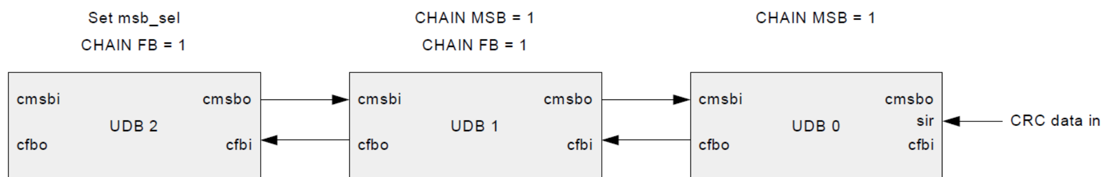


Figure 5.17: CRC/PRS Chaining Configuration

The CRC/PRS MSB signal (cmsbo, cmsbi) is chained based on the following:

- If a given block is the most significant block, the MSb bit (according to the polynomial selected) is configured using the MSB_SEL configuration bits. If a given block is not the most significant block, the CHAIN MSB configuration bit must be set and the MSb signal is chained from the next block in the chain. If a given block is the least significant block, then the feedback signal is generated in that block from the built-in logic that takes the shift in from the right (*sir*) and XORs it with the MSb signal. (For PRS, the *sir* signal is tied to '0'.)
- If a given block is not the least significant block, the CHAIN FB configuration bit must be set and the feedback is chained from the previous block in the chain.

The CRC/PRS MSb signal (*cmsbo*, *cmsbi*) is chained based on the following:

- If a given block is the most significant block, the MSB bit (according to the polynomial selected) is configured using the MSB_SEL configuration bits. If
- If a given block is not the most significant block, the CHAIN MSB configuration bit must be set and the MSB signal is chained from the next block in the chain.

5.5.9 CRC/Polynomial Specification

The following is an illustrative example of how to configure the polynomial for programming into the associated D0/D1 register. Consider the CCITT[21] CRC-16 polynomial, which is defined as x16 + x12 + x5 + 1. The method for deriving the data format from the polynomial is shown in Figure 5.18. The X0 term is inherently always '1' and therefore does not need to be programmed.

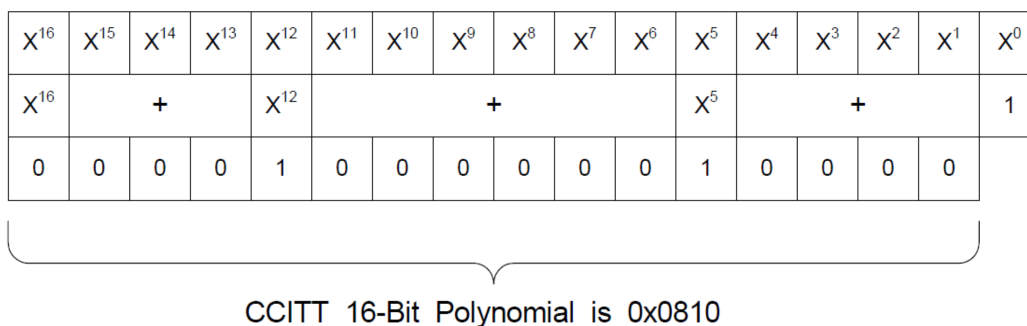


Figure 5.18: CCITT CRC 16 Polynomial

For each of the remaining terms in the polynomial, a '1' is set in the appropriate position in the alignment shown.[22]

Assuming that D0 contains the polynomial and A0 is used to compute CRC/PRS a suitable polynomial has to be selected and written into D0. Next a seed value is selected and written into A0.

5.5.10 External CRC/PRS Mode

A static configuration bit may be set (EXT CRCPRS) to enable support for external computation of a CRC or PRS. As shown in Figure 5.19, computation of the CRC feedback is done in a PLD block. When the bit is set, the CRC feedback signal is driven directly from the CI (*Carry In*) datapath input selection mux, bypassing the internal computation. The figure shows a simple configuration that supports up to an 8-bit CRC or PRS, inclusive. Normally the built-in circuitry is used, but this feature allows more elaborate configurations, such as up to a 16-bit, inclusive, CRC/PRS function in one UDB, using time division multiplexing. In this mode, the dynamic configuration RAM bit CFB_EN still controls whether the CRC feedback signal is ANDed with the SRCB ALU input. Therefore, as with the built-in CRC/PRS operation, the function can be interleaved with other functions, if desired.

[21] CCITT is an abbreviation for Comité Consultatif International Téléphonique et Télégraphhique which is an international standards organization involved in the development of communications standards.

[22] This polynomial format is slightly different from the format normally specified in HEX. For example, the CCITT CRC16 polynomial is typically denoted as 1021H. To convert it to the format required for datapath operation, shift right by one and add a '1' in the MSb location. In this case, the correct polynomial value to load into the D0 or D1 register is 1810H.

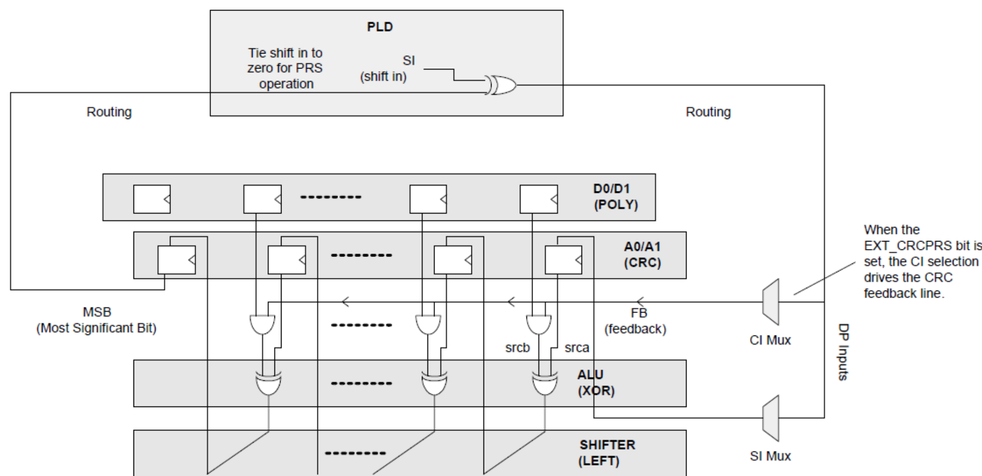


Figure 5.19: External CRC/PRS Mode.

5.5.11 Datapath Outputs and Multiplexing

Datapath outputs and multiplexing conditions are generated from the registered accumulator values, ALU outputs, and FIFO status. These conditions can be driven to the UDB channel routing for use in other UDB blocks as interrupts, DMA requests, or applied to globals and I/O pins. The 16 possible conditions are shown in Table 5.6. Conditions are generated from the registered accumulator values, ALU outputs, and FIFO status. These conditions can be driven to the UDB channel routing for use in other UDB blocks, use as interrupts or DMA requests, or to globals and I/O pins. The 16 possible conditions are shown in the Table 5.6. There are a total

Table 5.6: Datapath Condition Generation

Condition	Chain ?	Description
Compare Equal	Y	A0==D0
Compare Less Than	Y	A0<D0
Zero Detect	Y	A0==OOh
Ones Detect	Y	A0==FFh
Compare Equal	Y	A1 or A0 == D1
Compare Less Than	Y	A1 or A0 < D1
Zero Detect	Y	A1 == 00h
Ones Detect	Y	A1 == FFh
Overflow	N	Carry(msb) ^ Carry(msb-1)
Carry Out	Y	Carry out of MSB defined bit
CRC MSB	Y	MSB of CRC/PRS function
Shift Out	Y	Selection of shift Output
FIF00 Block Status	N	Depends on FIFO Config
FIFO1 Block Status	N	Depends on FIFO Config
FIFO0 Block Status	N	Depends on FIFO Config
FIFO1 Block Status	N	Depends on FIFO Config

of six datapath outputs. Each output has a 16-1 multiplexer that allows any of these 16 signals

to be routed to any of the datapath outputs.

5.5.12 Compares

There are two compares, one of which has fixed sources (Compare 0) and the other has dynamically selectable sources (Compare 1). Each compare has an 8-bit, statically programmed mask register, which enables the compare to occur in a specified bit field. By default, the masking is off (all bits are compared) and must be enabled. Comparator 1 inputs are dynamically configurable. As shown in Table 5.7, there are four options for Comparator 1, which applies to both the"less than" and the"equal" conditions.

The CMP SELA and CMP SELB configuration bits determine the possible compare configurations. A dynamic RAM bit selects one of the A or B configurations on a cycle-by-cycle basis. Compare 0 and Compare 1 are independently chainable to the conditions generated in the previous datapath (in addressing order). Whether to chain compares, or not, is statically specified in the UDB configuration registers. Figure 5.20 illustrates *compare equal* chaining, which is just an ANDing of the *compare equal* in this block with the chained input from the previous block. Figure 5.21 illustrates *compare less than* chaining.

Table 5.7: Compare Configurations

CMP SEL A CMP SEL B	Compare 1 Compare Configuration
00	A1 compare to D1
01	A1 compare to A0
10	A0 compare to D1
11	A0 compare to A0

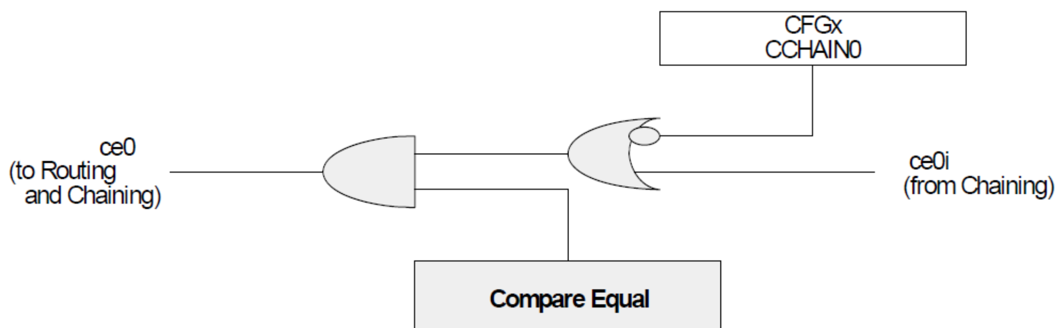


Figure 5.20: Compare *Equal* Chaining.

In this case, the "less than" is formed by the *compare less than* output in this block, which is unconditional. This is ORed with the condition where this block is equal, and the chained input from the previous block is asserted as less than.

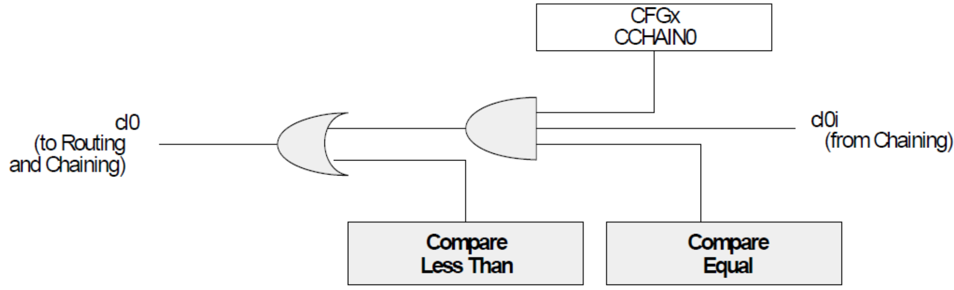


Figure 5.21: Compare *Less Than* chaining.

5.6 Dynamic Configuration RAM (DPARAM)

Each datapath contains a 16 bit-by-8 word dynamic configuration RAM, which is shown in Figure 5.22. The purpose of this RAM is to control the datapath configuration bits on a cycle-by-cycle

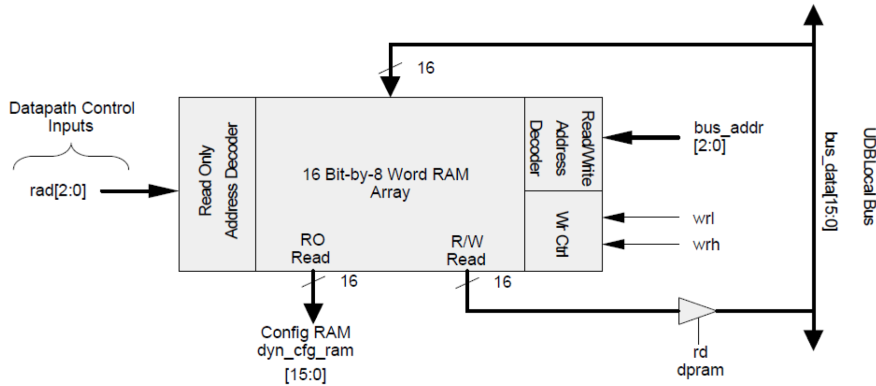


Figure 5.22: Configuration RAM I/O

basis, based on the clock selected for that datapath. This RAM has synchronous read and write ports for purposes of loading the configuration via the system bus. An additional asynchronous read port is provided as a fast path to output these 16-bit words as control bits to the datapath. The asynchronous address inputs are selected from datapath inputs and can be generated from any of the possible signals on the channel routing, including I/O pins, PLD outputs, control block outputs, or other datapath outputs. The primary purpose of the asynchronous read path is to provide a fast single-cycle decode of datapath control bits.

5.7 Status and Control Mode

When operating in status and control mode, this module functions as a status register, interrupt mask register, and control register in the configuration shown in Figure 5.23.

5.7.1 Status Register Operation

One 8-bit, read-only status register is available for each UDB and inputs to this register come from any signal in the digital routing fabric. The status register is non-retentive, i.e., it loses its state during sleep intervals and is reset to 0x00 upon awakening. Each bit can be independently

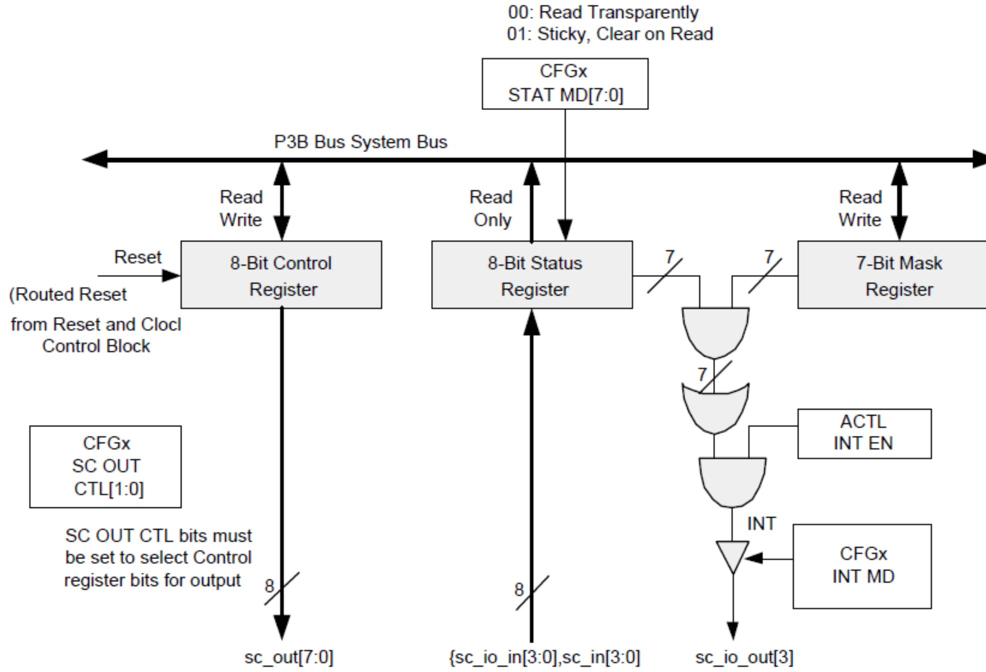


Figure 5.23: Status and Control Operation

programmed to operate in one of two ways, 1) for STAT MD = 0, a read returns the current value of the routed signal (transparent) and 2) for STAT MD = 1, a high on the internal net is sampled and captured (sticky, clear on read).[23]

An important feature of the status register clearing operation is that clearing of status is only applied to the bits that are set. This allows other bits that are not set to continue to capture status, and a coherent view of the process can be maintained.

5.7.2 Status Latch During Read

Figure 5.24 shows the structure of the status read logic. The sticky status register is followed by a latch, which latches the status register data and holds it stable during the duration of the read cycle, regardless of the number of wait states in a given read.

5.7.3 Transparent Status Read

By default, a CPU read of this register transparently reads the state of the associated routing net. This mode can be used for a transient state that is computed and registered internally in the UDB.

5.7.4 Sticky Status, with Clear on Read

In this mode, the associated routing net is sampled on each cycle of the status and control clock. If the signal is high in a given sample, it is captured in the status bit and remains high, regardless of the subsequent state of the associated route. When CPU firmware reads the status register,

[23] It is cleared when the register is read.

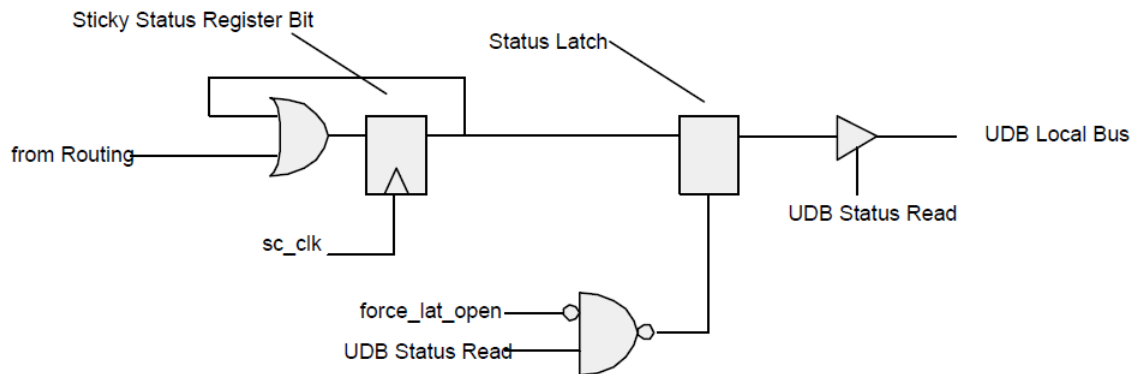


Figure 5.24: Status Read Logic

the bit is cleared. The status register clearing is independent of mode and occurs even if the block clock is disabled; it is based on the bus clock and occurs as part of the read operation.

5.8 Counter Mode

When a UDB is in counter mode, the control register operates as a 7-bit down counter with programmable period and automatic reload that can be used for UDB internal operations or firmware applications. 7-bit down counter. Routing inputs can be configured to control both the enable and reload of the counter. When enabled, control register operation is not available.

The counter has the following features:

- a 7-bit read/write period register, a 7-bit count register that is read/write but can only be accessed when the counter is disabled,
- automatic reload of the period to the count register on terminal count (0),
- a firmware control bit in the Auxiliary Control working register called CNT START, to start and stop the counter,[24]
- selectable bits from the routing for dynamic control of the counter enable and load functions: EN, routed enable to start or stop counting and LD, routed load signal to force the reload of period,[25]
- it is level sensitive and continues to load the period while asserted. The 7-bit count may be driven to the routing fabric as sc_out[6:0],
- the terminal count may be driven to the routing fabric as sc_out[7].

To enable this mode, the SC_OUT_CTl[1:0] bits must be set to counter output. In this mode the normal operation of the control register is not available. The status register can still be used for read operations, but should not be used to generate an interrupt because the mask register is reused as the counter period register. The use of SYNC mode depends on whether or not the dynamic control inputs (LD/EN) are used. If they are not used, SYNC mode is unaffected. If they are used, SYNC mode is unavailable.

5.8.1 Sync Mode

As shown in Figure 5.25, the status register can operate as a 4-bit double synchronizer, clocked

[24]This is an overriding enable and must be set for optional routed enable to be operational.
[25]When this signal is asserted, it overrides a pending terminal count.

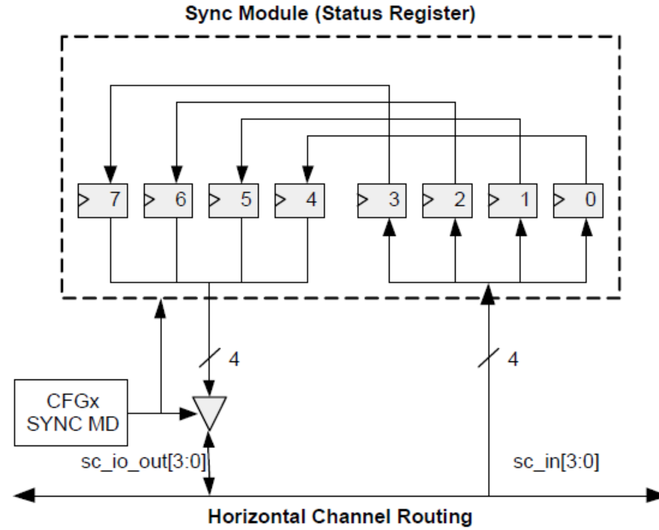


Figure 5.25: Sync Mode

by the current SC_CLK, when the SYNC MD bit is set. This mode may be used to implement local synchronization of asynchronous signals, such as GPIO inputs. When enabled, the signals to be synchronized are selected from UDB pins SC_IN[3:0], the outputs are driven to the SC_IO_OUT[3:0] pins, and SYNC MD automatically puts the SC_IO pins into output mode. When in this mode, the normal operation of the status register is not available, and the status sticky bit mode is forced off, regardless of the control settings for this mode. The control register is not affected by the mode. The counter can still be used with limitations. No dynamic inputs (LD/EN) to the counter can be enabled in this mode.

5.8.2 Status and Control Clocking

The status and control registers require a clock selection for any of the following operating modes:

- Control register in counter mode
- Status register with any bit set to "sticky" [26]
- Sync mode

The clock for this block is allocated in the reset and clock control module.

5.8.3 Auxiliary Control Register

The read-write Auxiliary Control register is a special register that controls fixed function hardware in the UDB. This register allows CPU firmware to dynamically control the built-in interrupt, FIFO, and counter hardware. The register bits and descriptions are shown below:

[26] Sticky bits are defined as bits that retain their current value until they are reset, e.g., by the CPU.

5.9 Boolean Functions

George Boole[27] published a seminal work in 1847, entitled "The Mathematical Analysis of Logic" that was followed by a second equally important work in 1854 entitled "An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities". His approach was to develop a type of linguistic algebra[28] based on the three constructs AND $(A \cdot B)$, OR $(A + B)$ and NOT $(\overline{A})$.[29] He was then able to show that they could be used to carry out basic mathematical functions and comparisons. Thus it became possible to express logical statements in terms of algebraic equations. His work ultimately formed the basis for much of modern computer technology. Claude Elwood Shannon[30] was the first use Boolean algebra in describing digital circuits.

Simply stated, Boolean Algebra allows any computable algorithm, or realizable digital circuit, to be expressed as a system of Boolean equations. AND, OR and NOT can be easily constructed from NAND gates which is equivalent to an AND gate followed by a NOT gate, as shown in Figure 5.26. Having a single type of building block, i.e., NAND gates, as the basis for these

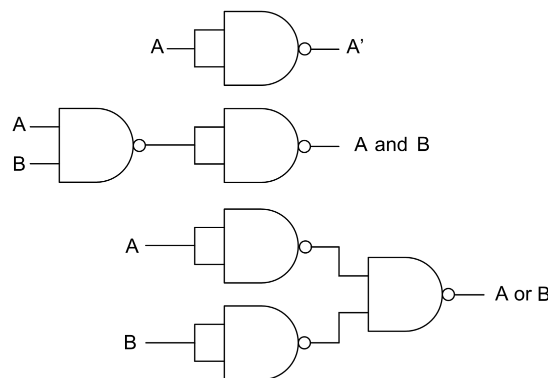


Figure 5.26: The NAND gate as the basic building block for AND, OR and NOT (inverter) gates.

three functions allows very complex circuits to be created from the same building block. Boolean functions operate on Boolean variables and the resulting value of a Boolean function is either one or zero.

The formal definition of a Boolean function is given by:

> A Boolean function is a mapping from the Cartesian product $x^n\{0,1\}$ to $\{0,1\}$, i.e., a function $F : x^n\{0,1\} \Rightarrow setB = \{0,1\}$ where $x^n\{0,1\}$ is the set of all n-tuples $\{x_1, x_2 \cdots .x_n\}$ and the x_n are either one or zero.

[27] George Boole (1815-1864), a mathematician, introduced not only a seminal theory on symbolic logic which was ultimately to be known as Boolean logic, but also two important treatises on differential equations and the calculus of finite differences.

[28] Which ultimately became universally referred to as "Boolean Algebra".

[29] Note that although AND and OR are binary operators, NOT is a unary operator since it operates on only one operand.

[30] Claude E. Shannon is regarded by many as the founding father of the electronic communications age. Both a mathematician and an engineer, he applied Boole's logical algebra to telephone switching circuits and authored a classic paper entitled "A Symbolic Analysis of Relay and Switching Circuits". His work on information theory beginning with his two-part paper entitled "A Mathematical Theory of Communication" continues to be widely studied and has contributed much to the evolution of modern computer technology.

The set B = {0,1} is arguably one of the most used sets in the world. Boolean algebra provides the operations and rules for working with this set and forms the foundation for development and use of digital circuits and for VLSI design. A Boolean algebra consists of a set of operators and a set of axioms. The operators for the Boolean algebra to be discussed here are +, $\cdot$ and $'$ for OR, AND and the complement[31], respectively. The order of precedence for these operators is complement, product and then sum.

The set of postulates include:

- closure,[32]
- the existence of identity elements for AND (1), and, OR (0) but not for NOT, $(A + 0 = A,\ A \cdot 1 = A)$
- associative: A + (B+C) = (A+B) +C,
- commutative: $A + B = B + A$ and $A \cdot B = B \cdot A$,
- distributive: $A + (B \cdot C) = (A + B) \cdot (A + C)$ and $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$,

and,

- inverse: $A + A' = 1$ and $A \cdot A' = 0$.

Boolean expressions are given either in terms of *minterms* or *maxterms* which are defined respectively as the *product* of N literals, each of which occurs only once, and the *sum* of N literals, each of which occurs only once. A *literal* a variable within a term of the expression that may, or may not, be complemented. A Boolean function is a mapping from a domain consisting of n-tuples of zeros and ones to a range consisting of an element of B. Boolean functions can be expressed as a *sum-of-products* (SoP)

$$F = (\overline{A} \cdot B) + (A \cdot \overline{B}) \tag{5.1}$$

or as a *product-of-sums* (POS)

$$F = (A + B) \cdot (\overline{A} + \overline{B} \tag{5.2}$$

It is possible to derive a sum-of-products expression for any digital logic circuit, no matter how complex, provided that a description of it in the form of truth table exists. However, using sum-of-products does not guarantee that the end result will be an optimal design. This is of concern because as a practical matter minimizing the number of gates required can result in very significant reductions in cost, better performance and often increased speed. What may appear to the causal observer, in what follows, as addition and multiplication operations is in fact the operations of OR and AND, respectively. Various notations have been adopted for these operations, e.g.,

$$\begin{aligned} A \cdot B &= AB = A \textbf{ OR } B = A \vee B \quad (5.3) \\ A + B &= A \textbf{ AND } B = A \wedge B \quad (5.4) \end{aligned}$$

The AND and OR operators are associative,

$$\begin{aligned} (A \cdot B) \cdot C &= A \cdot (B \cdot C) \quad (5.5) \\ (A + B) + C &= A + (B + C) \quad (5.6) \end{aligned}$$

[31]The complement of one is zero and the complement of zero is one.

[32]x is a Boolean variable if, and only if (iff), its values are restricted to elements of B under AND, OR and NOT.

commutative,

$$A \cdot B = B \cdot A \tag{5.7}$$
$$A + B = B + A \tag{5.8}$$

and distributive,

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C) \tag{5.9}$$
$$A + (B \cdot C) = (A + B) \cdot (A + C) \tag{5.10}$$

In addition, for any value A there exists and A' such that $A + A' = 1$ and $A \cdot A' = 0$. All of which leads to some very important and useful results, e.g.,

$$\begin{aligned}
A \cdot\ B &= B \cdot A \\
A + B &= B + A \\
A \cdot (B \cdot C) &= (A \cdot B) \cdot C \\
A + (B + C) &= (A + B) + C \\
A \cdot (B + C) &= (A \cdot B) + (A \cdot C) \\
A + (B \cdot C) &= (A + B) \cdot (A + B) \\
A \cdot A &= A \\
A + A &= A \\
A \cdot (A + B) &= A \\
A + (A \cdot B) &= A \\
A \cdot A' &= 0 \\
A + A' &= 1 \\
(A')' &= A \\
(A \cdot B)' &= A' + B' \\
(A + B)' &= A' \cdot B' \\
A + 1 &= 1 \\
A \cdot 1 &= A \\
A \cdot 0 &= 0 \\
A + 0 &= A \\
&\cdots
\end{aligned}$$

It should be noted that for any valid Boolean expression, if the + operators in the expression are replaced by · operators, the · operators by + operators and 0's for 1's and 1's for 0's the result is also a valid Boolean expression, although the values of the two expressions may not be the same. This property is referred to as *duality.*

DeMorgan's Theorem[33] states that the complement of the product of variables is equal to the sum of the complements of the variables and conversely the complement of the sum of variables is equal to the product of the complements of the variables[34], i.e.,

$$\overline{A + B} = \overline{A} \cdot \overline{B} \tag{5.11}$$

[33] Augustus De Morgan (1806-1871). DeMorgan a British mathematician and logician, born in India, who was contemporary of Charles Babbage and William Hamilton He introduced the phrase "mathematical induction" and served as a significant reformer of mathematical logic. He is best remembered for his work on purely symbolic algebras, De Morgan's laws and symbolic logic.

[34] Note that in general $\overline{A \cdot B \cdot C \cdot \cdots} = \overline{A} + \overline{B} + \overline{C} + \cdots$ and $\overline{A + B + C + \cdots} = \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \cdots$.

and

$$\overline{A \cdot B} = \overline{A} + \overline{B} \tag{5.12}$$

which often makes it possible to simplify Boolean expressions and thereby simplify the logic. Some of the most important algebraic rules for Boolean functions are shown in Table 5.8.

Table 5.8: Algebraic rules for Boolean functions

Associative	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C = A + (B + C$
Distributive	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
Idempotent	$A \cdot A = A$	$A + A = A$
Double Negation	$\overline{(\overline{A})} = A$	—
DeMorgan's	$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} + \overline{B}$
Commutative	$A \cdot B = B \cdot A$	$A + B = B + A$
Absorption	$A + (A \cdot B) = A$	$A \cdot (A + B) = A$
Bound	$A \cdot 0 = 0$	a
Negation	$A \cdot (\overline{A}) = 0$	$A + \overline{A} = 1$

5.9.1 Simplifying/Constructing Functions

A function can be expressed as a logic (circuit) diagram, truth table or expression. Logic diagrams show how the individual gates are interconnected. Examples of truth tables are shown in Tables 5.9 and 5.10. The number of possible functions given n inputs and m outputs can be expressed as

$$\mathrm{N} = 2^{m2^n} \tag{5.13}$$

so that for 2 inputs and one output there are $2^2 = 4$ functions, for two inputs and two output there are $2^8 = 256$ functions, for 3 inputs and two outputs there are $2^{16} = 65{,}536$ functions, etc.

In order to optimize a logic diagram for which there can be many different implementations possible, the designer can begin with a truth table, for example consider the truth table, shown in Figure 5.9, for the function F as defined in Equation (5.14).

Table 5.9: A simple truth table.

A	B	$A \cdot B$	$A + (A \cdot B)$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

$$
\begin{aligned}
F = A + A \cdot B &= A \cdot 1 + A \cdot B && (5.14)\\
&= A \cdot (1 + B) && (5.15)\\
&= A \cdot 1 && (5.16)\\
&= A && (5.17)
\end{aligned}
$$

Note that the initial expression for F with a requirement for two gates could be simplified resulting in an implementation requiring no gates. A more complex case whose truth table , shown in Table 5.10, is illustrated next.

Consider the following:

$$F = A \cdot \overline{B} + A \cdot B + B \cdot C \tag{5.18}$$

By employing the distributive, inverse, and identity properties together with DeMorgan's theorem the function can be significantly simplified, e.g., Equation (5.18) can be expressed as

$$
\begin{aligned}
F &= A \cdot (\overline{B} + B) + B \cdot C && (5.19)\\
&= A \cdot 1 + B \cdot C && (5.20)\\
&= A + B \cdot C && (5.21)
\end{aligned}
$$

which reduces the number of gates required to implement this function by 50%.

And finally, a still more complex example is given by

$$
\begin{aligned}
F &= \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C && (5.22)\\
&= \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C + A \cdot B \cdot C + A \cdot B \cdot C && (5.23)\\
&= \overline{(A} \cdot B \cdot C + A \cdot B \cdot C) + (A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot C) + (A \cdot B \cdot \overline{C} + A \cdot B \cdot C) && (5.24)\\
&= (\overline{A} + A) \cdot B \cdot B \cdot C + (\overline{B} + B) \cdot C \cdot A + (\overline{C} + C) \cdot A \cdot B && (5.25)\\
&= B \cdot C + C \cdot A + A \cdot B && (5.26)
\end{aligned}
$$

which reduces the number of gates from 14 to 5.

Table 5.10: A more complex example.

A	B	C	$A \cdot B$	$A \cdot \overline{B}$	$B \cdot C$	$A \cdot \overline{B} + A \cdot B + B \cdot C$	$A + B \cdot C$
0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	1	0	0	0
1	1	0	0	0	0	1	1
0	0	1	0	0	0	0	0
0	1	1	0	0	1	1	1
1	0	1	0	1	0	1	1
1	1	1	1	0	1	1	1

5.9.2 Karnaugh Maps[35]

Karnaugh maps can be used to convert truth tables and logic equations into logic diagrams and as a substitute for both. In addition, Karnaugh maps make it possible to simplify logic diagrams. Consider the truth table shown in Table 5.11. A Boolean function can be expressed as a sum-of-products derived from the corresponding Karnaugh map (K-Map) shown in Table 5.12 and by inspection is found to be

$$F = \overline{A}BCD + A\overline{B}CD + AB\overline{C}D + \overline{ABC}D \tag{5.27}$$

where each square of the Karnaugh map represents one row of the truth table. Note that the Karnaugh map is configured with respect to variables in a manner that allows only one variable to change as you move from one cell to another, whether horizontally or vertically, i.e., $\overline{AB}$, $\overline{A}B$, AB, $A\overline{B}$ and not $\overline{AB}$, $A\overline{B}$, $\overline{A}B, AB$, because $A\overline{B} \Rightarrow \overline{A}B$ is a change in two variables.[36] Any "cell" of the K-Map containing a one represents what is referred to as a *minterm*, i.e., a product term of N variables.

The process involved is largely a mechanical one as opposed to manipulating Boolean expressions, and is considerably simpler. As a practical matter this technique is useful for expressions of six, or less, variables. If more than six variables is involved the Quine-McCluskey (Q-M) methodology is preferable.[37]

The Q-M algorithm offers a number of advantages:

- There is no limitation on the number of input variables
- It always finds the "prime implicants"[38]
- The algorithm can be applied in the form of a computer program

Both K-Map and Q-M rely on a very simple expression, viz.,

$$A \cdot B + A \cdot \overline{B}$$

and it follows that

$$A \cdot B + A \cdot \overline{B} = A \cdot (B + \overline{B}) = A \cdot 1 = 1 \cdot A = A \tag{5.28}$$

It is this simple relationship that forms the basis for the Karnaugh map algorithm.

The following steps allow a K-map to be used to simplfy a Boolean expression:

1. Draw a "map" in the form of a table with each product term represented by a cell in the table. The cells must be arranged so that moving from one cell to another either horizontally or vertically changes one and only one variable.
2. Place a check mark in each box whose labels are product terms and their respective complements.

[35] Some of the material in this section is based in part on examples provided by Bob Harbort and Bob Brown, Computer Science Department, Southern Polytechnic State University and reproduced here with their permission.

[36] This type of arrangement is sometimes referred to as Gray coding for a binary system in which any two successive values, eg.g. bytes, differ by only one bit.

[37] The Quine-McCluskey algorithm is based on two fundamental properties of Boolean expressions: 1) $A \cdot \overline{A} = 1$ and 2) the distributive law. In addition to being implementable as an efficient computer algorithm, it provides a way to confirm that the resulting Boolean function is in *minimal* form.

[38] The a product term of a Boolean function F is a *prime implicant* iff the function's value is 1 for all minterms of the product term. In terms of K-maps, a prime implicant is any loop that is fully expanded. An *essential prime implicant* is any loop that does not intersect any other loop.

Table 5.11: Example Truth Table

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Table 5.12: Corresponding K-map

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	0	1	0	0
$\overline{A}B$	0	0	1	0
AB	0	1	0	0
$A\overline{B}$	0	0	1	0

3. Draw loops around each horizontally, or vertically, adjacent pairs of check marks.[39]
4. For each loop, form an unduplicated list of the terms. Multiple instances of a literal should be reduced to one instance, and a literal and its complement in the list should be deleted from the list.
5. Form the Boolean product of the terms remaining after step 4.
6. Form the Boolean sum of the products resulting from step 5.

The following simple example will illustrate the procedure outlined by steps 1-6. Assume that the expression to be simplified is $A \cdot B + A \cdot \overline{B}$ which is an expression with two variables, A and B. A table is drawn as shown in Figure 5.27. Check marks have been placed in the cells representing the AB and $A\overline{B}$ product terms. The loop drawn around these two cells contains A, B, A and $\overline{B}$. The B and $\overline{B}$ cancel and the duplicated A's are reduced to a single A. The end result is: $AB + A\overline{B} = A$. Next consider the expression $A \cdot \overline{B} + \overline{A} \cdot B + \overline{A} \cdot \overline{B}$. IN this case check marks are placed as shown in Figure 5.28. Two loops imply that there will be two terms in the simplified expression. The vertical loop yields $\overline{A}, B, \overline{A}$ and $\overline{B}$ and the horizontal loop contains $A, \overline{B}, \overline{A}, \overline{B}$ which are reduced to $\overline{A}$ and $\overline{B}$, respectively. After removing duplicates and invoking the inverse law the expression reduces to

$$A \cdot \overline{B} + \overline{A} \cdot B + \overline{A} \cdot \overline{B} = \overline{A} + \overline{B} \tag{5.29}$$

A third example illustrates the simplification of a three variable Boolean expression, viz.,

$$\overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$$

[39] A cell may be in more than one loop and a single loop can span multiple rows, columns or both, provided that the number of enclosed check marks is a multiple of 2, e.g., 1, 2, 4, 8, 16,...

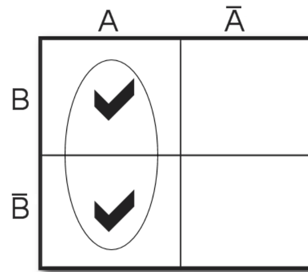


Figure 5.27: Karnaugh map for $A \cdot B + A \cdot \overline{B}$

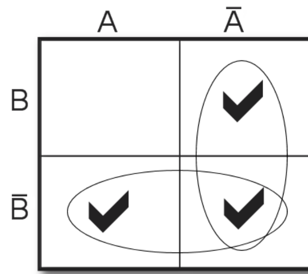


Figure 5.28: Karnaugh map for $A \cdot \overline{B} + \overline{A} \cdot B + \overline{A} \cdot \overline{B}$

The K-map for this example is shown in Figure 5.29. This example involves a *toroidal* loop

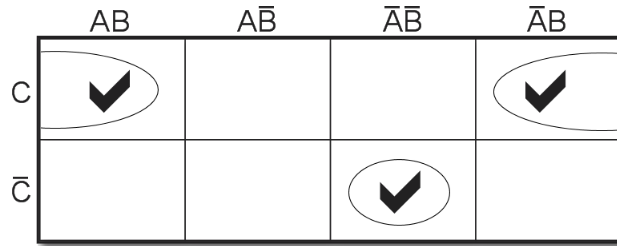


Figure 5.29: Karnaugh map for $\overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$

encompassing cells ABC and $\overline{A}BC$. The simplified expression in this case is given by

$$F = B \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{C} \tag{5.30}$$

Next consider a SoP with five product terms each consisting of three variables, viz.,

$$F = \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C \tag{5.31}$$

The K-map for this example is shown in Figure 5.30, and the equivalent logic circuit is shown in Figure 5.31. The truth table representing this configuration is shown in Table 5.13.

After removing redundant instances of variables and variables and their complements have been removed, the expression is reduced to $A \cdot C + B$ and can be implemented in discrete logic as shown in Figure 5.32.

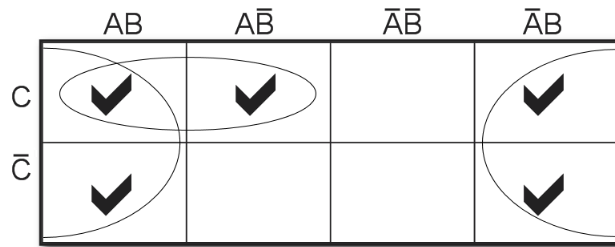


Figure 5.30: A five product K-map.

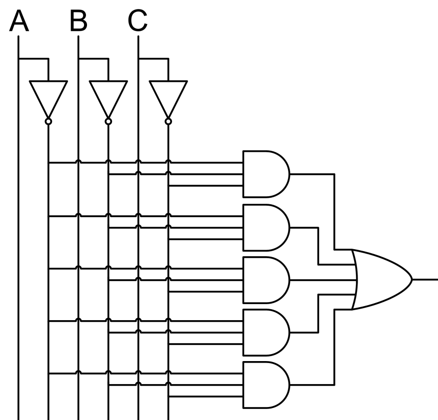


Figure 5.31: Logic circuit for a five term SoP expression.

Table 5.13: Truth Table for Eq. 5.31

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

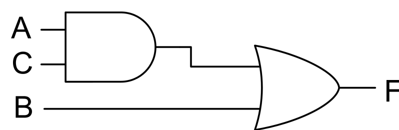


Figure 5.32: Simplified version of the logic circuit in Figure 5.31.

The loops in a Karnaugh map should be made as large as possible subject to the constraint that the number of check marks within any given loop must be an integer multiple of two. When a Karnaugh map consists of more than two rows, it represents more than three variables and the top and bottom edges are treated as adjacent. A loop that is within loops is not considered because all of its terms have been accounted for in the other loops.

5.10 Combinatorial Circuits

A combinatorial circuit is defined as any combination of the basic operations AND, OR and NOT that includes both inputs and outputs. Each of the outputs is related to a unique function. A classic example of a combinatorial circuit is the "half-adder" which is capable of producing a 1-bit sum, and carry, based on the following functions:

$$\text{Sum} = A' \cdot B + A \cdot B' \tag{5.32}$$

and the resulting carry bit, if any, by

$$\text{Carry} = A \cdot B \tag{5.33}$$

However, although a half-adder can generate a carry it does not have the ability to add a *carry in* (C_i) to the sum. This capability is embodied in what is referred to as a full-adder which can be represented in terms as

$$\text{Sum} = ABC_i + AB'C_i^{'} + A'BC_i^{'} + A'B'C_i \tag{5.34}$$

and *carry out* (C_o) by

$$C_o = AC_i + BC_i + AB \tag{5.35}$$

Thus a half-adder cascaded with an *n-1* number of full-adders can be used to add n bits. However, because combinatorial circuits are employed, some form of memory is needed. This is because for combinatorial circuits, any change in an input results in a change in the outputs[40] and therefore the circuits are *memoryless*.

Fortunately, it is possible to create a very simple memory device from the same basic building block as the logic functions, i.e., from a NAND gate as shown in Figure 5.33. This configuration is a two state, or *bistable* configuration for which R and S are normally both set to 1. If input R or S is toggled momentarily, then Q and Q' are forced into opposite states and will remain there until one of the inputs is toggled again. However, if both inputs are set to zero contemporaneously, Q and Q' are forced into the 1 state.

A simple modification of this circuit resolves this potential problem and requires the flip-flop to operate in a synchronous manner when changing state. Figure 5.34 shows the modification which involves the addition of three NAND gates, one of which is configured as an inverter. This configuration is known as a data or D flip-flop and has a clock input (Clk) which allows the flip-flops operation to be synchronous. A clock input, 0 –1– 0, will cause the data input to be copied to the Q output where it will be "latched" i.e., retained, until the next clock pulse. Flip-flops can be combined in either parallel configurations to function as memory into which bits may be stored and retrieved in parallel, e.g. as in the case of conventional registers or in a daisy-chain to function as shift registers.[41]

[40] There is of course some finite amount of propagation delay through a logic circuit, but for the purposes of the present discussion such delays will be ignored. However, it should be noted that propagation delays are often cumulative and in such cases it may not be appropriate to ignore them.

[41] Flip-flops are discussed in more detail in section 5.11.

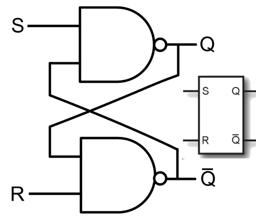


Figure 5.33: A NAND gate implementation of a RS flip-flop.

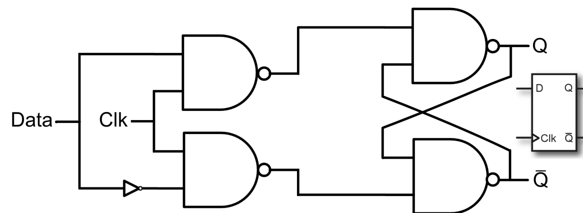


Figure 5.34: A D flip-flop.

5.11 Sequential Logic

Unlike combinational logic which has no internal state and whose output depends solely on the state of the input, at any given time, sequential logic output is a function of its internal state at any given time and inputs. Sequential logic is "clock-based" and relies on a combination of combinational logic and one, or more, flip-flops as shown in Figure 5.35. Flip-flops provide a

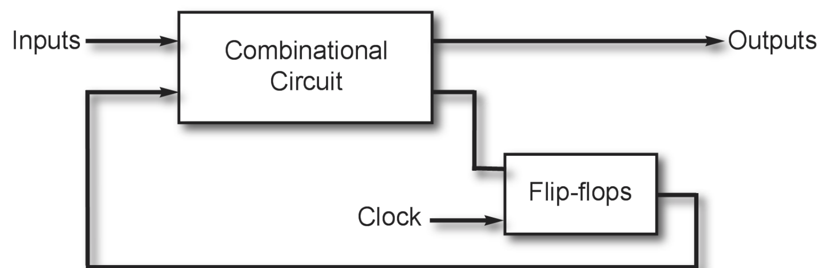


Figure 5.35: Sequential circuit block diagram.

mechanism for remembering a state and thus for storing data, a capability not achievable by using combinational logic alone. The simplest example of sequential logic is the flip-flop, which can be connected in various configurations, e.g., counters, timers, registers, RAM, etc. Counters are sequential circuits that have a clock signal as their input. The electronic flip-flop was invented by F.W. Jordan and William Eccles in 1919 as a bistable device consisting of two vacuum tubes[42]. The simplest from of flip-flop is the SR flip-flop, sometimes referred to as the SR latch. Its inputs consisted of S(et) and R(eset). The truth table for this device is shown in Table 5.14. A JK flip-flop, shown in Figure 5.36, is similar to a D flip-flop except that indeterminate states are avoided for cases in which both inputs are being held high by requiring that in such cases, the

[42]Sometimes referred to as a bistable multivibrator. Monostable (pulse) and astable (oscillator) functions using two vacuum tubes in similar configurations were also possible.

output "toggles"[43] with the clock, cf. Table 5.15.

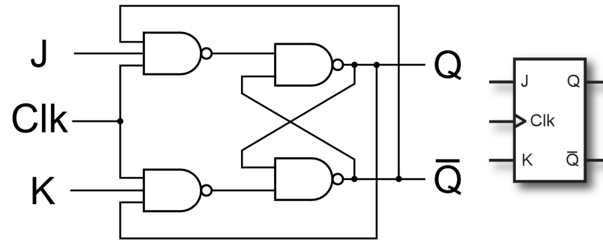


Figure 5.36: A JK flip-flop.

In some cases, additional control pins are provided for JK flip-flops which allow asynchronous clearing and presetting. The D, or data flip-flop, has the same value as the input when a clock "edge" occurs[44], as shown in Table 5.16.

Table 5.14: Truth Table for a RS Flip-flop

S	R	Q	$\overline{Q}$
0	0	Unchanged	Unchanged
0	1	0	1
1	0	1	0
1	1	Indeterminate	Indeterminate

Table 5.15: Truth Table for a JK Flip-flop

J	K	Q	$\overline{Q}$
0	0	Unchanged	Unchanged
0	1	0	1
1	0	1	0
1	1	Toggle	Toggle

Flip-flops can be configured in a variety of ways to provide very useful functionality, e.g., four flip-flops can be configured, as shown in Figure 5.37, to provide a register that accepts serial data as input and makes the data available in a parallel output format. Each time a clock pulse occurs data stored in each of the flip-flops will be moved one bit position to the right. The bit asserted at the input will move to the first flip-flop and be accessible at output Q0, the data formerly stored in the first flip-flop will move to second flip-flop and be accessible at the Q1 output and so on. Thus the bit stored originally stored in the fourth flip-flop is lost. Obviously, the number of clock pulses applied must correspond to the number of bits being added to the shift register

[43]Toggle refers to changing the state of a two state device, i.e., causing it to change to the other state, by some event, or action,

[44]Some D flip-flops are positive- and some are negative-edge triggered.

Table 5.16: Truth Table for a D Flip-flop

Clock	D	Q	$\overline{Q}$
Edge	0	0	1
Edge	1	1	0
Non-edge	Unchanged	Hold	Hold

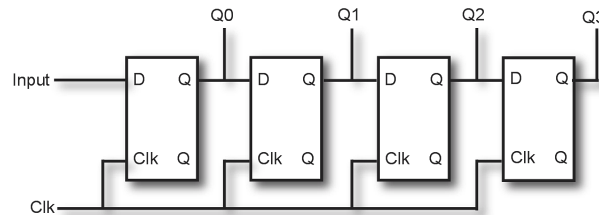


Figure 5.37: A serial shift register using D flip-flops (SIPO).

for storage. This type of register may be used as a method of converting serial input to parallel output (SIPO) and/or as a storage location for four bits.

Alternatively the four flip-flops can also be configured, as shown in Figure 5.38, to provide parallel input and parallel output (PIPO). Each time a clock pulse occurs the four input bits are stored in their respective flip-flop locations and appear in parallel on Q0, Q1, Q2 and Q3 respectively. If the input data is from a 4-bit parallel data bus, this configuration provides a way to retain bus data in a 4-bit register. This is a common technique for temporarily storing data in a microprocessor.

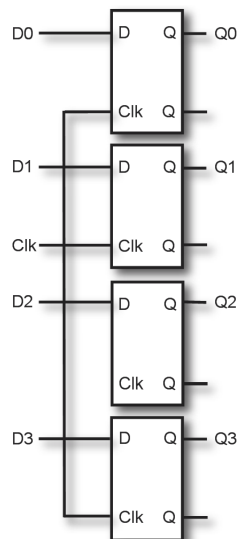


Figure 5.38: A parallel input/output register using D flip-flops.

An important consideration when configuring groups of flip-flops that share a common clock

signal is related to the phenomenon known as *clock skew.* This occurs when a device such as a flip-flop is edge triggered and the clock edge does not arrive at the clock input for each of the flip-flops at the same time. Clock skew can arise as a result in differences in the paths that the clock signal must traverse to reach the clock inputs of the flip-flops. It can also be compounded by the use of a group of flip-flops that are triggered on different edges of the clock signal. Also in some configurations of flip flops, gates are required in order to achieve the required sequential logic. In such cases the gates introduce delays if they are in the clock path. In addition to the spatial variations in clock edges i.e., clock skew, temporal variations can occur as well, the latter being referred to as *clock jitter*. Obviously, in a given situation, clock skew is not subject to variation from one clock signal transition to another for any given device, whereas, clock jitter can and often does vary as a function of time on a cycle-by-cycle basis. Clock skew can be introduced not only by path length differences but also by power supply, temperature and clock driver variations. Jitter can arise as a result of variations in the clock source, power supply/temperature variations and capacitive loading and/or coupling.

Flip-flops are often used as *static* memory devices in that they are bistable devices which are capable of storing a given state for as long as power is maintained as opposed to dynamic memory devices that store a state in the form of charge on parasitic capacitors and require continuous refreshing. While simpler and cheaper to manufacture dynamic memory is subject to noise adversely affecting their ability to retain a given state.

Some logic devices are limited to a maximum of two inputs. This restriction can be overcome by combining several devices as shown illustratively in Figures 5.39 and 5.40.

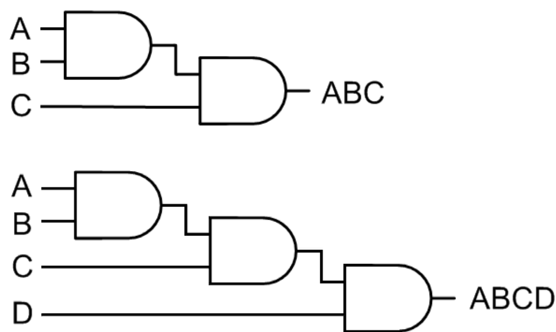


Figure 5.39: Three and four input configurations for AND gates.

PSoC Creator allows the designer to configure logic devices such as those shown in Figure 5.41. However, this type of expansion of inputs obviously increases propagation time and therefore latency and can result in race[45] problems, an example of which is shown in Figure 5.42.

Assume that input A was asserted previously at a point in time, $t<0$, sufficient to allow the logic shown in the figure to reach a steady state condition. If at $t = 0, A = 0$ is asserted then after a propagation delay of Δt_1, introduced by the inverter, $\overline{A} = 1$, as shown. This means that for duration of Δt_1, both A and $\overline{A} = 1$. This produces a pulse of width Δt_1 displaced in time by Δt_2. It is this pulse that is referred to as a "glitch" and said to be caused as a result of "race" conditions in the signal path.

[45]Race conditions can occur in logic circuits as a result of propagation time differences that result in an output changing to an inappropriate state, often referred to as a "glitch", caused by a delay in one or more input signals with respect to other inputs to the circuit.

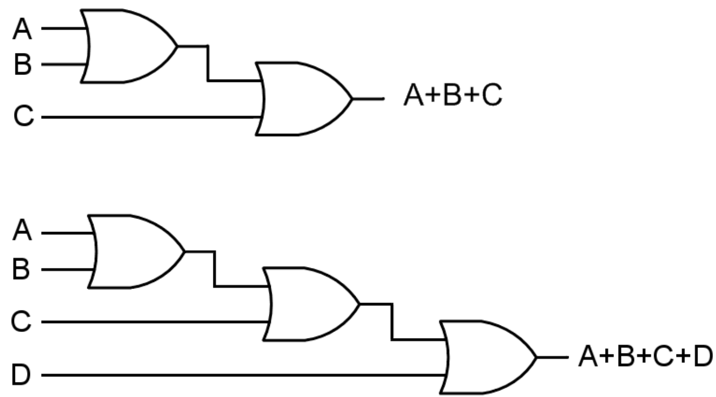


Figure 5.40: Three and four input configurations for OR gates.

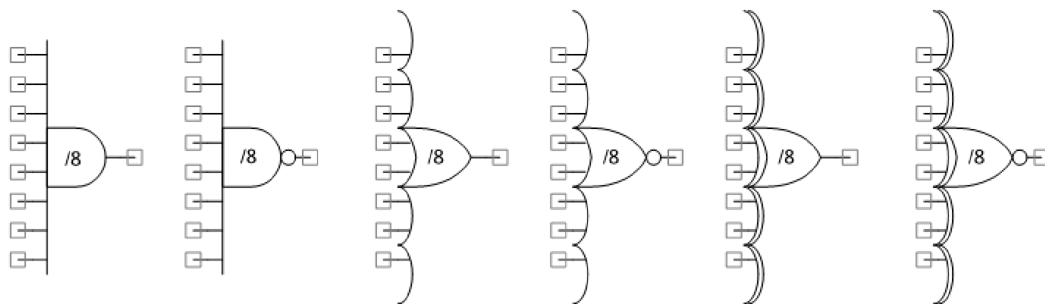


Figure 5.41: Examples of multiple input gates available in PSoC Creator.

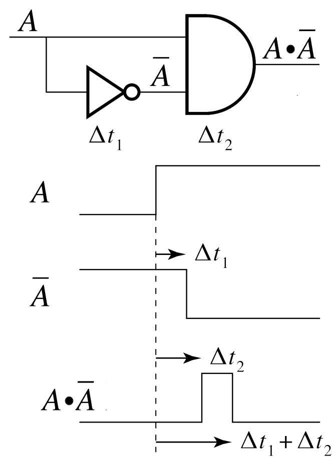


Figure 5.42: A simple example of a race condition resulting in a “glitch”.

5.12 Finite State Machines

The concept of a finite-state machine (FSM) is an abstraction of a system whose allowed states are restricted to only one of a finite number of states at a time and the "transitions"[46] between those states.[47] A transition between states occurs only as a result of inputs, sometimes referred to as being *event-driven.* While a given state machine may produce an output, or outputs, some state machines do not. It may be the case that the result of a state transition is simply to place the system in a different state. Some machines may have an *error state* to handle unanticipated and/or unexpected inputs. Once a state machine enters an *error state* it remains there, even in the presence of subsequent inputs. Transitions are governed by so-called"rules" or "conditions" and typically these are expressed in the form of case statements. Transitions are triggered by "events" which may be either external or internal. Switch statements and state tables are commonly use to implement FSMs.

FSMs are frequently used in natural language processing, text processing, cellular automata, natural computing, electronic design automation, communications, artificial intelligence, video games, vending machines, traffic control, speech recognition, speech synthesis, parsing, web applications, neurological system modeling, protocol design, process control, vending machines and many other applications.

FSMs are defined in terms of the:

- allowed states
- input signals,
- output signals,
- next-state function,
- output function,

and,

- an initial state.

and, as a result, FSMs are *sequential* machines.

The Moore state machine, shown schematically in Figure 5.43, has the characteristic that outputs are independent of inputs, i.e., outputs are created within a given state and can only change with a change of state.[48] State assignments may be either arbitrary, or specified. Arbitrary state assignments depend on either combinatorial, or registered, decoded state bits. Specified state assignments are based on either state bits, or on so-called "one-hot" encoding, e.g. as shown in Table 5.17.[49] For state machines using one-hot encoding, n flip-flops, often referred to as the *state memory*, can be used to represent the n states of the FSM.

The *state vector* is the value currently stored in the *state memory.* Moore FSM outputs are a function of the state vector, but the outputs of a Mealy FSM, shown in Figure 5.44, are a function of the inputs and the*state vector.* While this method does require n flip-flops to encode and decode the FSM's current state, decoding is simplified by virtue of the fact that no other logic is required to determine the current state of the machine. The outputs of Mealy machine are determined either by the present state, or by a combination of the current state and the then

[46]Transition" in the present context refer simply to the change from one state to another.
[47]In theory, any system utilizing memory can be treated as a state machine.
[48]Although inputs can cause a change of state, they do not determine the state to which the FSM moves.
[49]"One-hot" refers to the case in which given a string of bits, only one can be non-zero, e.g., 00010000. The inverse situation is referred to as one-cold, e.g., 11101111. On-hot code is often used decoder, ringer counter and some state machine implementations.

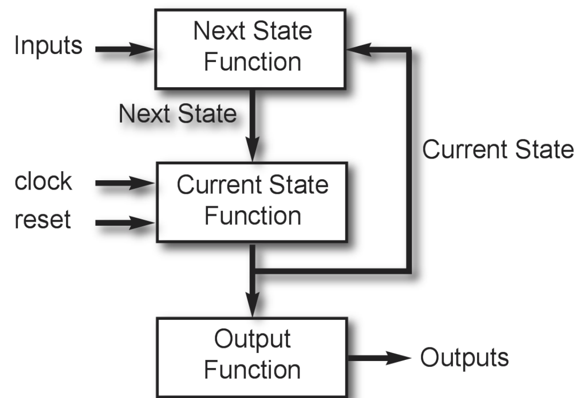


Figure 5.43: Moore state machine.

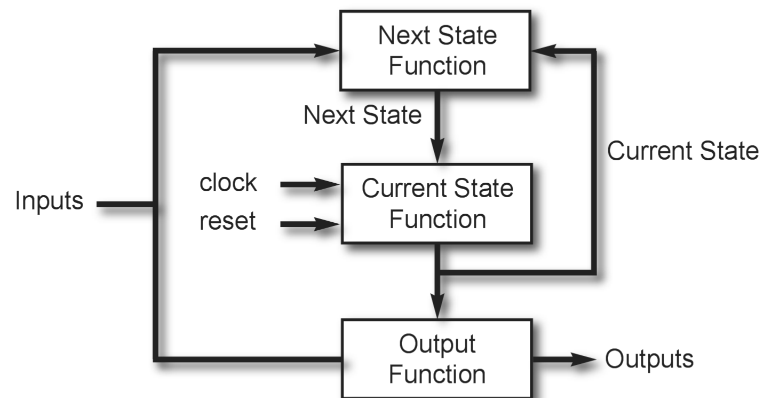


Figure 5.44: Mealy state machine.

Table 5.17: One-hot versus Binary Encoding

State	Binary Encoding	One-Hot Encoding
S0	00000001	00000001
S1	00000010	00000010
S2	00000011	00000100
S3	00000100	00001000
S4	000001001	00010000
S5	000001010	00100000
S6	00001011	01000000
S7	00001111	10000000

current inputs.[50] Some applications employ both Moore and Mealy FSMs and as a practical matter similar FSMs can be functional equivalents.

State machines are often represented by *state graphs* as shown in Figures 5.45 and 5.46. By convention, arcs and/or straight lines are used to represent state transitions and each node represents a specific state. In the case of *self-transitions*, the source state and target state are the same states. Moore outputs are given within the circle or "bubble" representing the state. Mealy outputs are shown on the associated arc or line.

State machines can also be represented as algorithmic state machines (ASMs) in which case the graphical representation is in the form of a flow chart with state, decision and conditional boxes. ASMs can be recast as state graphs and vice versa.

Typically, Mealy machines:

- typically have fewer states than that of their Moore counterparts,
- react faster to inputs,[51]
- has outputs that are a function of both current state and inputs that can change asynchronously,[52]
- outputs can change asynchronously,
- can have fewer states than a Moore FSM,

and,

- can sometimes introduce delays in critical paths.

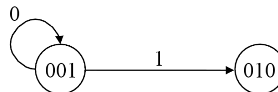


Figure 5.45: A very simple finite state machine.[53]

PSoC3/5 are quite capable of implementing state machines and this is facilitated in part by the fact that PSoC Creator supports look up tables (LUTs). LUTs have the characteristic that a particular combination of input values results in the output of a specific combination of outputs.

[50] It should be noted that Mealy FSM outputs can change asynchronously.
[51] Moore machines have to wait for the next clock cycle before changing state.
[52] This can give rise to "glitches". Moore FSMs do not produce glitches.

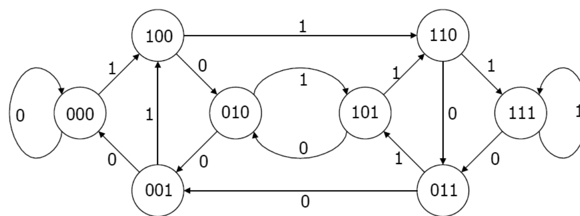


Figure 5.46: A more complex FSM representing a shift register.

This allow a LUT to provide virtually any logic function and in the case of PSoC Creator, each LUT component is configurable to have as few as one input inputs and one output, e.g., as *in0* and *out0* or as many as five inputs and eight outputs as *in0, in1, in2, in3, in4* and *out0, out1, out2, out3, out4, out5, out6, out7,* respectively. The default configuration is two inputs and two outputs.

Registering the outputs is accomplished by simply clicking on a check box in PSoC Creator's*Configure 'LUT'* dialog box, cf. Figure 5.47. Registering the outputs and routing some on the outputs back to the inputs allows state machines to be implemented. The actual implementation of LUTs is based on logic equations stored in the PLDs. LUTs save the designer the trouble of having to create them using combinatorial logic components and by registering a LUT it can be used to implement sequential logic. The registering of the outputs causes the LUT to register the output on the rising edge of the LUT's clock input. The clock speed should not

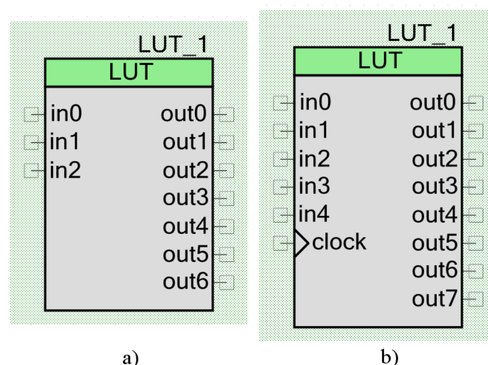


Figure 5.47: a) Unregistered versus b) registered PSoC3/5 LUT.

exceed 33 MHz if any of the LUT's outputs are connected to I/O. It should be noted that the LUT is implemented as a hardware-only design and therefore there is no LUT API.

As an example, consider a rising edge detector implemented as a Moore state machine, as shown in Figure 5.48, that produces a pulse each time a rising edge is detected[23]. Creation of a LUT-based state machine begins with the creation of a table which contains each possible *state* and all possible combinations of inputs. Next consideration turns to defining the *next state* for each state and the associated inputs. It is then possible to create the LUT. Once this table has been completed, its entries can be entered into the *Configure 'LUT'* dialog box in PSoC Creator. The implementation of the edge detector LUT is shown in Figure 5.49.

E= Edge Input

If E=1 at the Clock edge, then go to state 01

E=1

E=1

S0
Low input,
Waiting for
rise
(Output=0)

S1
Edge
Detected?
(Output=1)

S2
High input,
Waiting for
fall
(Output=0)

E=0

E=1

If E=0 at the Clock edge, then stay in state 00

E=0

E=0

E=0,
E=1

S3
Illegal State

This is a dummy state to complete the state transitions.

Note: The state machines should have minimum no. of state flops, because this minimizes the no. of illegal states. This is to ensure that if the machine malfunctions and makes an illegal transition, at least the erroneous destination will be a legal state, and the machine can recover.

Figure 5.48: An edge detector implemented as a state machine.

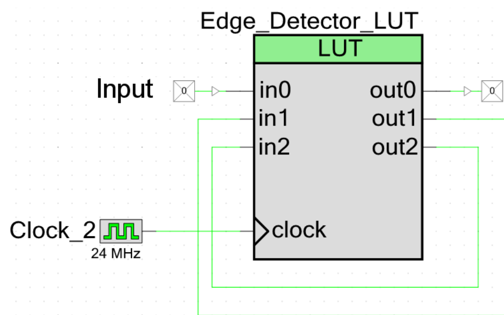


Figure 5.49: A PSoC3/5 implementation of an edge detector as a FSM.

5.13 Hardware Description Languages (HDL)

VLSI digital circuits often involve hundreds of logic cells and perhaps thousands of interconnections.Therefore the associated difficulty of developing such PDL applications capable of performing complex functions has given rise to what are termed hardware description languages which allow the designer to model digital systems. These HDL languages are supported by development environments that typically have the ability to provide schematic design, simulations/verification and the ability transform the design into a "configuration file and then download it to the targeted device. The HDL form of a design is a temporal and spatial description of the design and includes expressions that formally describe the design's digital logic circuits. The descriptions are text-based, include explicit time dependencies, and take into account interconnections between blocks that are expressed in a hierarchial order. The process of converting from the description of a logic circuit to an implementation defined in terms of gates is referred to as *synthesis*. The output of the synthesis process is a *netlist*[54].

An ideal HDL should be capable of supporting designs involving tens of thousands of gates, provide high level constructs for describing complex logic, support modular design methodologies and multiple levels of hierarchy, support both design and simulation, be capable of producing device-independent designs, support schematic capture as well as HDL descriptions, etc. In the discussion that follows several HDLs are discussed, each of which, to a greater or lesser degree meet these criteria.

5.14 Design Flow

Various modelling techniques are employed in designing embedded systems, e.g., dataflow for systems involving parallelism and signal analysis, discrete-event for systems involving explicit time dependencies, state machines for systems based on sequential decision logic, time-driven for systems that involve periodic and/or time dependent action, continuous time for systems involving dynamics, etc. HDLs are capable of facilitating each of these approaches in a wide variety of cases, but may have some limitations with respect to certain types of applications.

A design can begin with a graphical representations of the logic circuits, i.e., a schematic, or a purely text-based description of the design. Available tools often include both schematic and HDL editors and when the HDL, or schematic design, phase has been completed, it is then possible to simulate the design for the purposes of behaviorial evaluation, conducting preliminary performance evaluations, creation of test vectors, etc. Test bench waveforms can also be introduced as part of the simulation process. Following the synthesis phase of the design it is possible to carry out additional simulation tests to verify the performance of the logic design including timing, which although limited in scope in some cases, can still provide important information about the design.

Once the descriptions, constraints and netlists have been created, they can then be merged into a database so that the *place and route process* can be invoked. In modern development environments the floorplan and routing are displayed graphically providing the designer with additional control over the design by instituting manual changes in the design.

So detailed and complete are these descriptions that they can be used in conjunction with simulators that use the descriptions as input to study the corresponding circuit's behavior and performance in complete detail. Furthermore, once the modeling is completed using simulators, the descriptions can be employed as input to CAD tools to synthesize hardware designs. HDLs

[54] A netlist is a text-based description of the gates used in the design and their interconnections.

are used to create formal descriptions of digital circuits. Some simulators can actually interact with hardware implementations of the design to further optimize the system under design.

VHSP/*VHDL* was originally designed to ease the documentation challenges of *ASIC* designs but was soon found to be to facilitate the design of very high-speed integrated circuits ad is referred to as *VHSIC*. VHDL is a VHSIC hardware description language based on ADA and developed by the Department of Defense beginning in 1983.

5.14.1 VHDL

While some may choose to characterize VHDL as simply "yet another programming language" it is in reality much more. It came into being as a result of a government initiative in 1980 by the Department of Defense (DoD) and was originally intended to be a formal methodology for describing digital circuits. However, it soon became apparent that its scope could be extended to allow it to serve not only as a language standard for digital circuit descriptions but for simulation of digital circuitry as well. VHDL has gone through a number of reincarnations[55] and its notation is defined in each case by a language reference manual (LRM). It is regulated by the IEEE and is maintained as an international standard. VHDL supports both top-down/bottom up and as some have suggested even "middle-out".[2]

The VHSIC[56] hardware description language (VHDL) offers a designer a number of important benefits in developing new designs, particularly those that involve tens of thousands of logic gates. For example, VHDL supports very sophisticated and powerful constructs for describing complex logic, a modular design methodology, multiple levels of hierarchy and a VHDL description can used for both design and simulation. The resulting designs are device-independent and therefore highly portable so that the designer can select a the optimum vendor, device and synthesis.

A designer can begin with a very high level abstraction for a design, use VHDL to develop an architecture for the design and then decompose that structure into subsystems, sometimes referred to as "sub-designs". These subsystems can often in turn be decomposed further into subsystems of subsystems until one finally arrives, if required, at the equivalent of a standard set of basis modules, e.g., commonly available integrated circuits.

A simple module at the lowest level of this hierarchy might well consist of a device, or devices, with two inputs and one output, e.g, a gate. Such a module, referred to as an instance of *entity*, need not be decomposed any further and can be treated strictly in terms of its characteristics, i.e. the relationship(s) between the input and output signal levels, propagation delays, etc. [1] Modules at the base of such a hierarchy are typically described in *behaviorial*, or *functional* terms. However, if the basis modules employ feedback, the behaviorial/functional module descriptions become complex. Fortunately VHDL is designed to address this situation, as well.

VHDL is based on constructs such as *architectures*, *configurations entities*, *packages* and the corresponding *package bodies*. Architectures are functional descriptions of modules, configurations that define the architecture and entities required to build a model. Entities define interfaces and often involve a port list. Packages contain the definitions of data types such as constants, various data types and *subprograms*[57]. In addition, process code, which is a sequence of statements executed in a defined order, is employed as a concurrent object.

VHDL supports both sequential and concurrent statements. Sequential statements are con-

[55] While there are a number of versions of VHDL extant e.g., VHDL'87, VHDL'93, VHDL'2000, VHDL'2008, etc. version.VHDL'93 remains the most widely used version.

[56] Very-high-speed integrated circuit.

[57] Subprograms in VHDL are the analog of functions in C.

tained within functions, procedures or process statements. *If-then-else*, *case* and *loop* statements are examples of sequential statements. Concurrent statements are occur within the architecture in the form of statements containing concurrent procedure calls, signal assignments and component instantiations.[58]

Signals are used to transmit information between design statements, specifically between entities and processes. They are treated as globals in architectures and blocks. When declared in a PORT, signals must be assigned a direction. However, if they are declared in an architecture, block or package, no direction is required. Signal assignments typically include specifications, i.e., time expressions of the delay time that must occur before the signal is allowed to assume a new value. If the time expression is not specified the default value is zero femtoseconds.[59] It should be noted that variable updates occur immediately in VHDL while signal updates occur after a delay or at the end of a process.

The design flow diagram, shown in Figure 5.50, illustrates the various stages in development of a VHDL-based design. The basic steps in the design flow are as follows:

1. *Design entry* - this is often done with in the context of a computer-aided design (CAD) tool, and results in the design being available in a machine-readable format.
2. *Functional simulation* - step 1 produces the design description and this step simulates the design to confirm that the design does in fact meet the requirements specification. This is often referred to as a behaviorial, or functional simulation[60]and its purpose is to verify that the behavior is correct from a logic perspective.
3. *Synthesis* - a CAD tool is employed for this step to interpret the VHDL description and employ a sufficient set of standard building blocks, e.g., LUTs, multiplexers, registers, adders, etc., to implement the design. The step produces a netlist[61] which will be used by the next step, viz.,*Implementation*.
4. *Implementation* - this step involves the invoking of a translate phase (TRANSLATE) which translates the netlist produced by the previous step into a format consistent with the targeted device. A mapping process (MAP) maps the standard set of blocks used by the synthesis process onto the available devices in the target hardware. This is followed by allocation of the target resources and routing of all of the required interconnections between these resources (PLACING and ROUTING). At this juncture, because the actual propagation and other types of delays have been taken into account, it is possible to carry out what is referred to as the POST-PLACE and ROUTE simulation which represents a modeling of the actual behavior of the physical design.
5. *Programmer Download* - At this stage the design has been verified and is ready for downloading to the target device. This is accomplished by the creation of a file that is downloaded in serial fashion to a programmer for the target device.[62]

[58]There is a distinction to be made between concurrency and parallelism in that concurrency refers to parts of a program that at the conceptual level are to be executed simultaneously, i.e. logical concurrency. Parallelism implies that certain parts of a program are in fact executed simultaneously at the hardware level. Programs that execute instructions sequentially are therefore non-concurrent. Some compilers are capable of carrying out a dataflow analysis and outputting parallel code for hardware capable of supporting parallelism.

[59]fs is a acronym for a femtosecond and represents $1x10^{-15}$ of a second.

[60]It is important to bear in mind that this type of simulation does not any of the physical implementation details, e.g., actual propagation and other types of timing delays introduced by the various physical components involved is not taken into account by this level of simulation. Therefore while the simulation may be regarded as a necessary step, it is hardly sufficient, in and of itself. It is however, possible to include at this level statements which assign values for various delays, but these are of course noth synthesizable and are included merely to reflect the impact of the delays on behavior.

[61]In the case of PLDs and CPLDs a sum-of-products equations may be produced instead of a netlist.

[62]This step should be regarded as nonlinear in the sense that system performance can depend in a nonlinear manner on the components selected by this process.

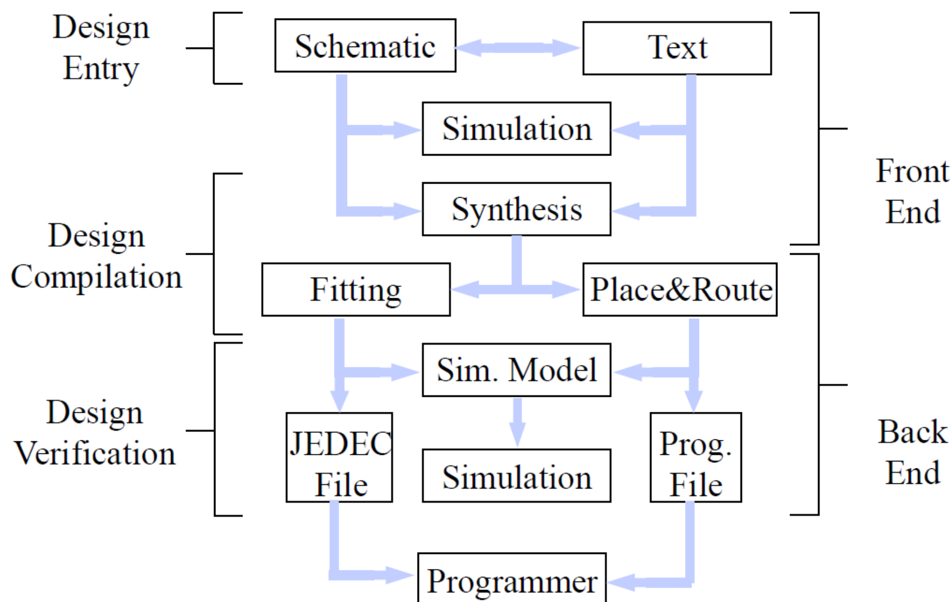


Figure 5.50: VHDL design flow diagram.[63]

As discussed previously, VHDL designs are descriptions, also referred to as "design entities", consisting of an *ENTITY* declaration that describes the design's I/O and an *ARCHITECTURE* body that describes the content of the design, e.g., a two input AND function can be expressed in VHDL as

```
ENTITY and2 IS PORT (
       a,b : IN std_logic;
          f:OUT std_logic);
END and2;
ARCHITECTURE behavioral OFand2IS
BEGIN
     f <= a ANDb;
END behavioral;
```

The ENTITY declaration is formally defined as

```
ENTITY entity_name IS PORT (
   -- optional generics
   name : mode type
   ...
);
END entity_name;
```

where *entity_name* is an arbitrary name, *generics* are used for defining parameterized components, *name* is the signal/port identifier[64], *mode* describes the direction of data flow and *type* defines the set of values a port name may be assigned. PORTS are points of communications, often associated with a device's pins, that are a special class of SIGNAL with an associated *name, mode* and *type.*

[64]This can be a separate list for ports for ports of identical modes and types.

MODE represents the direction of data flow and can be

- IN - data enters the entity but does not exit from it,
- OUT - data leaves the entity but does not enter and is not used internally,
- INOUT - Data goes in and out of the entity, i.e., it is bidirectional,

or,

- BUFFER - data exits the entity and is also fed-back internally.

5.14.2 VHDL Abstraction Levels

VHDL allows the designer to approach a design at varying levels of abstraction, viz., at the algorithm level which is merely a set of instructions to be carried out without regard for the clock, except perhaps loosely in terms of the ordering of execution of instructions, or delay issues. Alternatively a register level approach can be employed that is referred to as register transfer level (RTL). At this level of abstraction, the description includes clock dependence which gates all operations. However, propagation and the various forms of temporal delay are not supported. And finally, at the lowest level of abstraction, the description is expressed in terms of a network of registers and gates that are instantiated from standard libraries.

VHDL has available five very fundamental constructs:

1. Entity declarations that specify NAME and PORTS.[65]
2. Architecture bodies that model the circuits within an entity Configuration declarations that define which architecture is to be used with which entity.[66]
3. Package[67] declaration which is similar in function to that of a header file in a C program.
4. Package bodies that are similar to implementation files in C programs.

In addition to supporting multiple architectures within a given entity, VHDL allows the designer to determine which architecture is to be employed during the synthesis phase. The order of these constructs within a VHDL file are entity, architecture and configuration. The IEEE has defined certain standard VHDL libraries, e.g, IEEE 1164 which establishes both standard signals and data types.

Consider the case of a four input, single output logic function, an entity description can be of the form:

```
library IEEE;
use IEEE>std_logic_1164.all;
entity LogicFunction is
        port (
        a: in std_logic;
        b: in std_logic;
        c: in std_logic;
        d: in std_logic;
```

[65] Entities are analogous to the classic "Black Box" for which the internal functionality is hidden but the input and output ports, that is the interfaces, are specified. The entity is the functional equivalent of a software "wrapper'.'

[66] The architecture contains a detailed description of the entity's internal functionality/behavior.

[67] Packages are libraries of procedures, functions, overloaded operators, type declarations and components that consist of a BODY section and a declarative section. The constituents of a package can be used by more than entity in a design.

```
e: out std\_logic;
);
end entity LogicFunction;
```

The architecture body defines the internal functionality of the entity, i.e., the circuitry, based on one of the following four modalities:

1. Behavioral modeling in the form of a set of sequential assignment statements referred to as a *process*.
2. Dataflow modeling expressed in terms of a set of "concurrent" signal assignment statements.
3. Structural modeling in terms of a set of interconnected components.

or, as

4. Some permutation of the behavioral, dataflow and/or structural modeling.[68]

As an example, consider a single bit full adder as shown in Figure 5.51. The VHDL description for this circuit is

```
entity full_add_1 is
     port(
          a1: in bit; addend in
          a2: in bit; addend
          c1: in bit; carry in
          sum: out bit; sum
         c2: out bit); carry out
         end full_add_1
```

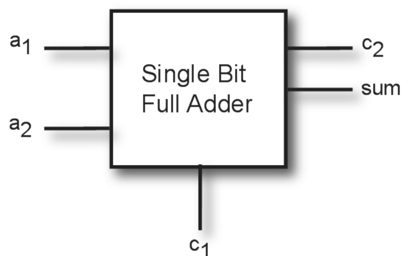


Figure 5.51: A single bit full adder

where port is defined in terms such as input, output or bidirectional. Thus a port is defined in terms of the signal associated with the port, its direction and type. The signal type can be either the single bits, 0 or 1, or in the form of a bit vector representing an array of bits. User-defined data types are also supported, such as bytes or mnemonics.

The previous example of a single bit adder can be extended as in the case of an 8-bit adder represented by the following code fragment:

```
entity: full_add_8 is
     port(
```

[68]Such combinations are often referred to as "mixed models".

```
        a1: in bit_vector(7 downto 0);
        a2: in bit_vector(7 downto 0);
        c1: in bit; carry
        sum: out bit_vector(7 downto 0); sum
        c2: out bit);
    end full_add_8;
```

It is also possible, using VHDL, to describe the behavior of a module at a higher level of abstraction for a single-bit adder, as illustrated by the code fragment:

```
          architecture dataflow of full_add_1 is
begin
   sum <= a1 xor a2 xor c1 after 3 ns;
   c2 <= (a1 and a2) or (a1 and c1) or (a2 and c1) after 3 ns;
end;
```

Identifiers are not case sensitive and must not contain any keywords. Underscores may be used in identifiers, but they must not occur at the beginning, or end, of an identifier and two or more underscores cannot occur in succession[69]. *Extended identifiers* are formally defined as:

```
extended_identifier ::= \ graphic_character {graphic_character}
```

An extended identifier is case sensitive and may contain spaces, consecutive underscores and/or keywords. Supported delimiters[70] are shown in Table 5.18. The supported character set for

Table 5.18: Delimiters Supported by VHDL

Single Char Delimiters	Description	Double Char Delimiters	Description
#	Literal Base	?>	Conditional
‘	Single Quote	?<	Conditional
“	Double Quote	**	Exponentiation
&	Concatenation	/=	Inequality
(	Left Parenthesis	??	Conditional
)	Right Parenthesis	?=	Conditional
+	“Addition” or “Positive”	?>=	Conditional
-	“Substraction” or “Minus”	?<=	Conditional
*	Multiplication	?/=	Conditional
:	Data :Type Separator	<>	Box
;	Instruction Terminator	>=	Greater Than Or Equal
/	Division	<=	Signal Assignment
?	Conditional	:=	Variable Assigment
|	OR Operator	=>	““Gets” or “Then”

VHDL’93 consists of 256 characters consisting of uppercase letters, lowercase letters, digits, and a collection of non-alphanumeric characters.

[69]The 1993 standard for VHDL (VHDL’93) permits the use of identifiers that are case sensitive, begin or end with a backslash, consist of graphical characters in any order and arbitrary length.

[70]Delimiters are defined as separators which have predefined meanings.

Character strings are delineated by double quotes, e.g.,

"This is a string"

becomes

""This is quoted a string""

Bit strings are expressed as arrays of type bit, e.g.,

literal_bit_string ::= base-specifier "bit_value"

5.14.3 VHDL Literals

VHDL supports five types of literals: bit strings[71], enumeration, numerical, strings and NULL. Bit string literals begin and end with ", may include underscores, e.g., B"0101_0101", and are treated as one-dimensional arrays that comply with VHDL'93 256 character specifications. Supported bases include binary(B), octal(O) and hexadecimal(X). While the bit string may contain underscores, the length of the string does not include the underscores. Enumeration literals may be bit, or character.

Numerical literals can contain underscores, the letters "E", or "e", to denote the inclusion of an exponent and "#" to define a base within the range of 2-16 inclusive. Physical types must have a space between the numerical value and the physical type's unit of measure. The NULL literal's use is restricted to pointers in cases for which the pointer is "empty". The "based" literals are formally defined as

```
based_literal::= base#based_integer{based_integer}#{exponent}
```

and

```
based_integer::=extd_digit{[underlined]extd_digit}
```

where

```
extd_digit::=digit|letters_A-F
```

and the base and exponent must be expressed in decimal form. Numeric literals default to decimal, may contain underscores to enhance readability but not spaces and should not have a base point or negative exponents. The use of scientific notation is restricted to integer exponents

Examples of VHDL Literals:

a) VHDL Bit String Literals:

B"10101010" -- decimal 170
B"1010_1010" -- decimal 170
O"252" -- decimal 170
X"AA" -- decimal 170

b) VHDL Numeric Literals:

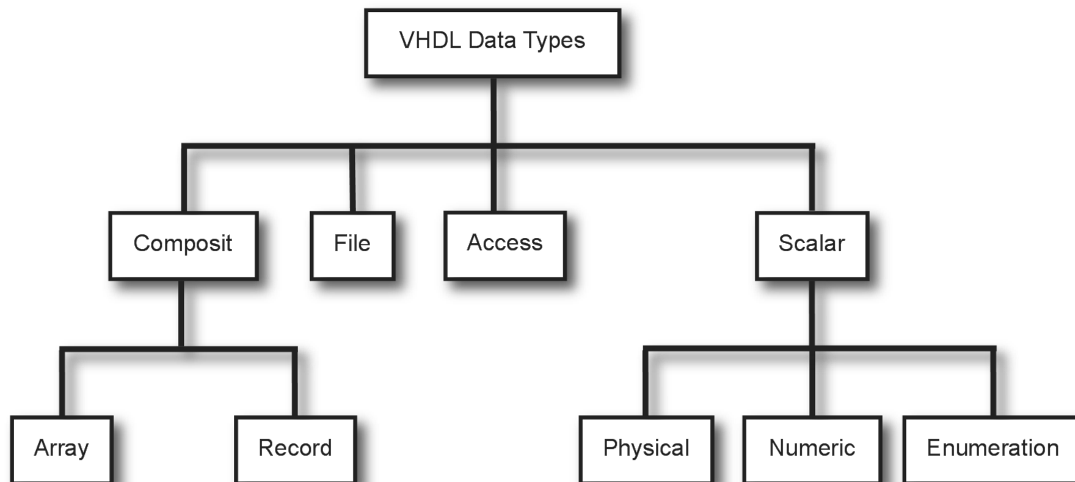


Figure 5.52: VHDL Data Types

5.14.4 VHDL Data Types

Each of VHDL's data objects have associated data types, cf. Figure 5.52, that define the allowable set of values for the data type. VHDL is a *strongly-typed* language[72], i.e., each data object is a predefined type. This constraint means that a data object of one type can not be assigned to an object of another data type.

VHDL supports the following data types:

1. Integers - allowable values -2147483647 to 214748347
2. Floating point (Real) - allowable values -1.0 E 38 to 1.0 E 38. Precision is a minimum of six decimals.
3. Physical - a numerical type that represents physical quantities such as time, mass, length, voltage, resistance, etc. The base unit must be specified in the declaration, e.g.,

```
type resistance is range 0 to 1E8
    units
      ohms;
      kohms=1000 ohms;
      Mohms=1E6 ohms;
    end units;
```

4. Arrays - no limit on array dimensions, can be indexed by any discrete type, logical and shift operations only applicable to arrays with bit or elements.
5. Signals - within a block or architecture signals are global. Signals are assigned values through the use of '<=' and receive default values through the use of ':=' Signals associated with a port must have a direction, but not within a block or architecture.

[71]Bit strings are frequently used to in initialize registers.

[72]The phrase strongly-typed is somewhat ambiguous but implies, inter alia, that the type of each variable must be declared prior to use and includes strict rules with respect to any variable manipulations. One advantage of such languages is that their respective compilers are able to catch many bugs prior to runtime. C++, C# and Java are regarded as strongly-typed whilst C is regarded as weakly or loosely-typed. It is perhaps more accurate to state that the former are more strongly typed languages than the latter.

6. Signal attributes[73] -
   signal_name'event (returns Boolean TRUE for signal event occurrence, else returns Boolean FALSE)
   signal_name'active (returns Boolean TRUE if a transaction on a signal occurs, else returns Boolean FALSE)
   signal_name'transaction (returns a signal of type bit that toggles for each subsequent transaction on the signal)
   signal_name'last_event (returns Boolean TRUE if a signal event occurs else returns Boolean FALSE)
   signal_name'last_active (returns Boolean TRUE if a signal event occurs else returns Boolean FALSE)
   signal_name'delayed(T) (returns Boolean TRUE if a signal event occurs else returns Boolean FALSE)
   signal_name'stable[T] (returns Boolean TRUE if no event on signal has occurred during time T, else returns Boolean FALSE. Default value of T is zero.)
   signal_name'quiet[T] (returns Boolean TRUE if no transaction on signal has occurred during time T, else returns Boolean FALSE. Default value of T is zero.)
7. Scalar attributes -
   scalar_type'left (returns the first or left-most value)
   scalar_type'left *(returns the last or or right-most value)*
   scalar_type'low (returns the lowest value)
   scalar_type'high (returns highest value)
   scalar_type'ascending (returns TRUE if T is ascending, else FALSE)
   scalar-type'value(s) (returns value T represented by the string value *s*)

and,

8. Array attributes:
   MATRIX'left(N) (left-most element index)[74]
   MATRIX'right(N) (right-most element index)
   MATRIX'high'(N) (upper bound)
   MATRIX'low(N) (lower bound)
   MATRIX'length(N) (number of elements)
   MATRIX'range(N) (range)
   MATRIX'reverse_range(N) (reverse range)
   MATRIX'ascending(N) (TRUE if index is an ascending range, else FALSE)

In VHDL, enumeration, numeric and physical data types are scalar. Numeric data types can be either real or integer [-2147482647, 2147482647]. Physical data types are scalar numeric values associated with a system of units and/or physical measurements. Time is supported as a predefined physical type but other physical types must be defined by the user. Time values may range from 0 to 1E20 with units in femtoseconds. Arrays, consisting of multiple elements of the same type are supported by VHDL with strings, bit_vectors (Bit_vector) and standard logic vectors (Std_logic_vector) being predefined arrays.

5.14.5 Pre-Defined Data Types and Subtypes

VHDL supports a number of predefined data types, viz, integers, reals, time (fs), bit (0,1), (true, false), bit_vector (an unconstrained array of bits), character (128 chars in VHDL'87 and 256 in

[73] Attributes are functions that return a value type or range of a data type.
[74] N is optional for any array attribute for which the matrix is a one-dimensional array.

VHDL'93), severity_level (note, warning, error, failure), file_open_kind (read_mode, write_mode, append_mode)file_open_status (open_ok, status_error, name_error, mode_error), string (an unconstrained array of characters). The predefined subtypes are natural (0-2147483647), positive (1-2147483647) and delay_length 90 fs - 2147483647).

The precedence for:

- integers, reals and time is abs, **, *, /, mod, rem, +(sign), -(sign), + (addition), - (subtraction), =, /=, <, <=, >, and >=,
- bit_vector(s) is NOT, &, sll, srl, sla, sra, rol, ror, =, /=, <, <=, >, AND, NAND, OR, NOR, XOR, and XNOR. The precedence for bit and is NOT, =, /=, <, <=, >, >=, AND, NAND, OR, NOR, XOR, and XNOR,
- natural and positive is the same as that for integers,

and

- delay_length is the same as that for time.

5.14.6 Operator Overloading

VHDL allows the user to assign new definitions to existing operators such as +, -, *, NAND, etc. when creating user-defined data types.[75] This object-oriented approach relies on VHDL determining what the appropriate operator action is for a given data type (argument). Objects can be overloaded by overloading, operators, parameters or subprogram names. Overloaded functions can have the same name but a different number of arguments or different argument types. In such cases VHDL uses the number, or type, of arguments to determine the appropriate action(s). Function names can also be in the form of an operator, so that a function can be called by symbols such as +, -, >, <, etc. For example the + operator could be "overloaded" to support vector addition, addition of two strings, etc.

5.14.7 VHDL Data Objects

VHDL data objects include variables, signals and the associated signal attributes. Variables must be declared before they can be used and only with the context of a subprogram or process. Variable declarations must also include a specification of the data type. Initializing variables at the time of declaration is optional but if left unspecified, then the default value is the leftmost element of the declared data type declared.

Signals are declared in a similar manner to that of variables and are subject to the to following conditions and constraints:

- They are declared as *either intermediate nodes* (architecture) or *ports* (entity)
- Entities can have port access.
- Assignment of values to input ports is supported by VHDL.
- "Signals" refer to nodes which may have voltage dependencies which are a function of time.
- Signal assignments employ the "<=" delimiter.
- A *transaction* is the scheduling of a value to a signal.
- An *"event"* is said to have occurred when a signal's value changes.
- Signals that are assigned a value without the specification of a delay will, during simulation, only change value after the simulation's sub-interval has transpired.

[75] The assignment of a new function to an existing operator is referred to "operator overloading".

- Signals have the following attributes:
  1. *X'active* returns TRUE if a transaction has occurred during the current simulation time. otherwise it returns FALSE.
  2. *X'quiet(n)* returns TRUE if no transaction has occurred during the previous 'n' seconds.
  3. *X'event* returns TRUE if the value of X changed during the current simulation time.
  4. *X'stable(n)* returns TRUE if X did not experience an event during the past 'n' seconds.
  5. *X'delayed* delays the signal X for n seconds.
  6. *X'last_active* returns the time that has elapsed since the last transaction.
  7. *X'last_event* returns the time that has elapsed since the last event.
  8. *X'last_value* returns the previous value of X.

5.14.8 VHDL Operators

VHDL expressions consist of operators and so-called primaries.[76] Logical operators such as AND, *NAND*, *OR*, *ROR*, NOR and *NOT* can operate on arrays as well as can be applied to one dimensional arrays, s or values of type bit. Operator types include:

1. Logical operators[77]: *AND, NAND, XOR, OR, NOR, XNOR* and *NOT*
2. Unary sign operators: plus (+) and minus(-)
3. Addition operators: plus (+),
4. Addition operators: Plus (+), minus (-) and concatenation[78] (&),
5. Shift operators[79]: Shift right logical[80] (srl), shift left logical (sll), shift left arithmetic[81] (sla), shift right arithmetic (sra), rotate left (rol) and rotate right (ror,)
6. Multiplication operators: multiply (*), divide [0, modulus[82] (mod) and remainder (rem)],
7. Exponentiation (**) is subject to the constraint that the lefthand operand must be an integer, or floating point, value and the righthand operand must be integer only.
8. Absolute value (abs) This operator can be applied to any numeric type within an expression.
9. NOT - the inversion operator.

The order of precedence for these operators from highest precedence to the lowest is: exponentiation, absolute value and not (inversion) followed by multiplication, addition, shifting, relational and finally logical operators. If two operators of equal precedence are encountered then the lefthand operator is evaluated followed by the righthand operator. All of these rules are applied beginning with the most deeply nested parentheses in the expression.

Arithmetic operations such as division and multiplication can be applied to floating point and integer values. If the righthand operand is negative, the lefthand operand must be a floating point value floating point numbers or physical types. While the exponential operator can be applied to either floating point or integer values, the righthanded operand must be an integer. Relational operators such as =, /=, >, >=, < and <= produce results but both the righthand

[76]The term *primaries* refers to function calls, object names, literals and parenthetical expressions.
[77]These operators are not subject to any precedence order so liberal applications of parentheses is recommended.
[78]The concatenation operator combines the bits on either side of the concatenation operator.
[79]Shift operators have two operands. The lefthand operand is the bit_vector to be shifted, or rotated, and the righthand operand is an integer value representing to number of shifts or rotates. A negative value for the latter results in the inverse operation being invoked.
[80]The *fill value* for sll and srl is '0'.
[81]The fill value for sla is the righthand bit and the lefthand bit for sra.
[82]This mod and rem operators are only applicable to integer types

and lefthand operands operand must be the same type. Two values are treated as equal provided that the corresponding elements of each are equal. The concatenation operator, typically used to join strings, can be applied to two one-dimensional arrays with as little as one element each. Single elements can be concatenated with multi-element arrays.

5.14.9 Conditional Statements

VHDL supports both *if-then-else* and *case* statements. Execution is subject to user defined conditions in the form of expressions which evaluate to values. If such an expression evaluates as true then the corresponding statements are executed otherwise the else statements are executed.

If statements are of the general form[83]

```
if <condition> then
     statements
         ...
[
elsif <condition> then
     statements
         ...
else
     statements
         ...
]
end if;
```

and case statements are of the form

```
case <expression> is

when <choice(s)> =>

<expression>;
      ...

when ...

[when others => ... ]

end case;
```

Case statements allow execution to depend on the value of a selection statement. Case statements must include all possible values of the expression to be evaluated, however, values that are not to be treated by the case statement can be included as "OTHERS' in conjunction with the reserved word "NULL" resulting in no action with for those values. Supported expression types include integer, enumerated and one dimensional character arrays.

5.14.10 FOR, WHILE, LOOP, END and EXIT

FOR and WHILE loops are supported in VHDL together with EXIT which is used to leave a loop[84] and END LOOP to end a loop. LOOP can be used to repeat a loop indefinitely, e.g.,

[83] Note that "conditions" are restricted to type , "elsif" contains one "e" and "end if" is two words.
[84] Also referred to as "jumping out of a loop".

```
loop
   some_activity;
end loop;
```

The formal syntax for both WHILE and FOR is

```
loop statement ::=
     [loop label :] [ while -expression | for ]
loop
    { statements }
end loop
```

The *while* reserved word evaluates a test condition prior to each iteration and if the expression evaluates as true the next iteration in invoked, otherwise the loop terminates. The *for* iteration loops for a predefined number of iterations. and a loop parameter keeps track of the number of iterations that have occurred. A *next* statement can be used to terminate the current iteration and an *exit* statement terminates the loop thereby passing control to the next statement to be executed. The *null* statement is typically used to indicate no action is to take place.

The *assert* statement provides exception handling and is of the following formal form:

```
assertion_statement::=[label : ]assertion;
```

where

```
assertion::=
        Assert condition
              [Report expression]
              [Severity expression];
```

If the status is not consistent with this condition and the *report* clause is present, a message, e.g., "assertion violation" occurs. The *severity* clause assigns a severity level, viz., *note*, *warning*, *error* or *failure*. If the severity clause is not present, then the security level defaults to *error*. The assert statement can be used to halt the execution of a simulation.

5.14.11 Object Declarations

Three types of objects are supported in VHDL, viz., variables, constants and signals. Constants are initialized with a specific value which may not be modified thereafter[85] unlike variables whose values may be modified after initialization. A *deferred constant* declaration occurs in a corresponding package declaration, but is assigned a value in the *package body*. Non-shared variables are treated as local variables in subprograms and processes and shared variables are treated as global variables. Variables are assigned values by use of " := ". If an object is merely declared without initialization, its value defaults to that of the first value occurring in the package body.

5.14.12 FSMs and VHDL

Finite state machines are often described in VHDL by using processes *sensitive* only to clock and asynchronous resets for state transitions. Outputs, in such cases, are expressed as concurrent statements external to the process. State machines can be viewed simply as black boxes and

[85] In effect, constants in VHDL are read-only.

therefore a behaviorial model utilizing an entity/architecture pair is sufficient to describe it.[86]The internal states can be defined in terms of enumerated types.

A combinatorial process can be used to provide the next-state conditioning logic, a synchronous process can be used to provide the current state variables and a third process can provide the output logic. Each of the processes operates concurrently in VHDL, and therefore the combination would function as a FSM. The next-state conditioning logic determines the next state as a function of the current state and the inputs. In VHDL the selection of the next state can be managed by the use of a *case* statement.[87] The synchronous process can handle register the current state condition and to reset the state machine to a predefined state. The output logic can be implemented in VHDL as a set of *if-then-else* statements.[88]

5.15 Verilog

Verilog is a hardware description language, similar in some respects to VHDL, that is used to model electronic circuits primarily at the register level.[89] Originally introduced in 1984, it provided designers with a description language that for most designs was much easier to learn and use than VHDL. With fewer data types than VHDL, limited casting allowed, no user-defined types supported and relying on primitive types, the language could employ a fast, memory efficient, simpler compiler than that required for VHDL. While originally a proprietary language, in 1990, Open Verilog International (OVI) was formed and a joint effort was undertaken to create a Verilog standard reference manual that ultimately led to the establishment of an IEEE standard for the language.

The Verilog reserved word list (aka keywords list) is shown in Table 5.19. Both line and block comments are supported in Verilog with two forward slashes representing the start of the comment which is assumed to extend to the end of that line. Block comments begin with /* and end with */ and cannot be nested.

Identifiers that begin with a backslash (\) and end with a whitespace, e.g., newline, space or tab are treated as "escaped" identifiers. However the leading backslash and ending whitespace are treated as part of the identifier.

In Verilog a bit can take of one of four values: 0, 1, X or Z corresponding to logic zero, logic one, an unknown logic value[90] or high impedance (floating).

5.15.1 Constants

Constant values can be expressed as either simple decimals, i.e., as a sequence of digits employing values 0-9 or as a sized constant[91] representing a based[92] number. String constants are treated as unsigned integer constants represented by a sequence of 8-bit ASCII values, with each such value representing a particular character.

[86]The entity defines the interface and the architecture defines the internal behavior.

[87]It is important specify all the possibilities for a given case statement, even if some possibilities are not used in order to avoid exceptions.

[88]It's a good practice to include an else statement for each if statement, because VHDL signals have "implicit memory".

[89]Verilog takes into account signal timing, propagation delays and edge transitions in the description and modeling.

[90]Unknown logic values are restricted to 0, 1 or Z or a transition from one of three to another of the three allowed values. X represents either a "don't know" or "don't care" state, or both.

[91]Sized constants consist of three tokens, viz., an optional size, a single quote followed by a base formed character and a sequence of digits representing the value.

[92]For example: hexadecimal, octal, binary or decimal.

5.15.2 Data Types

Verilog supports data types belonging to one of three classes, viz.,

- *nets* - The net data type represents a physical connection between hardware blocks and can be driven by a continuous assignment statement or the output of a module or gate. However, a net data type does not store its value. A net data type can be a wire[93], tri, supply0 or supply1 type. Wire and tri data types are identical in terms of syntax and functionality and are supported to delineate between wire nets that are driven by a single gate and tri nets that are driven by multiple drivers, supply0 and supply1 are the nets representing logic 0 (ground) and logic1 (power) when modeling power supplies. Declaration of a scalar net must include the range of bits, e.g.,

```
wire [7:0] dataA; /*dataA where bit0 is the LSB and bit7 is the MSB.

wire [0:7] dataA; /*dataA where bit0 is the MSB and bit7 is the LSB.
```

  A net can be either a scalar[94] or vector, the former representing individual signals and the latter representing bus signals. The *strength* of a net is specified by drive strength[95] and charge.[96]

- *Registers* - Register data types store their values, until changed by an assignment in *always* blocks functions or tasks, and are used as variables. To declare a reg type the reserved word *reg.* is employed and is declared by using the keyword *reg.*[97] The integer register data type is used for values that are not to be treated as registers. Register variables may be declared as either scalars or vectors. Vector register declarations include a specification of the range of the bits after *reg* or *integer*. The left and righthand values in this range specify the most significant and least significant bits, respectively.

- *Parameters* - Parameter values are used in parameterized models are treated as constants during runtime and are declared as follows[98]:

```
parameter parameter_assignment {,parameter_assignments}
parameter_assignment ::= parameter_identifier=constant_expression
```

  Examples of parameter use are given by:

```
parameter lsb = 0, msb =3; // lsb and msb are parameters
reg [msb,lsb] x          ; // x is a vector with range 3:0
parameter tPD = 7        ; // parameter tPD is used to represent
                              propagation delay
```

  Parameters are treated as strings of arbitrary length unless constrained by the user, e.g.,

```
parameter unconst_param = 12 /* unconstrained
                     // (size is determined by usage) */
parameter [3:0] const_param = 12; //constrained to 4 bits
```

[93] A wire represents a 1 bit interconnection between modules.
[94] By default all nets are treated as scalars.
[95] Specified as weak0, weak1, highz0, highz1, pull0, pull1, pullup or pulldown.
[96] Specified as small, medium or large.
[97] It should be noted that *reg* is not necessarily a hardware register or flip-flop but rather simply denotes the fact that the value is retained.Uninitialized *reg* values are treated as X, i.e., undefined.
[98] If a given parameter is not intended to be modified by a higher level module, the compiler directive *define* should be used.

Table 5.19: Verilog Reserved Words

always	and	assign	automatic	begin
buf	bufifo	bufif1	case	casex
casez	cell	cmos	config	deassign
default	defparam	design	disable	edge
else	end	endcase	endconfig	endfunction
endgenerate	endmodule	endprimitive	endspecify	endtable
endtask	event	for	force	forever
fork	function	generate	genvar	highz0
highz1	if	ifnone	incdir	include
initial	inout	input	instance	integer
join	large	liblist	library	localparam
nmos	no	noshowcancelled	not	notif0
notif1	or	output	parameter	pmos
posedge	primitive	pull0	pull1	pulldown
pullup	pulsestyle_oneevent	pulsestyle_ondetect	nrcmos	real
realtime	reg	release	repeat	rnmos
rpmos	rtran	rtranif0	rtranif1	scalared
showcancelled	signed	small	specify	specparameter
strong0	strong1	supply0	supply1	table
task	time	tran	tranif0	tranif1
tri	tri0	tri1	triand	trior
trireg	unsigned	use	vectored	wait
wand	weak0	weak1	while	wire
wor	xnor	xor		

5.15.3 Modules

A Verilog module encapsulates the description of a design that can be either:

1. a behavioral (algorithmic) description that defines the behavior of a circuit in abstract, high level algorithms, or expressed in terms of low level boolean equations,

or,

2. a structural description that defines the structure of the circuit in terms of components and resembles a net-list that describes a schematic equivalent of the design and supports concurrency.[99]

5.15.3.1 Module Syntax

A Verilog design consists of one or more modules[100] interconnected by ports. Each port has an associated name and mode, viz., *input, output* and *inout*. Module definitions cannot be nested. A module is defined using the following syntax:

```
module <name> (interface_list) ;{ module_item }
endmodule
interface_list ::= port_reference
| {port\_reference {, port_reference}}
port\_reference ::= port_identifier
| port_identifier [ constant_expression ]
```

[99] Structural descriptions contain a hierarchy in which the components are defined at different levels and the logic is defined in terms of gate primitives.

[100] Warp treats the keywords *macromodule* and *module* as synonyms.

```
| port_identifier [ msb_constant_expression : lsb_constant_expression ]
module_item ::= module_item_declaration
| continuous_assignment
| gate_instantiation
| module_instantiation
| always_statement
module_item_declaration ::= parameter\_declaration
| input_declaration
| output_declaration
| inout_declaration
| net_declaration
| reg_declaration
| integer_declaration
| task_declaration
| function\_declaration
```

Example:

```
// a module definition for a d flip-flop
module
my\_dff (clk, d, q);
input clk, d;
output q;
wire clk, d;
reg q ;
always @(posedge clk)
begin
q = d ;
end
endmodule
// a module definition for module
my\_dff (clk, d, q);
input clk, d;
output q;
wire clk, d;
reg q ;
        always @(posedge clk)
                begin
                    q = d ;
                end
endmodule
```

5.15.4 Operators

Verilog supports a variety of operations including *arithmetic*, *bit-wise*, *concatenation, conditional*, *equality*, *logical*, *reduction*, *relational*, *replication,* and *shift*:

- *arithmetic operators* - Binary operators for addition, subtraction, multiplication divisions and modulus and unary operations for specifying the sign of a value, i.e., plus or minus. Integer division is supported but truncates the fractional part. Register data types are treated as unsigned values and negative values are expressed in a twos-complement format. The modulus operator assigns the result the same sign as the that of the first operand.

- *bit-wise operators* - perform bit-wise operations on the respective bits of the two operands. If the two operands are of different bit lengths the shorter value, bitwise, will be padded with zeros sufficient to match the bit length of its counterpart. Supported bit-wise operations include and (&), inclusive or (|), exclusive or (ˆ) and exclusive nor (ˆ˜or ˜ˆ), i.e., equivalence.
- *concatenation operator* - uses braces to encapsulate the values to be concatenated. Each such value is delimited by a comma, e.g., {a, eb[4:0], c, 5'b11011}.
- *conditional operators* - have the following syntactic form

```
condition ? expression1 ; expression2
```

If *condition* evaluates as false, i.e., zero, then *expression2* is evaluated, otherwise *expression1* is evaluated. If *condition* evaluates as either z or v, then both *expression1* and *expression2* are evaluated and the resulting value is determined by a bit by bit examination based on Figure 5.53. If *expression1*, or *expression2*, are of type real the value of the whole expression is zero. If *expression1* and *expression2* are of different lengths, then the length of the entire expression is assigned the length of the longer expression and trailing zeros are added to the shorter expression as required.

	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

Figure 5.53: Value of conditional expressions containing x,z,1 and/or 0.

- *equality operators* - there are two types of equality operators, viz., case and logical. For case equality the operators are a===b (a is equal to be for 0,1,z and x) and a!==b (a is not equal to be for 0,1,z and x). For logical equality the operators are a==b and a!=b and in some cases the result may be undefined. These operators are compared bit-by-bit with zeros being added to make the two operands the same length. If either operand contains a z or an x, then the result is x for $a == b$ and $a! = b$. If either operand contains an x or a z, then a===b and a!==b can only be true if the respective bits in a and b have the same values of x and z.
- *logical operators* - are logical *negation* (|), logical *or* (||) and logical *and* (&&). Logical *negation* and logical *and* are evaluated from left to right.
- *reduction operators* - are the unary operators *and*, *or*, *xor*, *nand*, *nor* and *xnor* that are bit-wise operations on a single operand and produce a single-bit result, e.g.,

  &(4'b0101) = 0 & 1 & 0 & 1 = 1'b0.
- *relational operators* - are the less than, greater than, less than or equal to and greater than or equal to operators that produce a scalar value of zero is the relation is false, 1 if the relation is true and x if any of the operands contains unknown x bits. If any operand is x or z, the result is treated as false.
- *replication operators* - replicates a group of bits n times, e.g., {1, 1,{3{1,0}}}= 11101010.

- *shift operators* - performs right- or left-shifts on the righthand operand where the number of shifts, right (>>) or left (<<), is determined by the value of the righthand operand.[101]

The following is an illustrative example of a Verilog program designed to compute square root:

```
module sqrt32(clk, rdy, reset, x, .y(acc));
input  clk;
output rdy;
input  reset;
input [31:0] x;
output [15:0] acc;
// acc = accumulated result, and acc2 = accumulated acc^2
reg [15:0] acc;
reg [31:0] acc2;
// Track bit being worked on.
reg [4:0]  bitl;
wire [15:0] bit = 1 << bitl;
wire [31:0] bit2 = 1 << (bitl << 1);
// Output ready when bitl counter underflows.
wire rdy = bitl[4];
// guess h=next values for acc. guess2=square of that guess h.
// guess2 = (acc + bit) * (acc + bit)
//        = (acc * acc) + 2*acc*bit + bit*bit
//        = acc2 + 2*acc*bit + bit2
//        = acc2 + 2 * (acc<<bitl) + bit
// Note: bit and bit2 have only a single bit in them.
wire [15:0] guess  = acc | bit;
wire [31:0] guess2 = acc2 + bit2 + ((acc << bitl) << 1);
(* ivl_synthesis_on *)
always @(posedge clk or posedge reset)
if (reset) begin
        acc = 0;
        acc2 = 0;
        bitl = 15;
end else begin
                if (guess2 <= x) begin
                acc  <= guess;
                acc2 <= guess2;
        end
        bitl <= bitl - 5'd1;
end
endmodule
```

5.15.5 Blocking versus Nonblocking Assignments

In Verilog/Warp a *blocking statement*, which is part of a sequential block, must be executed prior to the execution of those statements that follow it. In the case of nonblocking statements, assignments occur without blocking the procedural flow. Blocking assignments employ the symbol "=" and nonblocking statements employ the symbol "<=" for assignment. Nonblocking statements allow events to be scheduled for a later time.

[101] Vacated bits are replaced by zeros.

5.15.6 *wire* versus *reg* Elements

wire elements are used in Verilog applications to connect the input and output ports of a module instantiation with other elements within a design. However, unlike their counterpart, *reg*, they are not able to store values and must be driven. In effect wires serve as "state-less" connection mechanisms. *wire* elements are only used in cases for which the model is based on combinatorial logic. *reg* elements perform a function similar to that of wires but have the ability to store values in a manner analogous to that of registers. These elements are used in both sequential and combinatorial logic models. While reg cannot be used on the lefthand side of an assign statement, it can be used to create registers in conjunction with always@(posedge clock) statements/blocks. *reg* can also be used on the lefthand side of always@block = or <= symbol. It can also be used as the input to a module, or within a module declaration, but not to connect to the output port of a module.

5.15.7 *always* and *initial* Blocks

In modeling combinatorial and sequential elements the *initial* and *always* blocks play important roles. *initial* blocks[102] are procedural blocks consisting of sequential statements that are executed only once, typically at the start of execution of a simulation, whereas an *always* block is always available for as long as the program is executing. An *always* statement contains a *sensitivity list*[103] that determines when the block of code associated with the *always* block is to be executed. Any change in the signals contained in the sensitivity list will cause the *always* block to be executed.

The standard format for an always statement is defined as:

```
always@(event_expression_1 [or event_expression_2]{or event expression_3}])
```

*event_expression*s can contain timing controls which are either *posedge* or *negedge* for positive or negative-edge triggering, respectively. If sequential triggers are employed in the sensitivity list, sequential logic is synthesized. Asynchronous, or synchronous, triggers may be used in the *sensitivity list*, but not both.

```
// Always block with asynchronous triggers:
        always @(x or y)
begin
        ...
end
/* Always block which realizes sequential logic with
rising edge of a clock: */
always @(posedge clock)
begin
        ...
end
/* Always block which realizes a sequential logic with
falling edge of clock and an asynchronous preload */
always @(negedge clock or posedge load)
begin
        ...
end
```

[102] Warp ignores *initial* constructs.
[103] Sometimes referred to as the *sensitive list*.

The following are equivalent syntactically:

```
always@(signal_1 or signal_2 or  signal_3 or signal_4)
always@(signal_1,signal_2, signal_3, signal_4)
always@(*)
always@*
```

where * refers to all of the signals within the *always* block.

5.15.8 Tri-State Synthesis

Warp does not synthesize tri-state logic. In order to include tri-state logic in a Verilog module the cy_bufoe[104] must be instantiated. The tri-state output of this module, y, must then be connected to an inout port on the Verilog module. That port can then be connected directly to a bidirectional pin on the device. The feedback signal of the cy_bufoe, yfb, can be used to implement a fully bidirectional interface, or can be left floating to implement just a tri-state output.

```
module ex\_tri\_state (out1, en, in1);
        inout out1;
        input en;
        input in1;
        cy\_bufoe buf\_bidi (
                .x(in1), // (input) Value to send out
                .oe(en), // (input) Output Enable
                .y(out1), // (inout) Connect to the bidirectional pin
                .yfb()); // (ouptut) Value on the pin brought back in
endmodule
```

5.15.9 Latch Synthesis

Warp synthesizes a latch whenever a variable inside an always block with asynchronous trigger, has to hold its previous value. The following code fragment synthesizes a latch.

```
// example: latch synthesis with if statement
always @ (signal1 or signal2)
begin
        if( signal1 )
                begin
                        out\_sig = signal2 ;
                end
end
```

5.15.10 Register Synthesis

A register is typically a set of flip-flops that share a common clock input and is used to store a group of bits. The register is updated when the next clock edge occurs. Most registers employ both a reset and load input controls. In the case of a shift register, the flip-flops are connected in a chain in which the output of one flip-flop becomes the input to the next flip-flop in the chain.

[104] cy_bufoe is a tri-state, non-inverting buffer with an active high output enable input.

This interconnection scheme allows data to be *shifted* to the next flip-flop each time a clock edge occurs. Shift registers can be serial-in-serial-out (SISO), parallel-in-parallel-out (PIPO), serial-in-parallel-out (SIPO) or parallel-in-serial-out (PISO). Thus in order to synthesize registers it is necessary to be able to synthesize flip-flops.

5.15.10.1 Edge-Sensitive Flip-Flop Synthesis

Warp uses the following templates to synthesize synchronous flip-flops. The template for the positive edge sensitive flip-flop is:

```
always @ (posedge clock\_signal)
synchronous\_signal\_assignments
```

and the template for the negative edge sensitive flip flop is:

```
always @ (negedge clock\_signal)
synchronous\_signal\_assignments
```

5.15.10.2 Asynchronous Flip-Flop Synthesis

Warp uses the following format to synthesize asynchronous flip-flops with reset, or preset.

```
always @ (edge\_of clock\_signal or
        edge\_of preset\_signal or
        edge\_of reset\_signal)
if (reset\_signal)
        reset\_signal\_assignments
else if (preset\_signal)
        preset\_signal\_assignments
else
        synchronous\_signal\_assignments
```

The *posedge* construct is used to specify an active high condition and the *negedge* construct to specify active low condition. The variables in the sensitivity list can appear in any order. Subsequent reset, or preset, conditions can appear in the else-if statements. The last else block represents the synchronous logic. The polarity of the *reset/preset* signal condition used in the sensitivity list and the polarity of the *reset/preset* condition in the *if/else-if* statements should be the same.

Example: A *posedge* reset_signal condition in the sensitivity list is required when the reset condition is one of the following forms:

```
if( reset\_signal)
if( reset\_signal == constant\_one\_expression)
```

A *negedge* reset_signal condition in the sensitivity list is required when the reset condition is one of the following forms:

```
if( !reset\_signal)
if( ~reset\_signal)
if( reset\_signal == constant\_zero\_expression)
```

Warp generates an error if the polarity restriction mentioned above is violated. Warp allows more than two asynchronous if/else-if statements before the synchronous else statement as shown in the following example.

```
// An example of two different preset signals:
module asynch\_rpp(in1, clk, reset, preset, preset2, out1);
input in1, clk, reset, preset, preset2;
output out1;
reg out1;
always @ (posedge clk or posedge reset or posedge preset or posedge
preset2)
        if (reset)
                out1 = 1b0;
        else if (preset)
                out1 = 1b1;
        else if (preset2)
                out1 = 1b1;
        else
                out1 = in1;
endmodule
```

The *posedge* and *negedge* keywords are use to specify active high and low conditions, respectively. Variables in the sensitivity list can occur in any order. The polarity of reset/preset signal conditions in a sensitivity list and the polarity of the reset/preset conditions in corresponding if/else-if statements must be the same.

A *posedge* reset_signal condition in the sensitivity list is required when the reset condition is one of the following forms:

```
if(reset\_signal)
if(reset\_signal == constant_one_expression)
```

A *negedge* reset_signal condition in the sensitivity list is required when the reset condition is one of the following forms:

```
if(!reset\_signal)
if(~reset\_signal)
if(reset\_signal == constant\_zero\_expression)
```

Warp generates an error if the polarity restriction mentioned above is violated. Warp allows more than two asynchronous if/else-if statements before the synchronous else statement as shown in the following example.

```
// An example of two different preset signals:
module asynch\_rpp(in1, clk, reset, preset, preset2, out1);
input in1, clk, reset, preset, preset2;
output out1;
reg out1;
```

5.15.11 Verilog Modules

Verilog modules[105] are used to encapsulate the description of a design expressed either as a behavioral or structural description. A behavioral description defines the behavior of a circuit in terms of abstract, high-level algorithms or in terms of low-level equations.

[105] A Verilog module may represent a single gate, flip–flop- or register or other much more sophisticated circuits.

A structural description defines the circuits structure in terms of components and resembles a net-list that describes a schematic equivalent of the design. Structural descriptions contain hierarchy in which components are defined at different levels.

A Verilog design consists of one or more modules[106] connected with each other by means of ports which provide a means of connecting various hardware elements. Each port has an associated name and mode (*input*, *output* and *inout*). A module is defined using the following syntax:

```
module <name>(interface\_list) ;{ module\_item }
endmodule
interface_list ::= port_reference
| {port\_reference {, port\_reference}}
port\_reference ::= port\_identifier
| port\_identifier [ constant\_expression ]
| port_identifier [ msb_constant_expression : lsb\_constant\_expression
module_item ::= module_item_declaration
| continuous_assignment
| gate_instantiation
| module_instantiation
| always_statement
module_item_declaration ::= parameter_declaration
| input_declaration
| output_declaration
| inout_declaration
| net_declaration
| reg_declaration
| integer_declaration
| task_declaration
| function_declaration
```

In Verilog, hierarchical designs are specified by instantiating one, or more, modules in a top level module not instantiated by any other module. The syntax of the module instantiation statement is as follows:

```
<module_name> [parameter_value_assignment]
<instance_name>
module_instance {, module_instance} ;
module_instance ::= instance_identifier
([list_of_module_connections])
list_of_module_connections ::= ordered_port_connection {,
ordered\_port\_connection }
| named\_port_connection {,named_port_connection }
```

One or more instantiations of the same module can also be specified in a single module instantiation statement. The four instantiation statements in the above example can be combined into one instantiation statement as follows:

```
my\_dff inst\_3(clk, d, q0),
        inst\_2(clk, q0, q1),
        inst\_1(clk, q1, q2),
        inst\_0(clk, q2, q) ;
```

[106] Module definitions cannot be nested.

A module connection describes the connection between the signals listed in the module instantiation statement and the ports in the module definition. This connection can be specified in two ways: ordered port association and named port association. In the case of ordered port association, the signals in the instantiation statement should be in the same order as the ports listed in the module definition. In the case of a named port association, the port names of instantiated modules are also included in the connection list.

```
my_dff inst_3(clk, d, q0)               ; // ordered connection list.
my_dff inst_3(.d(d), .q(q0), .clk(clk)) ; /* named association: q0 is
                                             connected to the port q of
                                             my_dff module. */
```

The port expression in the module connection list can be one of the following: a simple identifier, a bit-select of a vector declared within the module or a part-select of a vector declared within the module or some concatenation thereof.

The following describes the behavior of a counter that increments the count by 1 on the rising edge of a clock (trigger). It also contains an asynchronous reset signal that resets the counter to zero.

```
module counter (trigger, reset, count);
    parameter counter_size = 4;
    input trigger;
    input reset;
    inout [counter_size:0] count;
    reg [counter_size:0] tmp_count;
    always @(posedge reset or posedge trigger)
    begin
       if (reset == 1b 1)
             tmp_count <= {(counter_size + 1){1b 0}};
       else
             tmp_count <= count + 1;
       end
       assign count = tmp_count;
endmodule
```

5.15.12 Verilog Tasks

Tasks are sequences of declarations and statements that can be invoked repeatedly from different parts of a Verilog description. They also provide the ability to break up a large behavioral description into smaller ones for easy readability and code maintenance. A task can return zero or other values.

A *task* declaration has the following syntax:

```
task \textless task\_name\textgreater\ ;{ task\_item\_declaration}
statement\_or\_null endtask
task\_item\_declaration ::= parameter\_declaration
reg\_declaration
integer\_declaration
input\_declaration
output\_declaration
inout_declaration
```

Warp ignores any timing controls present inside a task. The order of variables in the task enable statement calling a task must be the same as the order in which the I/Os are declared inside a task definition. Only *reg* variables can receive output values from a task, i.e., wire variables cannot. Note that Datapath operator inferencing is not supported inside tasks. When datapath operators (+, -, *) are used inside tasks, at least one of the operands must be a constant or an input.

The following is an example of a module task:

```
module task_example(a,b,c,d,sum);
    output sum;
    input a,b,c,d;
    reg sum;
    always @(a or b or c or d)
    begin
        t_sum(a,b,c,d,sum);
    end
    task t_sum;
        input i1,i2,i3,i4;
        output sum ;
        begin
            sum = i1+i2+i3+i4;
        end
    endtask
    endmodule
```

5.15.13 System Tasks

Verilog supports a number of *system tasks* that support I/O and measurement functions. These tasks are all prefixed by the symbol "$" and include the following:

- $display - writes text to the screen

```
$display(<parameter_1>, <parameter_2>, <parameter_3>)
```

- $dumpfile - declare the output file name (VCD format)
- $dumpports - dump the variables (extended VCD format)
- $dumpvars - dump the variables.
- $fdisplay - print to the screen and add a newline.
- $fclose - close and release an open filehandle.
- $fopen - open a handle to a file for either a read or write.
- $fscanf - read a format-specified string from a variable.
- $fwrite - write to a file without a newline.
- $monitor - print the listed variables when any of them change value.
- $random - return a random value.
- $readmemb - read the binary file content into a memory array.
- $readmemh - read the hex file content into a memory array.
- $sscan - read a format specified string from a variable
- $swrite - print a line without the newline to a variable.
- $time - the value of the current simulation time.
- $write - write a line to the screen without a newline.

5.16 Verilog Functions

Similar to tasks, *functions* are also sequences of declarations and statements that can be invoked repeatedly from different parts of a Verilog design. As is the case with *tasks*, *functions* provide the ability to break up a large behavioral description into smaller ones for readability and maintenance. Verilog functions are formally defined as:

```
function [range_or_type] <function_name>
     function_item_declaration {function_item_declaration}
     statement endfunction
function_item_declaration ::= parameter_declaration
    | reg_declaration
    | integer_declaration
    | input_declaration
```

Unlike a *task*, a *function* returns only one value. The *function* declaration will implicitly declare an internal register which has the same type as the type specified in the *function* declaration. The return value of the function is the value of this implicit register. A *function* must have at least one *input* type argument. It can not have an *output* or *inout* type argument.

A *function* declaration can consist of the following types of declarations: *input*, *reg*, *integer*, or *parameter*. The order in which the inputs are declared should match the order in which the arguments are used in the function call. Timing controls and *nonblocking* assignment statements are not allowed inside a function definition. Datapath operator inferencing is not supported inside *functions*. When datapath operators (+, -, *) are used inside *functions*, at least one of the operands must be a constant or an input. The function inputs can not be assigned to any value, inside the function. All system task functions are ignored by Warp.

Example:

```
              module func_example(a,b,c,d,sum);
                      output[2:0] sum;
                      input a,b,c,d;
                      reg[2:0] sum;
                      always @(a or b or c or d)
                      begin
                         sum = func_sum(a,b,c,d);
                      end
                      function[2:0] func_sum;
                           input i1,i2,i3,i4;
                           begin
                              func_sum = i1+i2+i3+i4;
                           end
                endfunction
     endmodule
```

5.17 Warp™

Cypress Semiconductor supports a subset of Verilog known as Warp™[57]. However, there are a number of significant differences between Verilog and Warp, viz.,

- Warp requires that the first character in an identifier must be a letter.

- Warp renames identifiers beginning with an underscore by adding the prefix 'warp'.
- Warp does not support "escaped" identifiers.
- If an underscore is used in a constant, it is ignored by Warp.
- Warp allows parameters to appear on the righthand side of another parameter definition.
- Warp ignores the delay expressions, i.e., minimum, typical, and maximum.
- Warp treats the keywords macromodule and module as synonyms.
- Warp ignores the charge strength, drive strength and delay specified in the continuous assignment statements.
- Warp ignores all system tasks and system function identifiers.
- The following Verilog net types are not supported by Warp.
  1. tri0
  2. tri1
  3. wand tri
  4. and
  5. wor
  6. trior
  7. trireg
- Warp ignores the strengths associated with any net. Warp treats integers as 32-bit signed quantities and *reg* datatypes as unsigned quantities, by default, unless specified to be signed quantities.
- Warp does not support multiple drivers for register and integer variables.
- The time, real and realtime declarations are not supported in Warp.
- Ranges and arrays for integers are not supported by Warp. Arrays of register data types (memories) are also not supported in Warp.
- Warp does not automatically handle the size or the signed/unsigned nature of parameters.
- Warp uses the default values, if a parameter does not have a size constraint or a type (signed/unsigned/ integer/etc.) designation.
- Warp allows only *defparam* to be used to modify the parameters of immediate instances.
- Parameter values in a module can also be re-defined by using the *defparam* construct. At any level of the design, Warp allows the re-definition of parameters of the modules instantiated at that level only. More than one levels of hierarchical path names are not currently supported.
- Warp does not support the case equal operators === and !==.
- Although concatenation can be repeated using a repetition multiplier in Verilog, Warp requires that the repetition operator be a constant.
- Warp does not support range specifications in module instantiations (array of instances).
- Warp supports the following primitive gates: *and*, *nand*, *or*, *nor*, *xor*, *xnor*, *buf*, *not*, *bufif0*, *bufif1*, *notif0*, *notif1*.
- Warp does not allow assigning a value to a register variable using either blocking or non-blocking assignment.
- Nonblocking assignment statements within a function/task are not supported by Warp.
- Warp does not support parallel block.

- Warp partially supports *casex* and *casez* statements. For the *casex* statement, ?, x, z are allowed in a case-item expression but not allowed in a case expression. Similarly, for the *casez* statement ?, z are allowed in a case-item expression, but not allowed in a case expression.
- When Warp synthesizes any of the case statements, it synthesizes a memory element for each output assigned to it in the case statement, in order to maintain any outputs at their previous values, unless one of the following conditions occurs: 1) All outputs within the body of the case statement are previously assigned a default value within the always block, 2) The case statement completely specifies the design's behavior following any possible result of the conditional test.[107]
- Warp supports two kinds of loop statements: *for* and *while*.[108]
- Warp ignores the intra-assignment timing controls, delay-based timing controls and wait timing controls.[109]
- In structured procedures, Warp ignores the initial construct.
- Warp requires that an *always* statement have a sensitivity list.
- Warp ignores any timing controls present inside a task.
- Warp does not support the disabling of named blocks and tasks using the disable construct.
- Warp ignores all system tasks and system task functions.
- When an *ifdef* compiler directive is used, Warp compiles only the code within the *ifdef* Warp block.
- Warp issues a warning when it encounters any of the unsupported compiler directives.
- Warp does not synthesize tri-state logic.[110]
- Warp synthesizes a latch whenever a variable inside an *always* block, with an asynchronous trigger, has to hold its previous value.
- Warp uses the following templates to synthesize synchronous flip-flops. For a positive edge sensitive flip-flop:

```
always @ (posedge clock\_signal)
synchronous\_signal\_assignments
```

  and,

```
always @ (negedge clock\_signal)
synchronous\_signal\_assignments
```

```
is the template for the negative edge-sensitive flip-flop.
```

- Warp uses the following format to synthesize asynchronous flip-flops with reset or preset:

[107] The best way to ensure complete specification of design behavior is to include a default clause within the case statement. Therefore, to use the fewest possible resources during synthesis, either assign default values to outputs in the *always* block or make sure all case statements include a default clause.

[108] In Warp, the loop variable must be initialized to a constant value and the step assignment must be + or -.The following is the while loop template supported in Warp: *while (<comparison> <number>).*

[109] Event timing controls are partially supported (only *posedge* and *negedge* event timing controls are supported when used with an *always* @).

[110] In order to include tri-state logic in a module, the *cy_bufoe* must be instantiated. The tri-state output of this module, y, must then be connected to an *inout* port on the Verilog module. That port can then be connected directly to a bidirectional pin on the device. The feedback signal of the *cy_bufoe*, *yfb*, can be used to implement a fully bidirectional interface or can be left floating to implement just a tri-state output.

```
always @ (edge\_of clock\_signal or
          edge\_of preset\_signal or
          edge\_of reset\_signal)
   if (reset\_signal)
       reset\_signal\_assignments
   else if (preset\_signal)
       preset\_signal\_assignments
   else
       synchronous\_signal\_assignments
```

The *posedge* construct is used to specify an active high condition and the *negedge* construct is used to specify an active low condition. The variables in the sensitivity list can appear in any order. Subsequent reset or preset conditions can appear in the else-if statements. The last *else* block represents the synchronous logic. The polarity of the reset/preset signal condition used in the sensitivity list and the polarity of the reset/preset condition in the if/else-if statements must be the same.

- Warp allows more than two asynchronous *if/else-if* statements before a synchronous *else* statement.
- Warp allows the user to specify a particular case block to be implemented, e.g., a multiplexer (parallel case) rather than a priority encoder (full case). A parallel case or a full case is specified by including the directives *warp parallel_case* and *warp full_case* before a case statement. These directives can be specified within the Verilog comment section (line comment or block comment). The directive must follow the word "warp".

5.18 Verilog/Warp Component Examples

A common use for Verilog/Warp is the creation of special components, e.g., a divide by N, four bit counter can be easily created using the Verilog/Warp support provided by PSoC Creator, Cypress Semiconductor's development environment. After loading PSoC Creator and starting a new project, e.g., *CountByN*, navigate to the components tab in the Workplace Explorer and right click on Project 'CountByN'. This will bring up a menu from which you can select *Add Component Item*. The *Add Component* window will then appear at which point you must select *Symbol Wizard* and optionally provide a name for your component, e.g., *DivideByNCounter*. Next, click on the *Create New* button.

This will load the Symbol Creation Wizard whose window allows you to select the name, type and direction of the *DivideByN_Counter*'s terminals. Note that the counter output is labeled count[3:0] indicating that the output is four parallel bits. This window also displays a preview of the *DivideByN_Counter*'s symbol. In the current example, reset and clock are input terminals and count is a 4-terminal output as shown. Clicking OK will load the *DivideByN_Counter.cysym* page. Right click in area within this window away from the counter symbol. This will load a small menu from which you can select *Symbol Parameter...* At this point is necessary to define the parameter N in terms of its type and value. Select *int* and set the value to '1'.

Example 1: The source code template produced by PSoC Creator will be of the form:

```
//============================================================
module CountByN (
        count;
        clock;
        reset;
```

```
        );
                output [3:0]
                input clock;
                input reset;
        parameter N=1;
// '#start' body -- edit after this line, do not edit this line
        reg [3:0] count;
        always@(posedge clock or posedge reset)
                begin
                        if (reset) count \textless= 4'b0;
                        else count \textless= count + N;
                end
// '#end' -- edit above this line, do not edit this line
endmodule
```

Example 2: Similarly the Verilog/Warp code for a four bit counter with an enable terminal that count from 0 to some defined limit can be expressed as:

```
module Count4Enable  (
        count;
        clock;
        enable;
);
        output [3:0] count;
        input clock;
        input enable;
        Parameter Limit= 15;
//'#start body -- edit after this line, do not edit this line
        reg [3:0] count;
        always@(posedge clock)
        begin
                if (enable) begin
                        if (count == Limit) count = 4b'0;
                        else count <=count +1;
                        end
        end
//'#end' -- edit above this line, do not edit this line
//endmodule
```

Example 3: A Clocked register equivalent to can be expressed as:

```
        module DFF (
                clk;
                D;
                Q;
        )
                input clk;
                input D;
                output Q;
        // '#start' body -- edit after this line, do not edit this line
                reg Q;
                always@(posedge clk)
```

```
        begin
                Q <= D;
        end
//`#end` -- edit above this line, do not edit this line
endmodule
```

Example 4: A clocked register with an asynchronous reset can be implemented by the following:

```
module DFFR (
        clk;
        D;
        R;
        Q;

)
        input clk;
        input D;
        input R;
        Output q;
        reg q;
        always @ (posedge clk or posedge R)
        begin
                if (R) Q <= 1'b0;
                else Q <= D;
        end
endmodule
```

Example 5: A clocked register with an asynchronous "Set" can be implemented as:

```
module DFFS (
        clk;
        D;
        S;
        Q;
)
        input clk;
        input D;
        input S;
        output Q;
        reg Q;
        always @ (posedge clk or posedge S)
        begin
                if (S) Q <= 1'b1;
                else Q <= D;
        end
endmodule
```

Example 6: A 2-input, 1-output mux can be implemented by the following:

```
module muxA (
        sel;
        A;
        B;
```

```
        Z;
)
        reg Z;
        always@(sel or A or B)
        begin
                if (sel) Z = A;
                else Z = B;
                end
endmodule
```

Note that assignment in this example uses the "=" symbol since the assignments are combinatorial, i.e., there is no storage of values. This module is a representation of the boolean expression

$$Z = sel \cdot A + \overline{sel} \cdot B \tag{5.36}$$

5.19 Comparison of VHDL, VERILOG and Other HDLs

The decision as to which is the best approach to modeling a circuit or system depends heavily on the application, the technology to be employed, the sophistication of the designer, ease of associated tools use, steepness of the associated learning curves, compatibility with other tools, etc. Some designs are not appropriate for HDLs, e.g, simple designs, or designs that cannot take advantage of the benefits of HDLs.

Verilog is based on a simple language syntax and structure allowing a designer to learn Verilog quickly, model both digital and analog circuits[111] Verilog also allows a model's code to be monitored to identify errors at early stages in the design process. Verilog models typically require less memory and therefore often run significantly faster during simulations than is available from similar VHDL models.

VHDL does offer better reusability capability by allowing procedures and functions to be encapsulated in *packages*. VHDL supports libraries as stores for configurations, architectures and packages but no similar concept exists for Verilog.[112] Unlike Verilog, VHDL has functionality intended to facilitate the management of large designs, e.g., *generate* (structure replication), *generic* (generic models), *package*(model reuse) and *configuration* (design structure). Verilog supports reduction operators but VHDL does not. Verilog's support for system tasks and functions allows a designer to incorporate control commands into a description to facilitate debugging, this debugging technique is not supported in VHDL. However, concepts such as user-defined types are supported in VHDL but not in Verilog and there is much more support in VHDL for high-level modeling.

VHDL is often described as "verbose" when compared to other languages in that it offers more than one way of expressing things. VHDL is strongly-typed and Verilog is weakly-typed. VHDL provides a "rich" set of data types and Verilog is a smaller language and typically much easier to use. Verilog and VHDL are syntactically similar but there is no guarantee that Verilog models will behave the same in different tools. Verilog is generally regarded as much easier to learn than VHDL in part because Verilog is more "C-like" than VHDL.

[111] Verilog-AMS supports both analog and mixed-signal in part by supporting a continuous time simulator capable of solving differential equations in the analog domain and providing the ability to cross-couple the digital and analog domains.

[112] It should be noted that Verilog began life as an interpreter and therefore libraries were not supported.

SystemC[113] is sometimes used as an HDL to provide "VHDL-like" capability, but its use can be challenging, when modeling complex circuits. It allows the concept of time and concurrency to be employed in C++ applications, as for example when modeling synchronous hardware. Because it is C++ -based it is supported on a wide range of C++ platforms. SystemC has support for modules that communicate via ports, concurrent processes, channels[114], events, and fixed point/logic/extended standard data types.

The following is a example of a simple adder written in SystemC.

```
 include "systemc.h"
 #define WIDTH  4
 SC\_MODULE(adder) {
     sc_in<sc_uint\textless <WIDTH> >  a, b;
     sc_out sc_uint<WIDTH> > sum;
     void do_add() {
          sum.write(a.read() + b.read());
 }
SC_CTOR(adder)       {
SC_METHOD(do\_add);
sensitive << a <<  b;
 }
 };
```

5.20 Summary

In this chapter attention has been focussed on the virtues of programmable logic devices, boundary scanning techniques for testing programmable devices, Boolean functions and their simplification using Karnaugh maps. Macrocells and logic arrays are shown to form the basis for UDBs are discussed in some detail and the steps required to simplify Boolean expressions have been outlined in detail. Programmable logic devices based on combinations of macrocells and logic arrays and discussed in some detail and their use in one incarnation in the form of universal digital blocks. An integral part of using such devices is the ability to form and simply Boolean expressions derived from truth tables or Karnaugh maps. that represent the required logic.

A simple, but straightforward technique has been presented for evaluating Karnaugh maps, suggested by Mendelson [34], Harbort and Brown [10], et al, that simplifies Boolean expression to minimize hardware requirements in the subsequent implementations. PSoC3/5s universal digital block is discussed with resect to its internal architecture and relationship/interaction with the datapath. Backus-Naur notation is introduced within the context of a discussion of HDLs and the basic constructs of VHDL, Verilog and WARP are discussed and illustrated by example. Additionally, finite state machines are introduced and an example of a state machine implementation of a UART using Verilog was presented. PSoC3/5 architecture details and functionality were used throughout this chapter to illustrate key aspects of the material presented.

---

[113] SystemC is a collection of open source, C++ classes and macros that function as an event-driven simulation kernel that can be used to model concurrent processes capable of communicating in a simulated, real-time environment.

[114] Channels may be wires, bus channels, FIFO's, signals, buffers, semaphores,etc.

5.21 Exercises

1. Express the function given by Equation (5.22) in the form of a truth table. Use this table to sketch the associated logic diagram. Repeat this process for Equation (5.26) and write a brief comparison of the two logic diagrams listing the number and types of gates used in both cases.

2. Show that

$$F = A \cdot B \cdot \overline{D} + A \cdot \overline{B} \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{D} + A \cdot D + \overline{B} \cdot \overline{C} \cdot \overline{D}$$

can be reduced to

$$F = A + \overline{B} \cdot \overline{D}$$

3. Simplify the function F and show that F=1.[16]

$$F = A \cdot B \cdot C \cdot D + \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \overline{D} + A \cdot \overline{D} + \overline{A} \cdot B \cdot \overline{C} + A \cdot B \cdot \overline{C} \cdot D + B \cdot C \cdot D + \overline{A} \cdot C \cdot \overline{D}$$

4. Which of the following is a sum of products and which is a product of sums:

$$\overline{ABC} + A\overline{B}C$$
$$A\overline{B}\,\overline{C} + \overline{A}BC$$
$$(\overline{A} + B)(B + \overline{C} + D)$$
$$(A + \overline{B + C})(\overline{B + C})$$
$$\overline{A}\,\overline{B}\,\overline{C} + AB\overline{C} + A\overline{B}C$$

5. Sketch the logic circuit for: $(A + B + C)(\overline{A} + B + \overline{C})(A + B + \overline{C})$.

6. Show how to implement a NOT, OR and AND gate using 1, 2 and 3 NOR gates respectively.

7. Express each of the following expressions as a sum-of-products:

a) $(A + B) \cdot (\overline{A} + \overline{B})$

b) $A \cdot \ (B + C)$

c) $-(A + B \cdot \ C)$

8. Write a VHDL entity declaration with the following characteristics:

- Port A is a 12 bit output bus
- Port AD is a 12-bit, three-state bidirectional bus
- Port INT is a three-state output
- Port AS is an output that is used internally
- Port OE is an input bit
- Port CLK is an input bit

9. Given the following entity declaration for a comparator:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY compare IS PORT (
a, b: IN std\_logic\_vector(o TO 3);
aeqb: OUT std\_logic);
END compare;
```

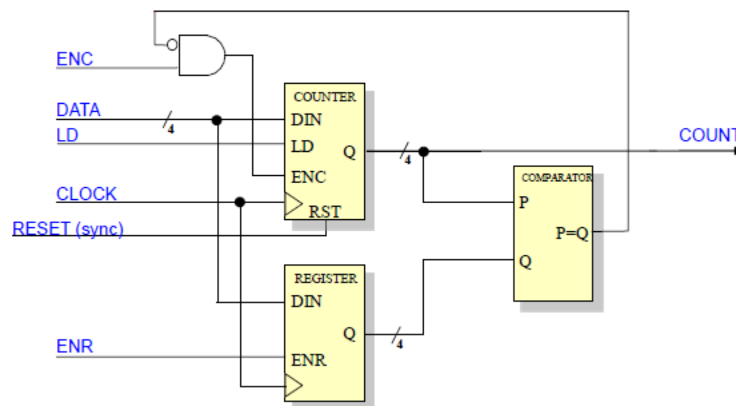
Write the VHDL code for an architecture that causes aebq to be asserted when a is equal to b using a) conditional assignment, b) boolean equations and c) a *process* with *sequential* statements.

10. Simplify $A \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot \overline{C}$ using a Karnaugh map.

11. The truth table for binary addition has three inputs: addedin, augend and carry in. The output consist of a sum and a carry. What is the truth table for the summing portion of binary addition? Simplify the expression representing this table by using a Karnaugh map.[115]

12. Draw the state diagram for a 3-bit binary counter as a state machine include any associated truth tables. Show how this counter can be implemented using combinatorial logic and D flip-flops.

13. Write an entity/architecture pair for the following circuit:



[115] Exercises 10 and 11 were suggested by Bob Harbort and Bob Brown, Computer Science Department, Southern Polytechnic State University.

Chapter 6

Mixed-Signal Processing

In this chapter, discussion focuses on mixed-signal processing[1]*, and in particular the various components often incorporated into an embedded system to provide the necessary functionality for a particular application. As in the previous chapters, PSoC3/5 serves the various needs for illustrative examples of key concepts throughout this chapter. It should be noted that many of the blocks, also referred to as "modules", found in devices such as PSoC3 and PSoC5 are in reality repeated instantiations of some fundamental hardware components with variations whose characteristics are controlled and/ or defined by registers. Therefore the discussion makes occasional references to the controlling and other related registers merely to highlight the functionality at a lower level of abstraction and to emphasize the fact that the behavior of the various modules can be changed dynamically under program control, and in real time.*

6.1 Mixed-Signal Evolution

Prior to 1970, applications of digital technology was somewhat limited as a result of the fact that vacuum tubes and the associated analog technology had dominated the world of electronic applications for more than two thirds of the 20th century. Although Transistor-Transistor Logic (TTL) was developed in 1961, the first widely-used, commercial versions, known as the "Texas Instruments 5400 Series" did not appear until 1963. This was soon followed, circa 1966, by Texas Instruments 7400 series which was widely adopted as the defacto standard for hardware logic components.

The 7400 series low cost, and the relative ease with which digital logic-based systems could be developed lead designers to make more and more use of microcontrollers and digital techniques. This was further motivated by the fact that analog components such as resistors, capacitors, inductors, as well as, vacuum tubes, tend to exhibit some degree of variation in component values as a function of aging, temperature, vibration, humidity, etc., which could substantially alter a system's performance However, given the fact that the real world is predominantly analog, it was necessary to combine both analog and digital techniques in implementing an embedded system in order to meet the requirements of increasingly more complex, and sophisticated embedded systems.[2] Most embedded systems involve both analog and digital signal processing techniques

[1]However, it has not been possible in this textbook to engage in a detailed discussion of signal processing. Instead, certain common signal processing applications will be discussed, e.g., mixing and other examples that involve both analog and digital signal processing, commonly encountered in embedded systems.

[2]Software-Defined Radios (SDRs), cell phones, digital television, etc., are excellent examples of a merging of digital and analog, i.e., mixed-signal, techniques to provide increasingly more sophisticated receivers and trans-

for handling I/O requirements, data acquisition and storage, data/signal conditioning, etc., and are hence often referred to as "mixed-signal" systems. PSoC3 and PSoC5 include both analog and digital modules that are interoperable, and highly configurable, as is shown in this chapter. Their mixed-signal architectures allow them to address a myriad of embedded system applications.

The reader would be well advised to bear in mind that in mixed-signal design it is often best to *Redde Caesari quae sunt Caesaris*[3] and use the digital/analog techniques and components that best meet the application's requirements and in the most beneficial combination. For example, in filter design, and deployment, there are frequency ranges for which digital techniques are quite unsuitable, in spite of their ability, in principle at least, to often provide far superior filtering than that of traditional analog techniques.

6.2 Analog Functions

Embedded systems are often called upon to handle a wide variety of inputs that include both digital and analog control/command signals. Analog functions employed in implementing embedded systems include:

- **Analog-to-Digital Conversion**
  Many transducers used in conjunction with embedded systems produce voltages or currents that are related to a parameter, or parameters, that the transducer is experiencing as inputs. Analog-to-digital conversion of the outputs of such transducers may be required to prepare these signals for acceptance and processing by the embedded system.
- **Current and Voltage Sensing**
  Transducers used in conjunction with embedded systems introduce a variety of voltage and current levels, some of which may be required to be in the form of analog signals that fall within certain ranges in terms of power, current and/or voltage.
- **Current and Voltage Output**
  Embedded systems are often required to provide specific current and voltage levels to external devices in ranges beyond the capacity of microcontrollers which are generally limited to output currents on the order of 25 milliamperes and voltages less than ± 12 vdc.
- **Analog Filters**[4]
  Many embedded systems include the ability to recover signals, remove interference, etc. and both digital and analog filters each play an important role in such cases. The most common types of filters are defined as:
  - highpass filters that pass frequencies above a certain frequency and block lower frequencies,
  - lowpass filters that pass frequencies below a certain frequency and block higher frequencies,
  - bandpass filters that pass all frequencies within a specified range and block all other frequencies,
  - notch filters, also referred to as bandstop or bandreject, filters that are extremely narrow band filters with steep sides that remove a narrow band of frequencies while passing, those frequencies above and below that band with constant, e.g., unity, gain,
  - allpass filters that pass frequencies within a specified range without altering their magnitude and at constant phase delay, and

mitters.

[3] "Render unto Caesar those things which are Caesar's ..."

[4] Assumed for the purpose of this section to be ideal filters, cf Section 6.10.1.

  – adaptive filters that can change their characteristics to meet changing conditions encountered by an embedded system.

- **Analog Mixing (Up-Conversion and Down Conversion)**
  Some transducers and other signal sources produce modulated carriers and the embedded system must be able to extract the data signal in such cases. Similarly some embedded applications require outputs to external devices in the form of modulated carriers.
- C**urrent-to-Voltage and Voltage-to-Current conversion**
  In addition to the fact that input signals can be in the form of current or voltage signals, the ability to convert from one to the other may be required be required for output signals provided by an embedded system.
- **Analog Signal Pre- and/or Post-Conditioning**
  It is often necessary to subject input signals to some form of pre- post-conditioning, e.g., filtering, voltage/current level shifts, up/down frequency conversion, etc., prior to/or after their processing by the embedded system
- **Amplification**[5]
  Amplification is an important consideration in many systems, whether as part of a signal conditioning requirement for input signals, or for driving external devices such as motors and other actuators.
- **Current/Voltage-to-Frequency Conversion**
  Depending on the type of I/O devices involved in an embedded system, it may be necessary to convert current and/or voltage to frequency.
- **Frequency-to-Voltage/Current Conversion**
  Depending on the type of I/O devices involved in an embedded system it may be necessary convert frequencies to voltages/currents.
- **Pulse Width Modulation/Demodulation**
  PWMs are often used to provide proportional control of external devices, e.g., motors, illuminators, etc., and demodulation of pulse width modulated signals.
- **Integration**
  Integration of signals is sometime employed as part of the input signal conditioning, or for other reasons, in an embedded system.
- **Pulse Shaping**
  Pulse re-shaping may be required to restore a signal that has been distorted, e.g., to improve pulse width, pulse height, overall shape and/or timing.
- **Differentiation**
  Some embedded systems require that analog signals be differentiated.
- **Analog Voltage-to-Reference Voltage Comparison** Many embedded systems employ comparators to compare an input signal to a reference value which serves as a threshold for action, or inaction, by the embedded system.
- **Track and hold amplifier**
  Track and hold amplifiers are used to maintain a signal input level, i.e., a sample, for a period of time to allow processing of the sample to be completed before the system accepts the next sample to be processed.
- **Unity Gain Buffer**
  Such buffers are used to avoid overloading a previous stage, or input source.

[5] Amplification of this type is sometimes referred to as "multiplication".

- **Voltage Summing**
  Allows an embedded system to sum multiple input/output signals.
- **Logarithmic Amplifier** "Log amps" are often used with transducers, or other devices, with a wide dynamic range in order to bring both high and low level signals into an acceptable range for either input to, or output from, an embedded system
- **Exponential Amplifier**
  These amplifiers can be used with sensors, or other sources, producing logarithmic signals.
- **Instrumentation Amplifier**
  These amplifiers are often used to make high accuracy, non-perturbing voltage/current measurements in an embedded system application.
- **Digital-to-Analog Conversion** OpAmps are sometimes used in conjunction with other components to provide an analog output of a digital input.

Operational amplifiers frequently play an important role with respect to the providing these types of analog functions, as shall be shown in the remaining discussions in this chapter. Much of the discussion that follows focuses on idealized operational amplifiers. Related discussions of some of the analytical aspects of non-ideal operational amplifiers can be found in Appendix E.

6.2.1 Operational Amplifiers (OpAmps)

The "Operational Amplifier" or "OpAmp"[6] was developed in the 1930s, under a Federal grant, with the specific goal, among others, of finding a replacement for mechanical integrators used in various military applications[7], e.g., the ball and disk integrator[8] which was subject to slippage and therefore error.[15] The result was a vacuum tube based design, that was to serve as a key component in a wide variety of military and civilian applications, and ultimately become the basic building block for a large number of early analog computers. The following example clearly illustrates how one might use such devices as the basis for analog computers capable of solving, inter alia, differential equations[9].

Consider the differential equation

$$\frac{d^2x}{dt^2} = -\omega^2 x \tag{6.1}$$

Given an electrical, or mechanical, integrator with the property[10] that

$$f(t)_{out} = -\int g(t)_{in} \tag{6.2}$$

it follows that substituting the LHS of Equation (6.1) into Equation (6.2) yields -dx/dt which can then be substituted into Equation (6.2) for a second time to produce x. If x is then multiplied by $-\omega^2$, the result is equal to d^2x/dt^2. The OpAmp configuration capable of solving Equation (6.1) is shown in Figure 6.1. This configuration is sometimes used as a sine wave generator to produce tones.

---

[6]Sometimes referred to as an "OpAmp".
[7]Airborne Sextants, fire control systems, etc.
[8]Hence the name operational amplifier because it was to perform the "operation" of integration.
[9]Analog computers have also proven superior to digital computers in solving certain types of partial differential nonlinear equations.
[10]Unless otherwise noted functions such as f(t) and g(t) are assumed to be "well-behaved".

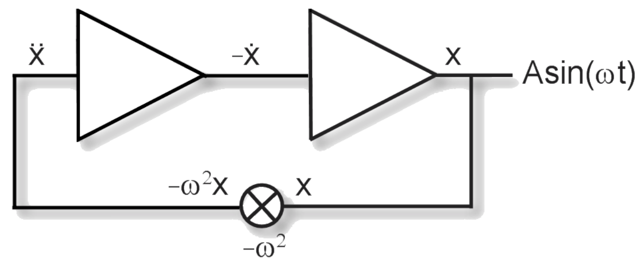


Figure 6.1: An analog computer solution of a differential equation.

It should be noted that this method of solving differential equations, and in particular complex systems of differential equations was to remain in popular use for almost fifty years. Much of the early verification of various aspects of chaos theory was carried out on analog computer systems.

The earliest commercially available operational amplifiers, circa 1941, were general purpose, DC-coupled[11], voltage amplifiers that had high gain and employed a feedback loop. The passive components employed in the feedback loop, and for input, usually a capacitor or resistor, allowed operational amplifiers to serve as integrators, differentiators, summers, scalers, multipliers, followers, etc. A short list of possible operational amplifier configurations is shown in Table 6.1. They were followed by a succession of solid state devices, initially based on discrete transistors

Table 6.1: Examples of OpAmp applications

OpAmp Applications	
Differentiator	Integrator
Summer	Subtractor
Multiplier (Amplifier)	Differential Amplifier
Preamp	Buffer
Precision Rectifier	Voltage Clamp
Oscillator	Waveform Generator
Pulse Shaper	Comparator
Analog Filter	Current/Voltage Regulator
Voltage-to-Current Converter	Current-to-Voltage Converter
Voltage-to-Frequency Converter	Frequency-to-Voltage Converter
Constant Current Source	Constant Voltage Source
Transimpedance Amplifier	Voltage Follower
Reference Voltage Supply	Current Injector
Phase Lead/Lag	Time Delay
Absolute Value	Peak Follower
AC to DC Converter	Full Wave Rectifier
Rate Limiter	

(1961) and ultimately in the form of integrated circuits, most notably the μA709 (1965) operating at significantly lower supply voltages, e.g., ± 15 vdc.

However, these early solid state devices devices were prone to oscillate, sometimes at such high frequencies, that the then commonly available oscilloscopes had a hard time"seeing" the oscil-

[11]DC-coupled refers to the fact that the amplifiers could handle both DC and AC signals.

lations. This type of oscillation was to plague some designers so much that they referred to operational amplifiers as "operational oscillators". Oscillation, and other problems associated with the $\mu A709$, were resolved in 1968 with the introduction of the $\mu A741$ which remains the low cost OpAmp of choice for many applications to this day.

The development of the Field Effect Transistor led to the introduction of FET-based OpAmps as the next step in the evolution of operational amplifiers providing much higher input impedances[12] and therefore significantly lower input currents and the capability of operating at much higher frequencies. Additionally, the requirement for external dual power supplies for OpAmps was removed by the introduction of devices such as the $LM324$ (1972) which has multiple OpAmps in a single package and operates from a single external supply.[13]

Modern operational amplifiers are usually classified in terms of their input/output type as a

- voltage-controlled voltage source (VCVS) whose gain is represent by A_o and defined as the ratio of output voltage to input voltage (v_o/v_i),
- voltage-controlled current source (VCCS)whose gain is represent by the symbol g_m as the ratio of output current to input current,
- a current-controlled voltage source (CCVS) represented by the symbol r_m and defined as the ratio of output voltage to input current (v_o/i_i).

or,

- current-controlled current source (CCCS) represented by the symbol A_i and defined as the ratio of output current to input current (i_o/i_i).

Examples of just a few of the many applications of operational amplifiers are given in Table 6.1.

6.3 Fundamental Linear System Concepts

Before proceeding with a discussion of operational amplifiers, analog/digital filters and other topics discussed in this chapter, some fundamental concepts must be introduced. Important definitions and figures of merit related to operational amplifiers will be presented to help characterize the behavior of operational amplifiers in a variety of configurations commonly found in, or related to, embedded systems. It has of course not been possible to cover these topics in great detail but s number of references are provided that should be of help for those interested in more detailed discussions.

6.3.1 Euler's Equation

Leonhard Euler (1707-1783) a Swiss physicist and mathematician made a number of important contributions to science including his discovery that

$$e^{j\theta} = \cos(\theta) + \mathrm{j}\sin(\theta) \tag{6.3}$$

and therefore

$$\mathrm{e}^{-j\theta} = \cos(\theta) - \mathrm{j}\sin(\theta) \tag{6.4}$$

[12] The input impedance of a typical $\mu A741$ is of the order of $2\,\mathrm{M}\Omega$. OpAmps with input impedances that exceed $10^{12}\,\Omega$ are now available.

[13] It should be noted that the $LM324$ is inherently a dual supply system. However, by employing a "virtual" ground it is possible to operate its OpAmps using only a single supply.

which leads to the important results that

$$\sin(\theta) = \frac{e^{j\theta} - e^{-j\theta}}{2j} \tag{6.5}$$

$$\cos(\theta) = \frac{e^{j\theta} + e^{-j\theta}}{2} \tag{6.6}$$

and because

$$\theta = \omega t = 2\pi f t = \frac{2\pi t}{T} \tag{6.7}$$

it follows that

$$\sin(\omega t) = \frac{e^{j\omega t} - e^{-j\omega t}}{2j} \tag{6.8}$$

$$\cos(\omega t) = \frac{e^{j\omega t} + e^{-j\omega t}}{2} \tag{6.9}$$

so that for well-behaved functions, i.e., functions that are continuous, periodic, etc., can be expressed as an infinite complex exponential series, viz.,

$$f(t) = \sum_{k=-\infty}^{\infty} g_k e^{-jk\omega_0 t} \tag{6.10}$$

which expresses a continuous, periodic function in the time domain as an infinite sum of discrete values in the frequency domain and ω_0 the fundamental frequency and its harmonics represented by $k\omega_0$.

If the function f(t) is an aperiodic, continuous-time signal it can be expressed in terms of a complex integral, known as the Fourier Transform, as

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} G(j\omega)e^{j\omega t}d\omega = \int_{-\infty}^{\infty} G(j2\pi f)e^{j2\pi f t}df \tag{6.11}$$

6.3.2 Impulse Characterization of a System

By determining the response of a LTI system to a very fast input pulse it becomes possible to ascertain the system's response to an arbitrary input. This type of analysis is facilitated by an important class of functions known as generalized functions which are particularly useful in understanding the behaviour of embedded systems.

Two of these functions are the Kronecker and Dirac delta functions. These functions have some very unique properties, e.g., the Dirac delta function, also known as the unit impulse function, is defined as

$$\delta = \begin{cases} \infty & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases} \tag{6.12}$$

subject to the constraint that

$$\int_{-\infty}^{\infty} \delta(x)dx = 1 \tag{6.13}$$

The Dirac delta function also has the property called *sampling* or *sifting* , viz.,

$$\int_{-\infty}^{\infty} f(x)\delta(x-x_0)dx = f(x_0) \tag{6.14}$$

The Kronecker delta function is given by

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{6.15}$$

or as an integer function as

$$\delta[n] = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n \neq 0 \end{cases} \tag{6.16}$$

Thus

$$\sum_{i=-\infty}^{\infty} a_i \delta_{ij} = a_j \tag{6.17}$$

In the case of continuous-time systems the Dirac delta function is used as the impulse function and for discrete-time systems the Kronecker delta function is used. A system's response to an impulse function is called the the *impulse response function.* As shown in a later section of this chapter, the Laplace transform of the impulse response function is the system's transfer function. Both the Kronecker and the Dirac delta functions are mathematical models of a real world pulse that can be used to determine the behavior of discrete- and continuous-time systems, respectively.

6.3.3 Fourier, Laplace and Z Transforms

Engineers, scientists and a variety of technologists often rely on a family of mathematical tools known collectively as "transforms". These powerful tools make it possible to analyze, in considerable detail, a wide variety of physical systems and phenomena. By transforming a problem into a different function space it is often possible to gain considerable insight into the characteristics and behavior of a system while avoiding what can be substantial mathematical analysis challenges. Inverse transforms are also available which allow the completed analysis to then be returned a spatial or temporal domain from which it originated.

Signals can be broadly classified as either periodic, or aperiodic, and discrete, or alternatively as continuous and either periodic or continuous. Powerful tools are often required to investigate such a diversity of signals and their processing.

In this and other chapters use will be made of the:

- Laplace transform[14] which originally developed as a technique for solving ordinary differential equations (linear). It provides a method of mapping continuous time-domain functions to the s-domain where $s = \sigma + j\omega$ that is defined in bilateral form as

$$\mathcal{L}\{f(t)\} = \int_{-\infty}^{\infty} f(t)e^{-st}dt \tag{6.18}$$

[14]MatLab provides, as part of its symbolic toolbox, laplace() and ilaplace() functions to computer Laplace transform and the inverse Laplace transform of a function, respectively.

- Fourier transform[15] which is a method of solution of differential equations that provides the steady state response of a system. It can be used to map discrete time signals, that is continuous[16], periodic functions to/from the frequency domain.

$$\mathcal{F}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}d\omega \tag{6.19}$$

- Fourier Series is a method of expressing well-behaved, continuous function in terms of an infinite series, or approximated by a a partial sum thereof, consisting of sine and/or cosine terms. This series produces a frequency domain representation of a periodic, continuous-time signal.

$$f(x) = a_0 + \sum_{n=1}^{N}[a_n cos(nx) + bsin(nx) \tag{6.20}$$

$$a_n = \int_{-\pi}^{\pi} f(x)cos(nx)dx \quad n \geqslant 0 \tag{6.21}$$

$$b_n = \int_{-\pi}^{\pi} f(x)sin(nx)dx \quad n \geqslant 1 \tag{6.22}$$

and the

- Z-transform which is the discrete-time equivalent of the Laplace transform and is a mapping from the time-domain to the z-domain and expressed as

$$\mathbf{Z}\{x[n]\} = \mathbf{X}(z) = \sum_{n=-\infty}^{\infty} x[n]z^{\{-n\}} \tag{6.23}$$

6.3.4 Linear Time Invariant Systems (LTIs)

A system which is definable in terms of a single input x(t) signal and a single output signal y(t) (SISO) such that there exists an F(x(t)) for which

$$y(t) = F(x(t)) \tag{6.24}$$

is said to be "linear"[17] if

$$F(x_1(t) + x_2(t)) = F(x_1(t)) + F(x_2(t)) \tag{6.25}$$

and,

$$F(ax(t)) = aF(x(t)) \qquad \forall a \in \Re \tag{6.26}$$

Furthermore, if

$$y(t-T) = F(x(t-T)) \qquad \forall T \in \Re \tag{6.27}$$

[15]The Fourier transform is equivalent to the Laplace transform when $s = j\omega$.

[16]The reader is cautioned to delineate between continuous time functions and the mathematical meaning of "continuous" when refereeing to a aperiodic functions.

[17]It is often suggested that nonlinear system with nonlinear terms that are deemed "small" can be treated as linear. However, in some systems it is the existence of small terms and not their magnitude which determines whether the system will behave in a quasi-linear fashion or is capable of becoming significantly nonlinear. If the signal levels are sufficiently low it may be possible to constrain a system to operating in a linear region, e.g. as is often the case with transistors.

the system is said to be linear and time invariant, or equivalently LTI. Linearity gives rise to a number of important benefits perhaps the greatest of which in the present context is superposition which allows the response of a LTI system to be determined by inputting the individual components of a signal into a system, determining the output in each case and then summing the individual responses to obtain the overall response of the system to the composite input signal.

If there exists a function h(t) referred to as the impulse function such that:

$$y(t) = \int_{-\infty}^{\infty} h(\nu)x(t-\nu)d\nu \tag{6.28}$$

the system can also be said to be LTI. Conversely, if a system is LTI then there exists an impulse function, $h(\nu)$. Equation (6.28) is called the *convolution integral* and h(ν) is referred to as the "unit impulse response". A step function can also be used to characterize a system just as completely as the unit impulse. However, for the purposes of these discussions an impulse function will suffice.

The existence of an impulse response function for a system allows an arbitrary input to be represented as a set of impulse functions of the appropriate amplitude and the response of the system to each such impulse function determined so that the response to an arbitrary input can be viewed as the sum of the responses to the impulse functions making up the input signal. The process of decomposing the input signal into a series of impulses is referred to as *impulse decomposition*. The combining of the resulting impulse responses is referred to as *synthesis*. However, there is an even simpler technique which relies on the impulse being known and the existence of an analytic expression for the input. This process is known as *convolution* and is discussed in a later section.

The characterization of systems and signals of the types under discussion in this and subsequent chapters typically depend less on the shape of the time domain input waveform, and more on their respective amplitude (gain) amplitude, frequency and phase of its spectral components. Therefore the ability to map time domain functions that fully embody the system's characteristics into the frequency domain is an important part of predicting behavior. Any LTI[18] system can be characterized, in principle at least, by its transfer function which is simply a function that gives the output of the system as a function of the input[19], or in more formal Laplace terms, a transform function, H(s), for a LTI system, is a linear mapping by the Laplace transform[20], $\mathcal{L}\{f(t)\}$, of the input, referred to X(s), to the output, Y(s) where the Laplace Transform is defined as

$$\mathcal{L}\{f(t)\} = \int_{0}^{\infty} f(t)e^{-st}dt \tag{6.29}$$

and its inverse[21], as

$$\mathcal{L}^{-1}\{f(s)\} = -\frac{1}{2\pi j}\int_{\alpha-j\infty}^{\alpha\ +j\infty} f(s)e^{st}ds \tag{6.30}$$

and for which $s = \sigma + j\omega$.

---

[18] Linear Time Invariant systems, in addition to being linear, exhibit no explicit time dependence. Such systems are completely characterized by the system's impulse response, or equivalently its step response.

[19] Assuming zero initial conditions.

[20] The Laplace transform allows LTI systems to be mapped to the frequency domain and completely characterized by their respective frequency transfer function H(s).

[21] Using the integral form of the inverse Laplace transform requires integration in the complex plain and it is often preferable to instead rely on partial fraction expansion, i.e., a sum of simpler fractions, and tables of known transforms.

Example 6.1
MatLab provides a very convenient method for finding the inverse Laplace transform of a complex function in the form of the "ilaplace" operator.

Assuming that:

$$H(s) = \frac{a_m s^m + a_{m-1} s^{m-1} + \ldots + a_1 s + a_0}{b_n s^n + b_{n-1} s^{n-1} + \ldots + b_1 s + b_0} = \frac{s(s-7)(s+4)}{(s+2)(s^2+5s+6)}$$

$[MatLab]$

$$>> \text{ilaplace}((s*(s-7))/((s+2)*(s^2+5*s+6)))$$
ans = 30*exp(-3*t)+(-29+18*t)*exp(-2*t)

6.3.5 Impulse and Impulse Response Functions

A transfer function is defined in the Laplace domain as the ratio of the output function to the input function assuming that the initial conditions are zero.

Therefore if

$$\mathcal{L}\{y(t)\} = Y(s) \tag{6.31}$$

and

$$\mathcal{L}\{x(t)\} = X(s) \tag{6.32}$$

Then the transfer function for a given system leads to the following

$$H(s) = \frac{Y(s)}{X(s)} \tag{6.33}$$
$$Y(s) = H(s)X(s) \tag{6.34}$$
$$y(t) = \mathcal{L}^{-1}\{H(s)X(s)\} \tag{6.35}$$

Table 6.2 shows that the Laplace transfer of the Dirac delta function is

$$\mathcal{L}\{\delta(t)\} = 1 \tag{6.36}$$

so that if $x(t) = \delta(t)$, $Y(s) = (1)H(s)$ and therefore the *impulse response* of a system occurs when an *impulse function* is applied to the input.

The computation of Laplace transform of a function f(t) is relatively straightforward because it involves integration in the real domain as opposed to the computation of the inverse Laplace transfer which is defined in terms of integration in the complex domain. In most cases the computation of the Laplace transform is straightforward and the explicit and sometimes tedious computation of the inverse based on integration in the complex plane can be avoided by employing tables of Laplace transform inverses. Table 6.2 shows some of the more common Laplace Transforms. A combination of partial fraction expansion and utilization of such tables is much easier than having to employ integration techniques in the complex plane. MATLAB provides an even simpler approach as shown in Example 6.2.

Example 6.2 MATLAB can be used to find the impulse response of a system's transfer function, e.g.,

$$H(s) = \frac{s+2}{s^3+4s^2+5} = \frac{1s^1+2}{1s^3+4s^2+0s^1+5} \tag{6.37}$$

by the following:

$[MatLab]$

```
>> num = [1 2]
>> den = [1 4 0 5]
>> impulse(num, den)
```

The graphical result is shown in Figure 6.2.

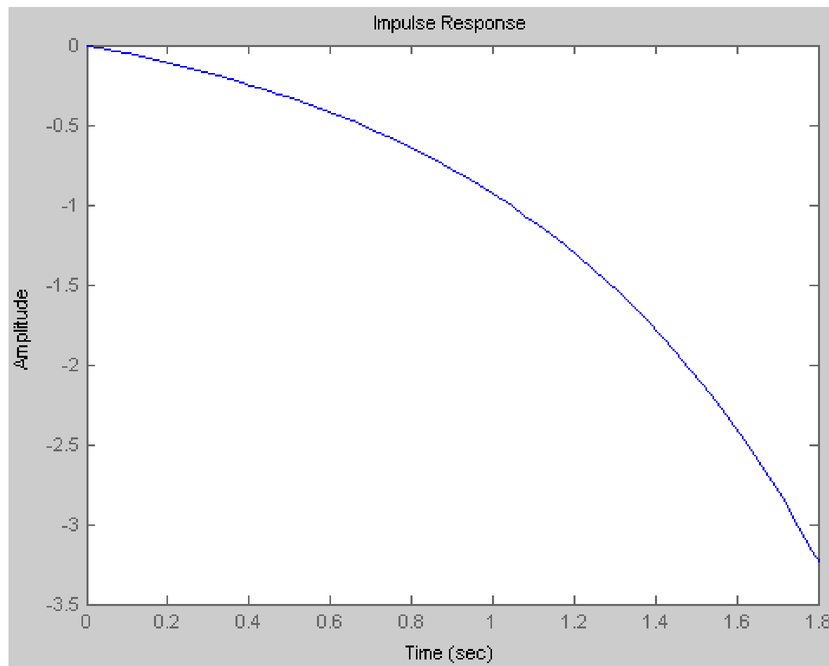


Figure 6.2: Impulse response for Example 6.2.

6.3.6 Transfer, Driving and Response Functions

Consider a causal[22], linear. time invariant (LTI) system which has a single input and single output (SISO) that can be represented by an ordinary differential equation with constant coefficients, e.g.,

$$y^n + a_1 y^{(n-1)} + \ldots + a_{n-2}\ddot{y} + a_{n-1}\dot{y} + a_n y = b_1 x^m + b_2 x^{(m-1)} + \ldots + b_{m-1}\ddot{x} + b_m\dot{x} + b_{m+1} \tag{6.38}$$

[22]Causal systems are defined as systems for which the output at a particular time, t_0, depends only on the input for $t \leq t_0$ and not on any time in the future, i.e. for all $t \geq t_0$.

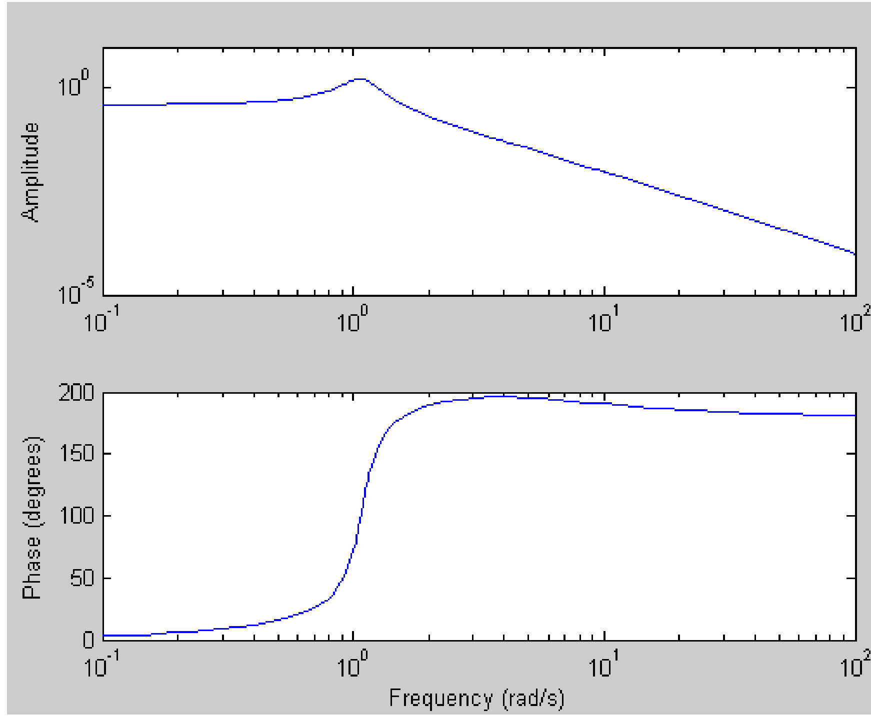


Figure 6.3: Bode plot for Equation (6.37).

Taking the Laplace Transform of both sides yields:

$$Y(s) = H(s)X(s) \tag{6.39}$$

and therefore,

$$H(s) = \frac{Y(s)}{X(s)} \tag{6.40}$$

where,

$$\mathcal{L}\{x(t)\} = \int_0^\infty x(t)e^{-st}dt = X(s) \tag{6.41}$$

and,

$$\mathcal{L}\{y(t)\} = \int_0^\infty y(t)e^{-st}dt = Y(s) \tag{6.42}$$

Y(s) is referred to as the response function, X(s) as the driving function and H(s) as the transfer function. The most general form[23] of a transfer function for continuous-time systems of the type under discussion is represented by:

$$H(s) = \frac{a_m s^m + a_{m-1}s^{m-1} + \ldots + a_1 s + a_0}{b_n s^n + b_{n-1}s^{n-1} + \ldots + b_1 s + b_0} = \frac{M(s)}{N(s)} \tag{6.43}$$

[23]It assumed that at least for the sake of this discussion that H(s) is a rational function, i.e., it can be written as the ratio of two polynomials which is usually the case.

Table 6.2: Some Common Laplace Transforms.

Time Domain	Description	Frequency Domain
δ	Unit Impulse	1
A	Step	$\frac{A}{s}$
t	Ramp	$\frac{1}{s^2}$
e^{-at}	Exponential Decay	$\frac{1}{s+a}$
$sin(\omega t)$	Sine Function	$\frac{\omega}{s^2+\omega^2}$
$cos(\omega t)$	Cosine Function	$\frac{s}{s^2+\omega^2}$
te^{-at}		$\frac{1}{(s+a)^2}$
t^2e^{-at}		$\frac{2}{(s+a)^3}$
$e^{-at}sin(\omega t)$	Decaying Sine	$\frac{\omega}{(s+a)^2+\omega^2}$
$e^{-at}sin(\omega t)$	Decaying Cosine	$\frac{s+a}{(s+a)^2+\omega^2}$

and in an equivalent factored form as:

$$H(s) \quad = \quad \frac{(s-z_{m)}(s-z_{m-1})\ldots(s-z_2)(s-z_1)}{(s-p_{n)}(s-p_{n-1})\ldots(s-p_2)(s-p_1)} \tag{6.44}$$

As discussed in Section 6.10.3 of this chapter, the transfer function for a simple RC circuit is given by:

$$H(s) = \frac{sRC}{1+sRC} \tag{6.45}$$

which has both a zero for s = 0 and a pole for s=-1/RC. Poles/Zeros refer to points in the complex plane for which the denominator/numerator of the transfer function becomes zero, respectively.

This can be formally expressed as

$$lim_{s\to z_i}\text{H(s)} = 0 \tag{6.46}$$

and

$$lim_{s\to p_i}\text{H(s)} = \infty \tag{6.47}$$

for the general form of a transfer function of the type shown in Equation (6.44) which is expressed in terms of the roots of the denominator and numerator of a complex transfer function. If the system is to be stable, then the poles must lie in the left hand side of the complex plane.

6.3.7 Common Mode Voltages

OpAmps are inherently two input devices and therefore input signals[24] that are common to both must be taken into account when analyzing an OpAmps characteristics. The *common mode input* voltage is defined as:

$$v_{icm} = \frac{(v_{i1} + v_{i2})}{2} \tag{6.48}$$

Similarly the ***common mode output*** voltage is defined as:

$$v_{ocm} = \frac{(v_{o1} + v_{o2})}{2} \tag{6.49}$$

6.3.8 Common Mode Rejection

Embedded systems employing OpAmps are often in environments containing a variety of sources of electronic noise[25], as well as, signal. An important figure of merit for an OpAmp is the value of a parameter known as the Common Mode Rejection Ratio, or CMRR.

The output of an OpAmp can be expressed as

$$v_o = A_d(v_{i1} - v_{i2}) + A_{cm}\left[\frac{v_+ + v_-}{2}\right] \tag{6.50}$$

where A_d is the differential gain and A_{cm} is the common mode gain. CMRR is a quantitative measure of a device's ability to reject common mode signals, i.e., signals applied to both inputs and has been formally defined by the IEEE as:

$$CMRR = 10\,log_{10}\left[\frac{A_d^2}{A_{cm}^2}\right] = 20\,log_{10}\left[\frac{A_d}{\mid A_{cm} \mid}\right] \tag{6.51}$$

Obviously it is desirable for the CMRR to be as low as possible, particularly when the signal of interest is small relative to the ambient common mode signals such as signals originating from thermocouples, thermistors, etc.

6.3.9 Total Harmonic Distortion (THD)

An important parameter for many devices and applications with both inputs and outputs is known as Total Harmonic Distortion, or THD. Nonlinearities in a system can give rise to unwanted harmonics which are "injected" into a signal, and THD is an important measure of such effects. In the case of a pure sine wave, THD is defined as the ratio of the sum of the higher harmonics present to the first harmonic of the distorted signal, i.e.,

$$\text{THD} = \sum_{n=2}^{\infty} \frac{P_n}{P_1} = \sum_{n=2}^{\infty} \frac{V_n^2}{V_1^2} = \frac{P_{total} - P_1}{P_1} \tag{6.52}$$

[24] In this case inclusive of signals containing or representing noise. Note that in some environments it is possible for the desired signal to appear on both inputs albeit at different signal strengths.

[25] Noise is sometimes referred to as "the part you don't want", whereas signal is defined as "the part you do want".

where P_n is the power of the nth harmonic and P_{total} represents the total power of the distorted signal and V_n is the amplitude of the voltage of the nth harmonic. THD is sometimes also combined with noise and defined as:

$$\text{THD + N} = \frac{\sum \text{Harmonic Power + Noise Power}}{\text{Fundamental Power}} \tag{6.53}$$

If the output signal is weakly distorted, it is possible to use a Taylor series[26] expansion to model the output signal in terms of the input signal and thereby quantify the distortion.[13]

That is,

$$v_o = a_0 + a_1 v_i^2 + a_3 v_i^3 + a_4 v_i^4 + \cdots = \sum_{n=0}^{\infty} a_n v_i^n \tag{6.54}$$

where,

$$a_n = \frac{1}{n!}\left[\frac{d^n v_o}{dv_i^2}\right]_{v_i=0} \tag{6.55}$$

which for an input of the form

$$v_i = V cos(wt) \tag{6.56}$$

can be expressed as

$$v_o = \left[a_0 + \frac{1}{2}V^2 a_2\right] + \left[a_1 + \frac{3}{4}V^2 a_3\right] V cos(\omega t) + \left[\frac{a_2}{2}\right] V^2 cos(2\omega t) + \left[\frac{a_3}{4}\right] V^3 cos(3\omega t) + \cdots \tag{6.57}$$

where a_0 and a_1 represent the DC component and circuit gain, respectively. This result shows that second and third order harmonics occur within the first four terms of this series which for many applications is sufficient to characterize the harmonics distortion. Second and third order distortion are defined as

$$\text{HD}_2 = \frac{1}{2}\frac{a_2}{a_1}V \tag{6.58}$$

$$\text{HD}_3 = \frac{1}{4}\frac{a_3}{a_1}V^2 \tag{6.59}$$

and the total harmonic distortion is given by

$$THD = \sqrt{HD_2^2 + HD_3^2 + HD_4^2 + \cdots} \tag{6.60}$$

6.3.10 Noise

Noise[27] has been characterized as "... the part we don't want..." and it is present in every real world system to a greater or lesser degree. Before beginning a discussion of noise it will

[26] The Taylor series is a series expansion of a function based on its value and that of its derivatives at a single point. In this particular case, the series is actually a Maclaurin Series since it is being evaluated at the origin, i.e., in the neighborhood of $v_i = 0$.

[27] "Like diseases, noise is never eliminated, just prevented, cured, or endured, depending on its nature, seriousness, and the cost/difficulty of treating it." from Analog-Digital Conversion Handbook, by D.H. Sheingold, Analog Devices.

prove helpful to define some key concepts, e.g., methods for arriving at average values for a given parameter. Root mean square, or RMS as it is commonly referred to, is defined by the following:

$$\text{RMS value of } f(t) = \sqrt{\frac{1}{T}\int_0^T f^2(t)dt} \tag{6.61}$$

where T represents a characteristic time interval, e.g., the period of the function f(t). In case of truly random noise[28], its average value will be zero, however it does in power being dissipated, thus the RMS value of the noise is an important parameter when considering circuit/device noise.

Example 6.3
Assuming a frequency of 60 Hz and a wave form given by $f(t) = (1.697\text{x}10^{-3})sin(t)$, Equation (6.61) becomes

$$RMS\, f(t) = \sqrt{\frac{1}{T}\int_0^T [169.7\sin(\text{t})]^2(t)dt} = \frac{1.697\text{x}10^{-3}}{\sqrt{2}} \approx 1.20\,\text{millivolts}$$

for $T = 1.66\text{x}10^{-2}$, i.e. 60 Hz.

Noise is present in all circuits, and in multiple forms, including:

- *White Noise*[29] is a generic term that refers to any noise source for which noise as a function of frequency is constant, and usually within a specified range.
- *Thermal*[30] - J. B. Johnson [16] was the first to report the existence of thermal noise by noting that the statistical fluctuation of electric charge in conductors results in a random variation in potential across a conductor. H. Nyquist [32] confirmed Johnson's observations by providing a theoretical basis for what Johnson had observed. The random motion of charge carriers gives rise to a what is approximately Gaussian noise, i.e., statistical noise with a probability density that is a *normal* distribution, i.e. Gaussian. In the case of resistors, the RMS value of the voltage associated with such noise is given by:

$$v_{rms} = \sqrt{4kT\Delta fR} \tag{6.62}$$

  and therefore where k is Boltzmann's constant, T is the temperature of the resistor in Kelvin, Δf is the bandwidth of interest, and R is the value of the resistance. Note that if both sides of Equation (6.62) are squared then

$$v_{rms}^2 = 4kT\Delta fR \Rightarrow \frac{v_{rms}^2}{R} = 4kT\Delta f = P \tag{6.63}$$

  which is the noise power, P, dissipated in the resistor[31]. The noise power spectral density is a measure of the noise present in a 1 Hz bandwidth and is defined as:

$$P_{sd} = \frac{P}{\Delta f} = 4kT \tag{6.64}$$

[28] Noise that is in reality completely random probably doesn't exist, but for the sake of these discussion "relatively random" shall suffice.

[29] A true white noise source would be required to supply infinite energy across an infinite spectrum, therefore physical white noise sources are necessarily restricted to finite portions of the spectrum. Approximations to a white noise source are sometimes referred to as non-white, colored or pink noise sources.

[30] Thermal noise is also referred to as Johnson, Nyquist or Nyquist-Johnson noise.

[31] Some portion of the noise can also be distributed throughout any circuit that the resistor is connected or that is within close proximity. Since the noise power is directly proportional to Δf, one way to minimize circuit noise is to limit the bandwidth as much as possible

and has units of V^2/Hz. Thermal noise, in the case of an MOS device, is modeled as a current source in parallel with drain and source. Noise in resistors can be modeled as a noise source in series with an ideal noise-free resistor with the noise power being expressed as the ratio of noise power to 1 milliwatt and designated as dBm.

Example 6.4
Noise power relative to 1 milliwatt can be expressed as

$$P_{rel} = 10\log_{10}\left[\frac{P_{noise}}{1x10^{-3}}\right] = 10\log_{10}\left[P_{noise}\right] + 30 \quad \text{dBm}$$

so that in the case of thermal noise in a resistor, if $R = 50\,\Omega, \Delta f = 10\,kHz$ and T = 300 K

$$P_{rel} = -134dBm \tag{6.65}$$

- *Flicker noise* is modeled as a voltage source in series with the gate for example of a CMOS, MOSFET or similar device and results from trapped charged carriers. It is inversely proportional to frequency and is related to DC current flow. The average mean square value is given by:

$$\overline{e^2} = \int \left[\frac{K_e^2}{f}\right] df \tag{6.66}$$

$$\overline{i^2} = \int \left[\frac{K_i^2}{f}\right] df \tag{6.67}$$

  where K_e and K_i are voltage and current constants, respectively, for the device under consideration and f is frequency.
- *Burst* noise is found in semiconductor devices and may be related to imperfections in semiconductor materials and heavy ion implants. It occurs at rates less than 100 Hz.
- *Avalanche* noise is found in Zener diodes and occurs when a PN junction is in reverse breakdown mode. In such cases, the reversal of the electric field in the junction's depletion region allows electrons to develop sufficient kinetic energy to collide with the crystal lattice's atoms and thereby create additional electron-hole pairs. Avalanche noise sources are sometimes used as a "white noise" sources for testing filters, amplifiers, etc.[32]
- *1/f Noise* origin is unclear although it is known to be ubiquitous and that in many situations the transition between so-called "white noise"[33] and 1/f noise occurs in the region between 1 to 100 Hz.
- *Shot Noise* is created by current flow as a result of charges crossing a potential barrier such as that of a PN junction and is given by

$$\overline{i_n^2} = \overline{(i - i_D)^2} = 2\int q\, i_D\, df \tag{6.68}$$

  where q is the charge on an electron[34] and *df* is the frequency differential. Note that shot noise is not a function of temperature and that its value is constant with respect to frequency.

[32] A zener diode operating in avalanche mode is capable of producing 'white noise" up to frequencies as high as several hundred MHz.
[33] White noise is noise which contains equal amounts of noise at all frequencies.
[34] The charge on an electron is $1.62x10^{-19}$ coulomb.

6.3.11 Multiple Noise Sources

Modern electronic devices ~~are~~ contain multiple noise sources and therefore it is important to determine how such noise is to be combined in order to determine the overall noise signal. Noise sources can be internal, external or a combination of both. In addition to internal noise sources interacting with external sources to introduce noise in an embedded system, different parts of an embedded system can interact to produce noise. Although noise is a random process and therefore cannot be predicted, it is possible to predict the noise power, in some cases. Resistors, which are a common elements in operational amplifier implementations, are sources of noise that can, in some cases, be significant concern. An ideal resistor in series with a noise voltage source can be used to model actual resistors.

For example, two resistors independently giving rise to noise can be represented by

$$\overline{e_1^2} = \int 4kTR_1 df \tag{6.69}$$

$$\overline{e_2^2} = \int 4kTR_2 df \tag{6.70}$$

respectively.

If the average mean voltage, $\overline{E_{total}}^2$ is the voltage resulting from the two resistors being connected in series and E_{total} is given by:

$$E_{total} = e_1(t) + e_2(t) \tag{6.71}$$

then

$$\overline{E_{total}(t)^2} = \overline{\left[e_1(t) + e_2(t)\right]^2} = \overline{e_1(t)^2} + \overline{e_2(t)^2} + \overline{2e_1(t)e_2(t)} \tag{6.72}$$

However, in this case e_1 and e_2 are independent noise sources and therefore the average value of the product of $e_1(t)$ and $e_2(t)$ is zero and therefore:

$$\overline{E_{total}(t)^2} = \overline{e_1(t)^2} + \overline{e_2(t)^2} \tag{6.73}$$

Thus the average mean square value of multiple noise sources is the sum of the average mean square value of the noise from each source whether the sources are current or voltage sources.

6.3.12 Signal-to-Noise Ratio

Because signals and noise coexist, it is important to have a measure of the the relative strengths of each, in part, as a way of quantifying how significant noise is in a given system. It is formally defined as the ratio of the signal power to the noise power and frequently measured in dB.

Thus:

$$SNR = \frac{P_{signal}}{P_{noise}} \tag{6.74}$$

and, in terms of dB:

$$SNR_{dB} = 10\, log_{10}\left[\frac{P_{signal}}{P_{noise}}\right] = 20\, log_{10}\left[\frac{v_{signal}^2}{v_{noise}^2}\right] \tag{6.75}$$

6.3.13 Impedance Matching

There are many situations for which it is necessary to consider how much power is being transferred from a source to a sink, i.e., to a load. In some cases it is desirable to deliver as much power from the source to the load as possible. However, in other situations it is important to deliver as little power as possible to the load, or the next stage, if only to avoid loading the previous stage and degrading the signal.

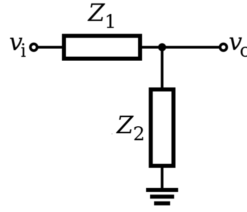


Figure 6.4: Impedance matching example.

Assuming that Z_1 and Z_2 are the source and load impedances, respectively, as show in Figure 6.4 and that

$$Z_1 = R_1 \tag{6.76}$$
$$Z_2 = R_2 \tag{6.77}$$

so that

$$i_i = \frac{v_i}{R_1 + R_2} \tag{6.78}$$

and therefore

$$P = i_i^2 R_2 = \left[\frac{v_i}{(R_1 + R_2)}\right]^2 R_2 \tag{6.79}$$

and setting

$$\frac{dP}{dR_2} = \frac{d}{R_2}\left(\left[\frac{v_i}{(R_1 + R_2)}\right]^2 R_2\right) = 0 \tag{6.80}$$

implies that

$$R_1 = R_2 \tag{6.81}$$

In which case the case second derivative can be shown to be negative and therefore Equation (6.81) must hold, i.e., in order to deliver the maximum power to the load the resistance of the source must be equal to the resistance of the load.

In order to carry out a similar calculation for the case in which the source and the load have both resistive and reactive components, refer again to Figure 6.4. The magnitude of the current passing through Z_1 and Z_2 is given by

$$| i_i |= \frac{| v_i |}{| Z_1 + Z_2 |} \tag{6.82}$$

and power dissipated in Z_2, i.e., the power dissipated in the resistive component of R_{Z2} is given by

$$P = i_{RMS}^2 R_2 = \left[\sqrt{\frac{1}{2\pi} \int_0^{2\pi} I^2 \sin^2(\omega t) dt} \right]^2 = \left[\frac{I}{\sqrt{2}} \right]^2 R_2 \tag{6.83}$$

and therefore

$$P = \frac{1}{2} \left[\frac{| v_i |}{| Z_1 + Z_{2|}} \right]^2 R_{Z2} = \frac{1}{2} \frac{| v_i |^2}{| Z_1 + Z_2 |^2} R_{Z2} = \frac{1}{2} \left[\frac{| v_i |^2}{(R_1 + R_2)^2 + (X_1 + X_2)^2} \right] R_{Z2} \tag{6.84}$$

Again setting the derivative of power P with respect to Z_2 to zero yields the result that

$$R_1 + X_1 = R_2 - X_2 \tag{6.85}$$

and therefore

$$R_1 = R_2 \tag{6.86}$$

$$X_2 = -X_2 \tag{6.87}$$

which is equivalent to requiring that Z_1 be the complex conjugate of Z_2.

6.4 OpAmps and Feedback

As discussed in Chapter 1, the generalized SISO system can be represented as shown in Figure 6.5. Positive feedback is less frequently employed with operational amplifiers because feeding back a positive signal can cause the amplifier to saturate. Negative feedback[35], however, is widely used with operational amplifiers in a a variety of contexts and with a wide range of important and useful results.

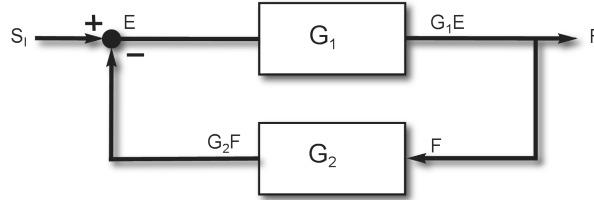


Figure 6.5: A generalized SISO feedback system.

In general,

$$\frac{f(t)}{s(t)} = \frac{G_1}{1 + G_1 G_2} \tag{6.88}$$

and in the present case

$$\frac{v_o}{v_i} = -\frac{A_0}{(1 - \beta A_0)} = \text{Closed Loop Gain} = A_f \tag{6.89}$$

where A_0 is the open loop gain of the amplifier, β is the feedback coefficient and βA_0 is the loop gain. If $\beta A_0 >> 1$, then $A_f \approx 1$ and if $\beta A_0 << 1$, then $A_f \approx A_0$. In the event that $\beta A_0 \approx 1$, the system can be expected to become unstable and oscillation may occur.

[35] Feedback amplifiers began to appear as early as the 1920's, as a result of the efforts of Harold S. Black, a Western Electric engineer interested in developing better repeater amplifiers.[5][6][17]

6.5 Operational Amplifier Incarnations

6.5.1 The Ideal Operational Amplifier

The so-called "Ideal OpAmp"[36] is assumed to have the following characteristics:

- Infinite input impedance, regardless of the amplitude, or frequency, of the input signal, i.e. input current to both inputs is zero.
- Zero output impedance, regardless of the output frequency.
- Infinite "open-loop"[37] gain, where gain is defined as the ratio of output voltage to input voltage.
- Zero input offset.[38]
- An infinite slew rate.[39]
- Introduces zero degrees of variation from a 180° of phase shift from input to output, as a function of frequency.
- No nonlinear effects at any frequency.
- No noise at any frequency.
- Output power is delivered to the load without internal loss within the OpAmp.

It is helpful to think of the ideal OpAmp in terms of the following five rules:

1. "For any output voltage in the linear operating region of an OpAmp with negative feedback, the inputs are at virtually the same potential." [17]
2. No current enters either of the OpAmp's input terminals.
3. KCL[40] is to be applied liberally in analyzing various configurations of an OpAmp.
4. Input voltages times their respective closed loop gains, add algebraically at the output.
5. Voltages applied to either input are multiplied by the non-inverting gain.

6.5.2 Non-Ideal Operational Amplifiers

While the discussion in this chapter has thus far focused on ideal operational amplifiers, it is important to consider the characteristics of actual operational amplifiers in order to appreciate in what manner, and to what extent, they deviate from their idealized counterpart. Although ideal operational amplifiers do not exist, in many cases they can be sufficiently approximated by real world devices. The reality is that commercially available operational amplifiers often vary significantly from the ideal operational amplifier, e.g., the input impedance is not infinite but typically in the M Ω range, open loop voltage gain ranges from 100K to 1M+, etc. A brief review of the comparisons between ideal OpAmps and real OpAmps follows.

[36] While such an ideal device does not actually exist, OpAmps are available with input impedances as high as $10^6\Omega$ for bipolar devices and $10^{12}\,\Omega$ for FET (Field Effect Transistor) devices, gains as high as 10^9, output impedances as low as 100Ω and a gain-bandwidth product of 20 MHz.

[37] Open-loop gain is the amplifier's gain without feedback.

[38] This implies that when the input is zero volts, the out is also zero volts.

[39] Slew rate is defined as the maximum rate of change with respect to time of the output voltage for all possible input voltages, typically in terms of volts/μsecond. This upper limit is caused, in the case of operational amplifiers, by limitations of charge and discharge rates of capacitors within the amplifier.

[40] KCL refers to Kirchoff's Current law which states the sum of all currents into the node of a circuit must equal the sum of all currents out of the node.

- Input Impedance - The input impedance of an OpAmp is characterized by two parameters: 1) common mode impedance and 2) differential impedance. The former is the impedance of each of the inputs with respect to ground and the latter refers to the impedance between the two inputs. An ideal amplifier is assumed to have infinite input impedance. Real OpAmps have finite input impedances, although in some cases the impedance can be as high as 10^{14} Ω.
- Output Impedance - The output impedance of a typical OpAmp is non-zero and nominally 100 Ω.
- Input Current depends on the type of OpAmp input stage. For those with JFET or MOS, the input current can be in the range of 1-10 pA. While this represents relatively small current, in the presence of large impedances, significant voltages can arise. In most cases the currents involved are different for the inputs which can give rise to an *offset voltage* as defined below.
- Gain - While the open loop[41] gain is assumed to be infinite for the ideal OpAmp, in reality open-loop DC gains vary from 100,00 to 1,000,000+. For many applications employing negative feed, this range of gain can be quite acceptable. When real OpAmps are used with negative feedback[42] as is shown later in the chapter the closed loop gain is a function of the amount of feedback employed.
- Offset voltage - Because the transistors in an operational amplifier are not actually identical, grounding the inputs does not assure that the output will be zero. The input bias currents associated with each of the inputs can be assumed to be different for each input. The offset voltage is by definition the input that is required to provide an output of zero volts.
- Slewing is the rate of change of the output is not infinite, as has been assumed for the ideal operational amplifier, due in part to capacitances within the OpAmp. Slew rates of 5 volts/microsecond and higher are typical.
- Saturation - Dynamic range is often important when employing OpAmps, and therefore, the closer the output can be to the rails the better. However, it is possible to drive the output into "saturation" if the gain is set sufficiently high to cause an output that attempts to exceed the supply voltage as shown in Figure 6.6.
- The power supply rejection ratio is defined as:

$$PSRR = \frac{\Delta V_{ps}}{\Delta v_o} \tag{6.90}$$

  and is a measure of the effects of power supply voltage variations, including noise in the OpAmp's output. (The parameters ΔV_{ps} and Δv_o are expressed as RMS values.)

- Power dissipation - There are no intrinsic power limitations associated with ideal OpAmps. However, solid state devices by their nature, are inherently power, current and voltage limited. A real OpAmp is generally limited to output currents that do not exceed 25 mA and maximum voltages of $\pm$ 15 volts.

6.5.3 Inverting Amplifiers

An ideal inverting amplifier has the following transfer characteristic:

$$\frac{V_{out}}{V_{in}} = -A \tag{6.91}$$

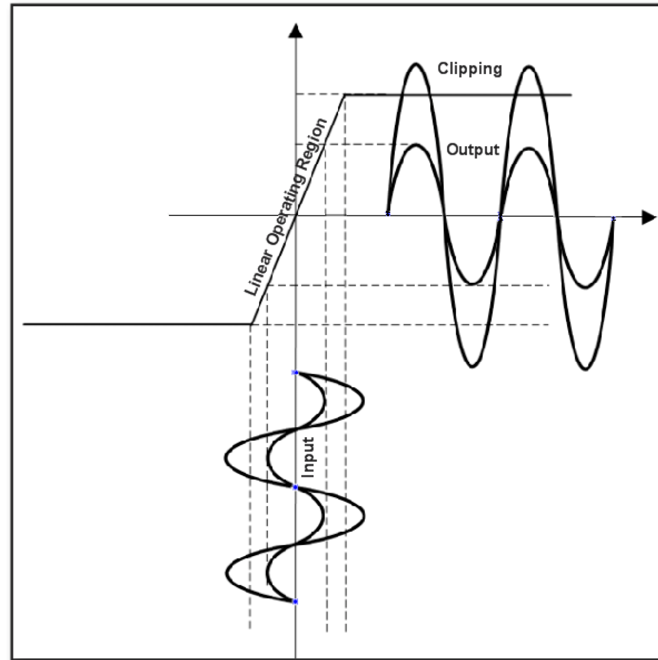


Figure 6.6: An example of clipping.

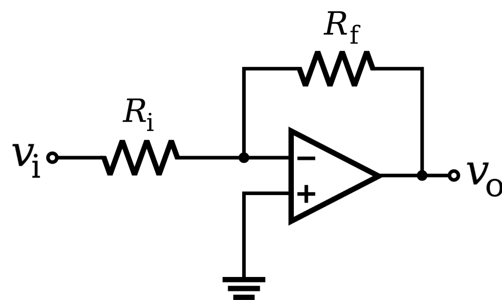


Figure 6.7: Inverting amplifier configuration.

Figure 6.7 shows the configuration of an inverting amplifier and since:

$$i_i = \frac{V_{in}}{R_i} = i_f = -\frac{V_o}{R_f} \tag{6.92}$$

$$V_o = -\frac{R_f}{R_i} V_i = AV_i \Rightarrow A = -\frac{V_o}{V_i} = \frac{R_f}{R_i} \tag{6.93}$$

for the ideal OpAmp.

6.5.4 Miller Effect

Operational amplifiers employing negative feedback are subject to a phenomenon first discovered with vacuum tubes known as the "Miller Effect" which arose because of unintended capacitive coupling between the input and the output. In the case of operational amplifiers this effect can significantly reduce their performance at high frequencies.[29]

As shown in Figure 6.8, given an operational amplifier with a gain of A,[43] the input current is given by:

$$i = \frac{v_i - v_o}{Z_1} = \frac{v_i - Av_i}{Z_1} = v_i \left[\frac{1-A}{Z_1}\right] \tag{6.94}$$

and because the input impedance is given by:

$$Z_i = \frac{v_i}{i_i} \tag{6.95}$$

it follows that:

$$Z_i = \frac{Z_1}{1-A} \tag{6.96}$$

Therefore, if Z_1 is a capacitor the effective input capacitance is increased by a factor of $1 - A$, and if Z is an inductor, or a resistor, it is reduced by that same factor.

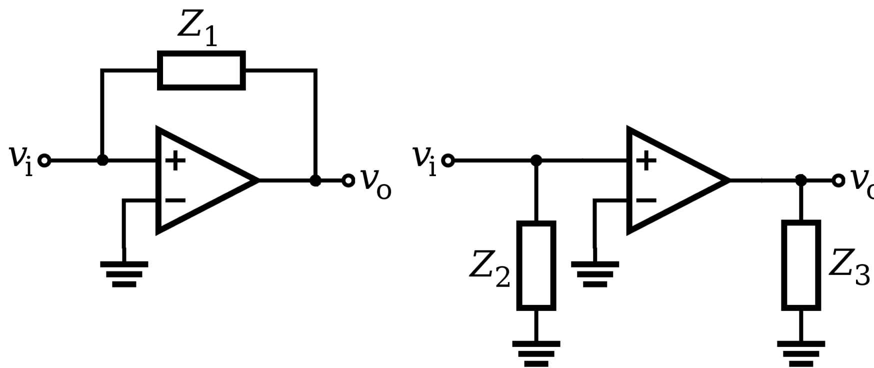


Figure 6.8: Miller effect.

[41] Open loop gain implies gain in the absence of any feedback.
[42] Positive feedback can cause the output to saturate, i.e., to be driven out of the linear range.
[43] Note that in most cases A < 0, i.e., it is negative.

If the feedback impedance Z_1 is replaced by Z_2 and Z_3 as shown in Figure 6.8 , then

$$Z_2 = Z_3\left[1 - \frac{v_o}{v_i}\right] = Z_3(1 - A) \tag{6.97}$$

$$Z_3 = \frac{AZ_1}{A-1} \approx Z_1 \tag{6.98}$$

Thus the Miller effect reflects the fact that parasitic capacitances can be viewed as equivalent to the presence of unintended input and output capacitances, i.e., Z_2 and Z_3, respectively, as shown in Figure 6.8. A technique known as “compensation” is used in some cases to minimize the adverse effects of the Miller Effect. Replacing Z_1 with Z_2 and Z_3 means that the current through Z_1, Z_2 and Z_3 must be the same. If the gain A is sufficiently large, the input capacitance can function as a short and thus block the input signal.

Finally,

$$Z_3(v_i - v_o) = Z_2 v_i \tag{6.99}$$

$$j\omega C_{out}(v_i - v_o) = j\omega C_{in} v_i \tag{6.100}$$

$$C_{out}\left[1 - \frac{v_o}{v_i}\right] = C_{in} \tag{6.101}$$

$$C_{in} = [1 - A]C_{out} \tag{6.102}$$

6.5.5 Noninverting Amplifier

For the case of the noninverting amplifier configured as shown in Figure 6.9 and

$$\frac{v_o}{v_i} = A \tag{6.103}$$

and,

$$v_0 = \left[\frac{R_1 + R_2}{R_1}\right] v_i \Rightarrow A = 1 + \frac{R_2}{R_1} \tag{6.104}$$

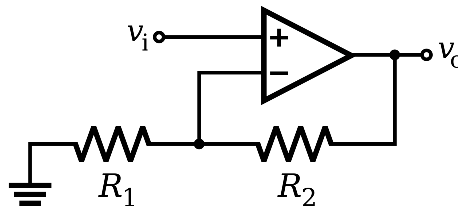


Figure 6.9: A noninverting amplifier.

6.5.6 Summing Amplifier

Similarly, an ideal weighted[44] summing, or “adder”, amplifier is based on an operational amplifier configured as shown in Figure 6.10.

[44] Resistor values can be selected to combine signals of different amplitudes.

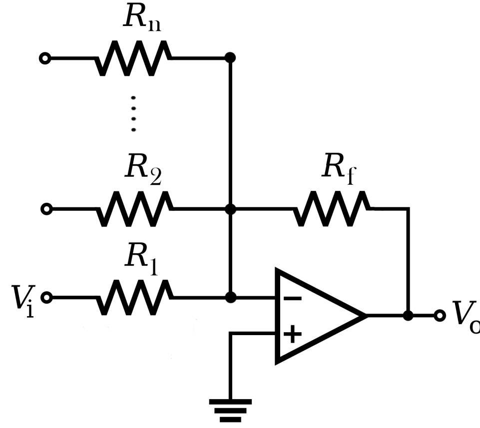


Figure 6.10: Summing amplifier.

Because

$$v_o = \left[\frac{v_1}{R_1} + \frac{v_2}{R_2} + \cdots + \frac{v_n}{R_n}\right] R_f = \left[\frac{R_f v_1}{R_1} + \frac{R_f v_2}{R_2} + \cdots + \frac{R_f v_n}{R_n}\right] \tag{6.105}$$

and therefore,

$$V_o = \left[A_1 v_1 + A_2 v_2 + \cdots + A_n v_n\right] \tag{6.106}$$

and if,

$$R = R_1 = R_2 = \cdots = R_n \Rightarrow A = A_1 = A_2 = \cdots = A_n \tag{6.107}$$

Equation (6.106) becomes

$$v_o = A\left[v_1 + v_2 + \ldots + v_n\right] \tag{6.108}$$

6.5.7 Difference Amplifier

In this example, as shown in Figure 6.11, the fact that the difference amplifier circuit is linear allows superposition to be imposed so that by inspection, and referring to Equation (6.104), it follows that:

$$v_o = A_2 v_{i2} - A_1 v_{i2} = \left[\frac{1 + \frac{R_1}{R_2}}{1} + \frac{R_3}{R_4}\right] v_{i2} - \left[\frac{R_2}{R_1}\right] v_{i1} \tag{6.109}$$

and if

$$R_1 = R_2 = R_3 = R_4 \tag{6.110}$$

then,

$$v_o = v_{i2} - v_{i1} \tag{6.111}$$

and the difference amplifier is referred to as a “differential amplifier”.

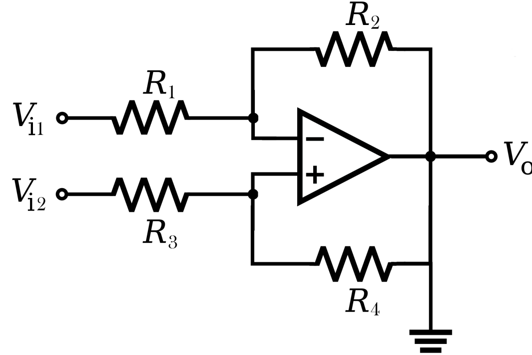


Figure 6.11: Difference amplifier configuration.

6.5.8 Logarithmic Amplifier

When dealing with a signal that has a large dynamic range it can sometimes be difficult to keep high levels of the signal from masking the lower levels. One technique for addressing this problem is to use a logarithmic amplifier configuration, as shown in Figure 6.12, to effectively expand the

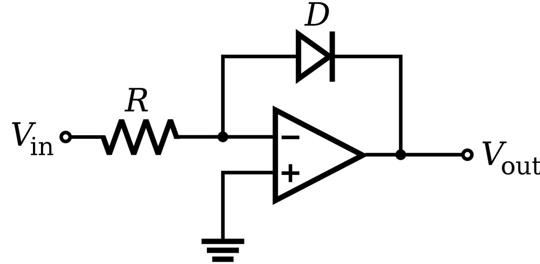


Figure 6.12: An example of a logarithmic amplifier.

lower levels of a signal and compress the higher levels so that both fall into a detectable range that can best be handled by the embedded system. The current through a diode is well known to be given by:

$$i_d = i_s\left[e^{\frac{v_d}{(nV_T)}} - 1\right] \approx I_s e^{\frac{v_d}{nV_T}} \tag{6.112}$$

where v_d is the voltage across the diode, i_s is the reverse bias saturation current, V_T is the "thermal voltage"[45] and the so-called ideality factor is assumed to be equal to one, in most cases.

Therefore,

$$i_d = \frac{v_i}{R} = \frac{i_s e^{\frac{v_d}{V_T}}}{R} \tag{6.113}$$

can be expressed as

$$v_o = -V_T ln\left[\frac{v_i}{i_s R}\right] \Rightarrow v_d = -V_T ln[v_i] - constant \tag{6.114}$$

[45]The thermal voltage is given by kT/q where T is the absolute temperature of the diode's PN junction, q is the charge on an electron and k is the Boltzmann constant that at room temperature is $\approx 25mV$.

where *constant* $= i_s R$.

6.5.9 Exponential Amplifier

As shown in Figure 6.13, an exponential amplifier can be configured by placing a diode at the input to the OpAmp with Equation (6.112) representing the input current and therefore:

$$v_o = i_d R = i_s \left[e^{\frac{v_d}{(nV_T)}} - 1 \right] R \approx I_s R e^{\frac{v_d}{nV_T}} = \alpha R e^{\beta v_i} \tag{6.115}$$

where $\alpha = I_s$ and $\beta = \frac{1}{nV_T}$.

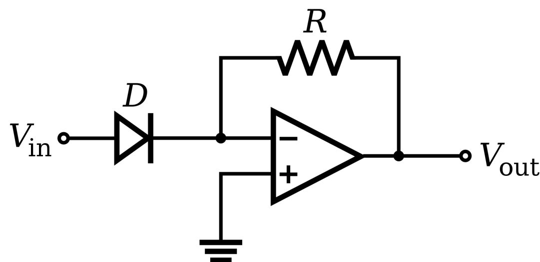


Figure 6.13: An exponential amplifier (e^{v_i}).

6.5.10 OpAmp Integrator

One of the most common configurations of OpAmps, at least historically, has been as an integrator as illustrated in Figure 6.14.

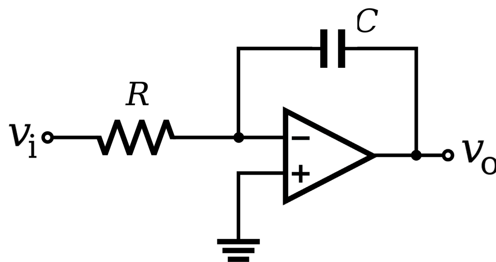


Figure 6.14: An OpAmp configured as an integrator.

The current into the circuit is given by

$$i = \frac{v_i}{R_i} = i_C = -C\frac{dv_0(t)}{dt} \Rightarrow \frac{dv_0(t)}{dt} = -\frac{v_i}{RC} \tag{6.116}$$

which is a first order, linear differential equation whose general solution can be expressed as

$$v_0(t) = -\frac{1}{RC}\int_0^t v_i dt + constant \tag{6.117}$$

where the constant refers to the voltage on the capacitor at the start of the integration cycle, i.e.,

t=0. It should be noted that in some applications it is necessary to reset the integrator, typically by shorting the integrating capacitor, as for example in the case of a constant input voltage whose application is significantly greater than the RC time constant, because the time integral of a constant integrator is a linear function of time which could ultimately lead to saturation of the integrator. One application for this type of circuit has been in implementing the dual-slope, analog-to-digital converter.

6.5.11 Differentiator

An OpAmp can also be configured as a differentiator by using a resistor for feedback and a capacitor for input as shown in Figure 6.15.

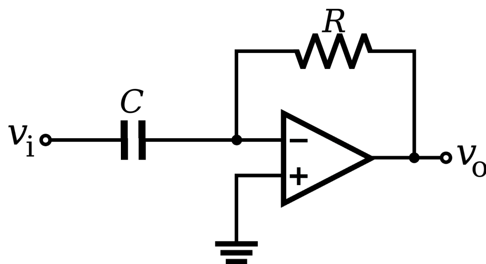


Figure 6.15: An idealized differentiator.

The amount of charge, q, stored in a capacitor is given by:

$$q = CV \tag{6.118}$$

Thus the output voltage[46] is given by:

$$i_i = \frac{dq}{dt} = C\frac{dv_i}{dt} = -\frac{v_o}{R} \tag{6.119}$$

and therefore,

$$v_o = -RC\frac{dv_i}{dt} \tag{6.120}$$

Unfortunately, differentiators tend to amplify high frequency noise and therefore represent perhaps the least used configuration of OpAmps. In some applications a resistor is placed in series with the input capacitor in order to limit the gain (R_f/R_i) of higher frequency components, while still allowing the low frequency gain to be determined by the capacitor and feedback resistor. However the cutoff frequency is subsequently determined by:

$$f_{cut\ off} = \frac{1}{2\pi R_i C} \tag{6.121}$$

6.5.12 Instrumentation Amplifiers

The so-called instrumentation amplifier, one configuration of which is show in Figure 6.16, is used in applications for which a small differential signal, often in the presence of a strong common mode signal, must be measured. Instrumentation amplifiers are designed to ignore the common mode

[46]Note that the positive input terminal is grounded and therefore the negative input terminal can be expected to be a ground also.

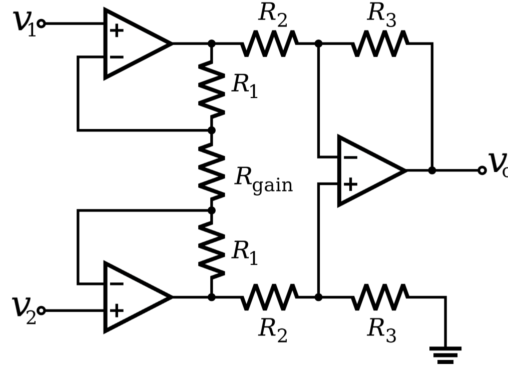


Figure 6.16: A classical instrumentation amplifier configuration.

signal while amplifying the differential input signal. Furthermore, such signals are often provided by sources of relatively low input impedance. This circuit provides very high input impedance that assures that the input signal is not subjected to an impedance that will degrade the input signal. As shown, this particular configuration consists of a differential input amplifier followed by a difference amplifier both of which are discussed in this chapter. The former providing very high input impedance and common mode rejection while the latter provides single ended output.

$$v_{o1} = v_{i2} + (v_2 - v_1)\left[1 + \frac{R_1}{2R_2}\right] \tag{6.122}$$

$$v_{on} = v_2 + (v_1 - v_2)\left[1 + \frac{R_1}{2R_2}\right] \tag{6.123}$$

$$v_{op} - v_{on} = (v_2 - v_1)\left[1 + \frac{R_1}{2R_2}\right] \tag{6.124}$$

$$v_{o2} = v_{cm} + \frac{v_d}{2}\left[1 + \frac{R_1}{2R_2}\right] \tag{6.125}$$

$$v_{o1} = v_{cm} - \frac{v_d}{2}\left[1 + \frac{R_1}{2R_2}\right] \tag{6.126}$$

and,

$$v_o = [v_{o1} - v_{o2}]\frac{R_4}{R_3} \tag{6.127}$$

$$v_o = v_d\left[1 + \frac{R_1}{2R_2}\right]\frac{R_4}{R_3} \tag{6.128}$$

In the sections that follow, discussion will focus on various configurations of a fundamental building blocks used in both PSoC3 and PSoC5, unless otherwise noted, that are referred to as the switched-capacitor and continuous-time (SC/CT) blocks.

6.5.13 Transimpedance Amplifier (TIA)

Embedded systems often make use of a variety of sensors some of which supply currents that are proportional to the parameter being sensed, e.g., photodetectors, photomultipliers, etc. Similarly, some external peripheral devices require current for input. As a result there is a need for interfaces that convert current-to-voltage and/or voltage-to-current. OpAmps can be very useful when configured as shown in Figure 6.17. A specific example involving a photodetector is shown[47] in

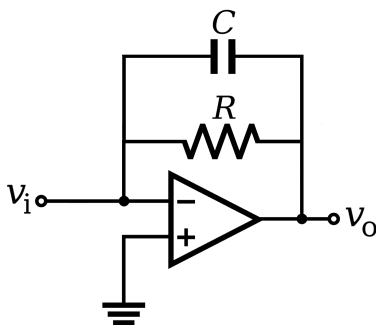


Figure 6.17: A generic TIA.

Figure 6.18. The output of the transimpedance amplifier is a voltage proportional to the current flowing in the photodiode as the result of radiation detected from a laser.

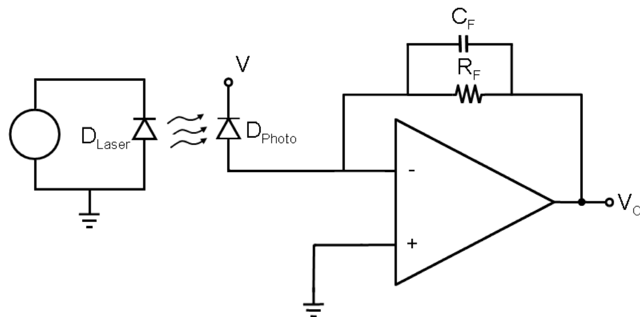


Figure 6.18: A typical application of TIA and photodetector.

The output voltage of this configuration is given by:

$$v_{out} = -i_{photo}R_f \tag{6.129}$$

A small capacitor, C_F, is sometimes used to assure that the transimpedance amplifier remains stable.

6.5.14 Analog Comparators

It has been suggested that the comparator is the fundamental building block of mixed-signal design[45]. Comparators compare the differential voltage resulting from the voltages applied to both inputs and produce an output voltage that has the same sign and a magnitude as that of one

[47] It should be noted that the actual circuit would be subject to additional capacitances that are ignored in this discussion, viz, a capacitance introduced by the photodiode and the OpAmp's common mode capacitance.

of the supply voltages. Analog comparators are basically differential amplifiers with extremely high open loop gain and high slew rates[48]. They can be employed to compare one analog signal to another, e.g., a reference voltage, in terms of sign and magnitude. This is particularly useful for applications requiring the monitoring of various types of threshold which can be expressed as a voltage level, e.g., light levels, temperatures, fluid levels, etc., particularly in situations which require a rapid response to levels reaching some predefined threshold.

Generic OpAmps can be used for this purpose but have some potentially serious limitations in such applications, e.g., slower slew rates than obtainable with devices designed specifically to function solely as comparators. Also, comparators are designed to work with large differential inputs which OpAmps are not. Although OpAmps have high input impedance and draw very little current they can be damaged by large differential voltages and their input characteristics in terms of impedance and input (bias) current, can depart significantly from their otherwise normal values when inputs exceed a few hundred millivolts. The idealized form of a comparator is shown in Figure 6.19.

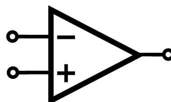


Figure 6.19: An ideal comparator.

Because comparators are intended to function as nonlinear devices, they are not used with negative input feedback in order to avoid degrading their switching speeds. When used with positive feedback the comparator functions as a bistable[49] device. While similar to operational amplifiers in that idealized comparators are assumed to have infinite gain, require no input current, and have zero offset/bandwidth, unlike their OpAmp counterpart, comparators are designed to saturate and recover quickly, are not compensated, operate either in an open loop mode or with positive feedback and typically have open collector, open drain or open emitter outputs. Although an analog device, it functions as an analog input device with a "digital", i.e., binary, output.

If the voltage applied to the positive input terminal is greater than that applied to the negative input terminal, then the output voltage rapidly increases until it reaches the positive rail voltage[50]. Similarly, if the voltage applied to the positive terminal is less than that applied to the negative terminal, the output rapidly becomes equal to the negative supply voltage. But there is a potential problem, viz., what happens when the two inputs are such that difference is "zero"[51] or perhaps more importantly when the difference between the two inputs is approximately zero.

If the inputs are such that

$$| V_+ - V_n | < \epsilon \tag{6.130}$$

for sufficiently small ϵ, noise can cause a transition, or multiple undesirable transitions to occur,

[48] Slew rates for comparators are usually expressed in terms of propagation delays.

[49] That is there are only two stable states, the output is either at the positive rail potential or the negative rail potential.

[50] The phrase "positive rail voltage is a term of art that refers to the positive supply voltage level. Most OpAmps operate between either positive and negative supply voltages or the equivalent by employing so-called "virtual grounds". In each case, the effective positive and negative supply voltages are referred to respectively as the positive and negative rails. If an OpAmp's output is "on one of the rails" meaning either at the positive or negative supply voltage, the output is said to be "saturated'.'

[51] Zero volts in actual analog circuits is a topic in and of itself and shall not be treated in detail in this textbook except to note that in practice designs relying on a potential of precisely zero for functioning should be avoided.

and, in addition, the comparator may begin to function as a linear device with respect to output. This condition allows noise to be transmitted from the input to the output and therefore to devices external to the comparator. The problem also arises with OpAmps because of the inherent difficulty in establishing and/or maintaining either an absolute, or differential, value of zero volts.

Ideally the comparator should function in a manner that assures that when the differential voltage between the input terminals crosses zero the output state changes. Adding hysteresis establishes not one but two trigger points, $V_{+switch}$ and $V_{-switch}$, for a change of state.

In such cases, the hysteresis voltage is defined by:

$$V_{+switch} - V_{-switch} = V_{hysteresis} \tag{6.131}$$

If an offset voltage is present it becomes the mean value of $V_{+switch}$ and $V_{-switch}$ and not zero. Unfortunately, the offset voltage is a function of both the supply voltages and the temperature. However, using positive feed back can improve the situation substantially as is shown in the next section.

6.5.15 Schmidt Triggers

As discussed in the previous section, one of the challenges presented by comparators is their behavior near threshold. As the voltage level approaches a threshold value, noise can cause a transition to occur prematurely. Perhaps worse is the possible that noise could cause multiple premature transitions near threshold. The addition of hysteresis is one way to minimize this effect. The technique is to feed back some of the output signal to the positive input. Schmitt triggers are special cases of comparators that are typically used to improve pulse shape and as a way of generating very fast rise/fall time pulses. Pulses tend to degrade over time and the Schmitt trigger has been commonly employed to "sharpen-up"[52] such degraded pulses. The Schmitt trigger[40] was invented by Otto H. Schmitt[53], circa 1937, in part to study squid nerves and has the interesting property of limited memory of prior events in the form of hysteresis which is a property exhibited by a variety of systems, e.g., those involving magnetic materials. Two bistable configurations of the Schmitt trigger are shown in Figures 6.20 and 6.21. The positive input to the non-inverting Schmitt trigger can be derived by noting that:

$$v_+ = (v_o - v_i)\left[\frac{R_1}{R_1 + R_2}\right] + v_i = \left[\frac{R_1}{R_1 + R_2}\right]v_o + \left[\frac{R_2}{R_1 + R_2}\right]v_i \tag{6.132}$$

and therefore:

$$v_+ \approx \frac{R_1}{R_1 + R_2}v_o \tag{6.133}$$

which represents the amount of hysteresis. If the comparator is in a "saturated" state, for which the output voltage is equal to the "positive rail", e.g., +15 vdc and $R_2 = 14R_1$, then the hysteresis is ±1 vdc.

[52]Perhaps an unfortunate use of the language, but the basic idea is to restore the pulse to faster fast rise and/or fall times, referred to by some as "squaring-up" the pulse.

[53]Schmitt described his invention as a simple hard valve circuit, ie.e. vacuum tube, which provides positive off-on control with any differential from 0.1 v to 20 volt while requiring less than a microampere to do so. Transition time was approximately 10μseconds

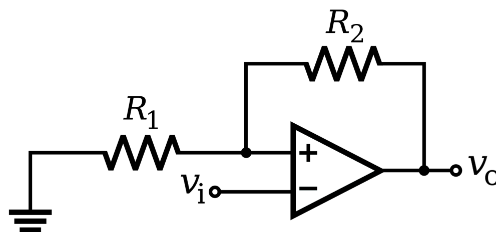


Figure 6.20: Inverting Schmitt trigger.

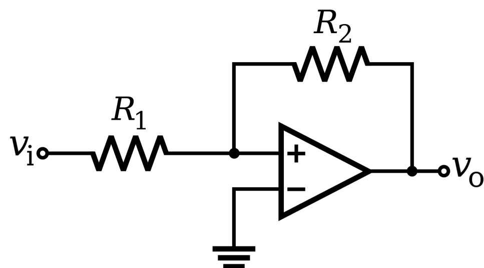


Figure 6.21: Non-Inverting Schmitt trigger.

6.6 Switched-Capacitor Blocks

Switched-capacitor modules are based on a very simple and fundamental concept that allows resistors to be replaced with capacitor-switch combinations that function as resistors. This technique was developed in part as a result of the difficulty of fabricating precision resistor values at the chip level. Switches, capacitors and operational amplifiers have proven relatively easy to manufacture at the chip level. Combinations of switches and capacitors are an attractive alternative to resistors, particularly in light of the fact that in such applications capacitor which is desirable in that such capacitors track with temperature variances.

The fundamental concept involved in switched-capacitors is illustrated in Figure 6.22. A

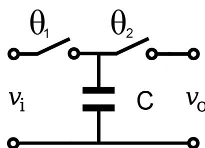


Figure 6.22: The basic switched-capacitor configuration.

capacitor is alternately connected to ground and/or an input/output voltage connections v_1 and v_2 by the clocks Θ_1 and Θ_2. With Switch 2 open and Switch 2 closed, the charge Cv_i is transferred to the capacitor. Switch 1 then opens and switch 2 closes allowing the charge, Cv_o to be transferred to the load.

Thus a net charge,

$$\Delta q = C(v_o - v_i) \tag{6.134}$$

is transferred during each cycle of period T_s.

The operation of these switches is subject to the following requirements:

1. the Θ_1 and Θ_2 switches must not be closed at the same time,
2. the Θ_1 switch must be open before the Θ_2 switch closes,
3. the Θ_2 switch must be open before the Θ_1 switch closes,

and finally,

4. the frequency, f_s, used for switching must allow the capacitor to fully charge, and discharge during each cycle.[54]

By definition:

$$i = \frac{\Delta q}{\Delta t} \tag{6.135}$$

and,

$$R = \frac{v}{i} \Rightarrow R = \frac{v\Delta t}{\Delta q} = \frac{v\Delta t}{vC} = \frac{T_s}{C} = \frac{1}{f_s C} \tag{6.136}$$

it should be noted therefore, that the ratio of two resistances is simply the inverse ratio of their corresponding capacitive equivalents, i.e.,

$$\frac{R_1}{R_2} = \frac{\frac{1}{f_s C_1}}{\frac{1}{f_s C_2}} = \frac{C_2}{C_1} \tag{6.137}$$

and thus the ratio of switch capacitance, is independent of the clocks, and consequently so are the equivalent resistances. Furthermore, while in effect the charge is being delivered in discrete packets by this method, just as charge in a resistor that has been subjected to an applied potential difference is also delivered in the form of quanta, each of which carries a fixed amount of charge, viz., the charge on an electron, $1.6x10^{-19}$ coulombs.

Filters and other analog switched-circuits make frequent use of RC constants which can be implemented using only capacitances and switches. In addition to requiring less real estate than their resistor counterparts, switched-capacitors provide better linearity, closer tolerances ($\pm 1.0\%$), better matching ($\pm 0.1\%$) wider range and allow the RC time constants to be varied by varying the switching frequency. Figure 6.23 shows a switched-capacitor integrator.

However, switched-capacitance inputs and outputs are subject to the same issue as that of any sampled system, viz, “what you find, depends on when you look”.

[54]This constraint is imposed to assure that the value of charge on the capacitor accurately represents the input voltage during that cycle.

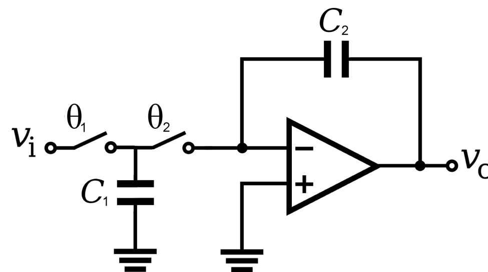


Figure 6.23: A switched-capacitor integrator.

6.7 Switched-Capacitor and Continuous-Time Devices

The PSoC3/5 switched-capacitor (SC) and continuous-time (CT) module shown in Figure 6.24 is a general purpose, highly optimized block that can be configured as a:

- CT, unity gain, amplifier,
- CT, programmable gain amplifier,
- CT transimpedance amplifier,
- CT mixer,
- Delta Sigma modulator,
- Operational amplifier,
- Sampled Mixer,

or a

- Track and hold amplifier,

with programmable power and bandwidth, routability to GPIO, routable reference selection and sample and hold capability.

Table 6.3: Register-Selectable Operational Modes.

SC_MODE[2:0]	Operational Mode
[000]	OpAmp
[001]	Transimpedance Amplifier
[010]	CT Mixer
[011]	DT Mixer NRZ S/H
[100]	Unity Gain Buffer
[101]	First-Order Modulator
[110]	Programmable Gain
[111]	Track & Hold Amplifier

However, it should be noted that the SC/CT block in PSoC3/5 differs from that implemented in PSoC1 in that the former has been optimized to carry out the specific functionality referenced above in terms of gain, bandwidth product and slew rate. Therefore implementation of functionality such as integrators, differentiators and filters, using OpAmps and discrete external components, as discussed in previous sections in this chapter, in some cases may be more appropriate or, alternatively, the PSoC1 should be used.[13]

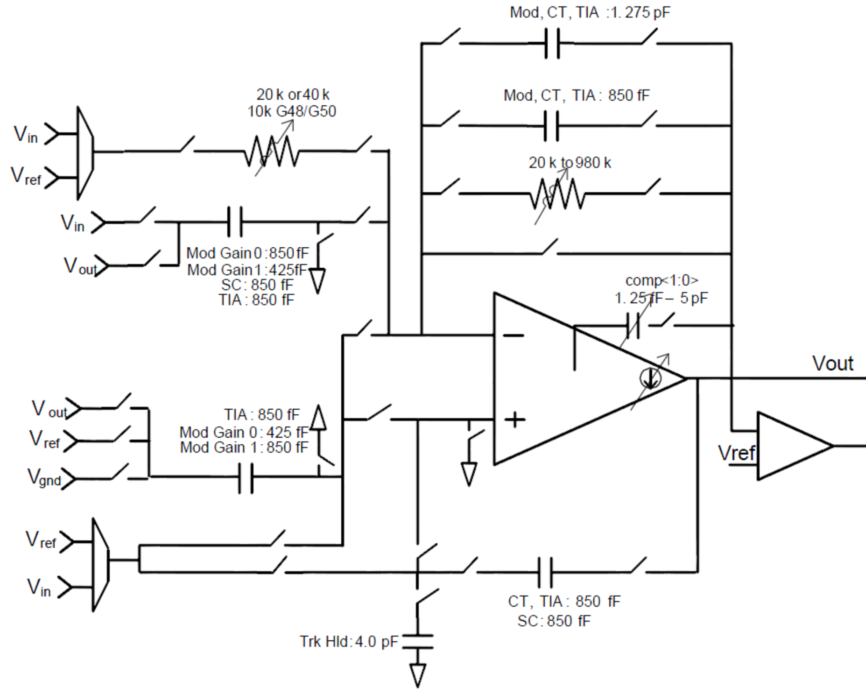


Figure 6.24: Switched-capacitor and continuous-time block diagram.

The simplest configuration of the SW/CT block is the OpAmp. In this mode the internal resistors and capacitors associated with the SW/CT block feed and input terminals are disconnected. This allows external components to be used for input and feedback and the techniques discussed in earlier sections to be employed. This mode can be selected by setting the MODE[2:0] bits in the SC[0...3]_CR0 to 000. The OpAmp is two stage, rail-to-rail amplifier with a folded cascode[55] first stage and Class A[56] second stage which is internally compensated. The value of the compensation capacitor and the drive strength of the output stage are both programmable to accommodate different loading conditions. The appropriate setting is a function of the minimum required slew rate[57] and load capacitance.

The load current is given by:

$$i_{load} = C_{load}\left[\frac{\Delta v}{\Delta t}\right] \tag{6.138}$$

where C_{load} includes both the internal capacitance and the capacitance of the load.

Assuming a value of 10 pF for the internal capacitance, the drive controls SC_DRIVE[1:0] should be set according to the slew requirements at the output in the SC[0..3]_CR1[1:0]register bits.

[55]Cascode refers to a two stage amplifier which consists of a transconductance amplifier and a current buffer. It is capable of providing higher bandwidth, output impedance, input impedance and higher gain than a single stage amplifier while significantly improving input/output isolation since there is no direct coupling between input and output. This configuration is not subject to the Miller effect (cf. Section 6.5.4).

[56]Class A amplifiers produce outputs that are undistorted replications of the input and conduct during the entire input cycle. Alternatively, they are linear amplifiers that are always conducting.

[57]Which is the desired rate of change of the signal with respect to time.

Table 6.4: Miller capacitance between amplifier output and output driver.

SC_COMP[1:0]	CMiller (pF)
00	1.30
01	2.60
10	3.90
11	5.20

Table 6.5: SC/CT block drive control settings.

SC_Drive[1:0]	$i_{load}(\mu A)$
2'b00	280
2'b01	420
2'b10	530
2'b11	650

This OpAmp configuration has three control options for modifying the closed loop bandwidth and stability applicable to all configurations:

1. Current through the first stage of the amplifier (BIAS_CONTROL),
2. Miller capacitance between the amplifier input and output stages (SC_COMP[1:0]),

and,

3. Feed back capacitance between the output stage and the negative input terminal (SC_REDC[1:0]).

The bias control doubles the current through the amplifier stage. The BIAS_CONTROL should be set to 1 to provide greater overall bandwidth once the circuit is stabilized rather than using the option of less current in the first stage. The bias current can be doubled by setting the SC[0..3]_CR2[0] register bit. The SC_COMP bits set the amount of compensation and directly affects the amplifier's gain-bandwidth. The Miller capacitance should be set to one of the four values for the SC[0..3]_CR[3:2] as shown in Table 6.6.

There is also an option related to the capacitance between the output driver and the negative input terminal that affects the stability-capacitance option. This option contributes to a higher frequency zero and a lower frequency pole which reduces the over all bandwidth and provides some additional phase margin at the unity gain frequency, depending on the CT configuration. Table 6.7 shows the available settings.

PSoC3 and PSoC5 each have 4 operational amplifiers configured as shown in Figure 6.25 which have the following features:

- 25 ma drive capability,
- 3 MHz gain-bandwidth into a 200 pF load
- Low noise

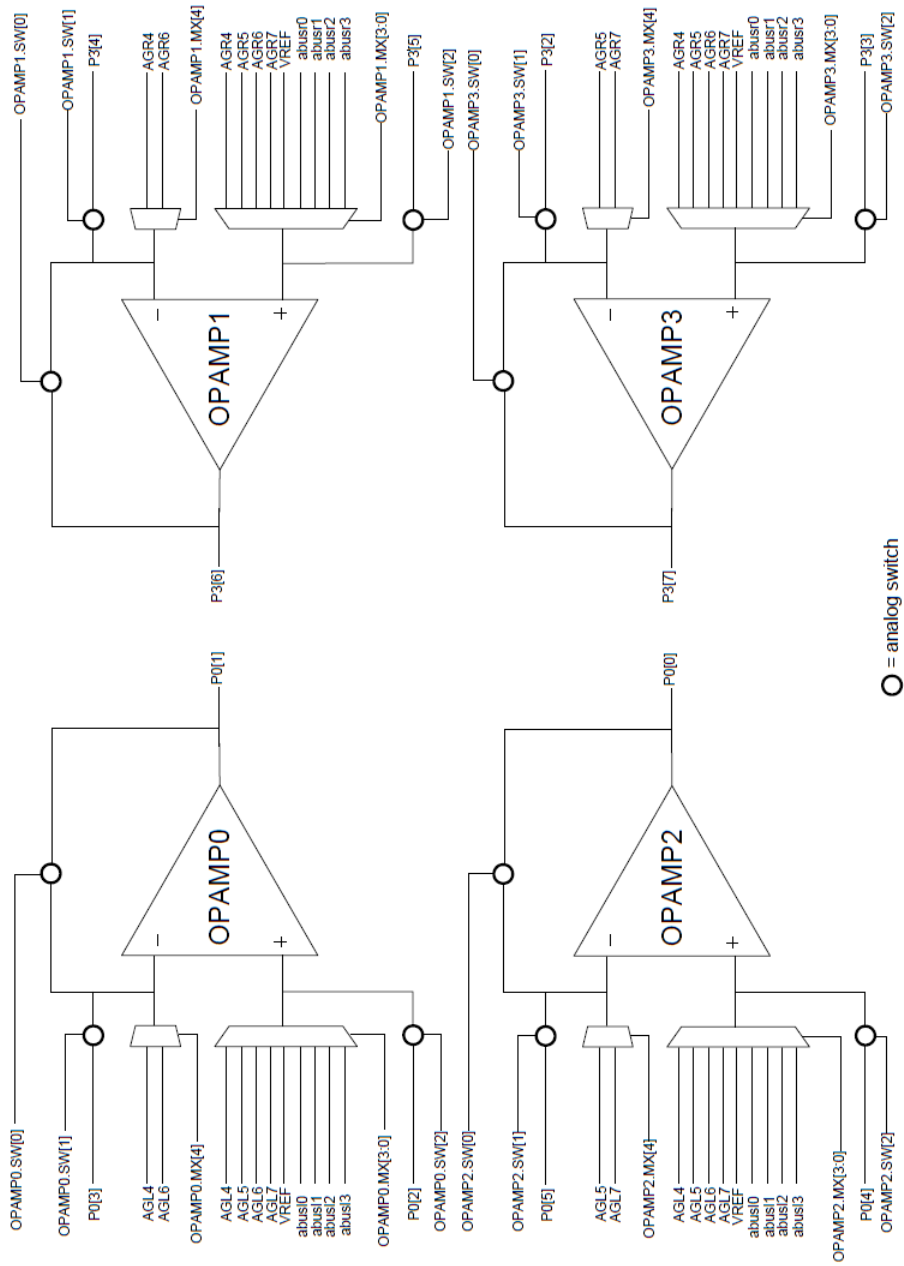


Figure 6.25: PSoC3/5 operational amplifier connections.

Table 6.6: Miller Capacitance between Amplifier Output and Output Driver.

SC_COMP[1:0]	CMiller (pF)
00	1.30
01	2.60
10	3.90
11	5.20

Table 6.7: C_{FB} for CT Mix, PGA, OpAmp, Unity Gain Buffer and T/H Modes.

SC_REDC[1:0]	C_{FB}(pF)
00	0.00
01	1.30
10	0.85
11	2.15

- Less than 5 mV of offset
- Rail-to-rail capability to within:
  1. $50mV$ of V_{ss} or V_{dd} for a 1 mA load.
  2. 500 mv of V_{ss} or V_{dda} for a 25 mA load.
- A slew rate of $3V\mu$s for a 200 pF load.[58]

The OpAmps are configurable either as uncommitted OpAmps, or as unity gain buffers. Access to the negative and positive inputs of the OpAmps is provided by muxes and analog switches. An analog global, local analog bus or reference voltage is connected to an input via a mux. A GPIO is connected to an input via an analog switch.

6.7.1 PSoC3/5 OpAmps and PGAs

PSoC3 provides Operational Amplifiers and Programmable Gain Amplifiers as shown in Figure 6.26. The OpAmp is able to function either in a basic operational amplifier configuration, or as a simple follower. PSoC3/5 OpAmps can also be used by employing available internal resistors, capacitors and multiplexers, or in conjunction with external components, as shown in Figure 6.27. Output current from the OpAmp should not exceed 25 mA and output loads should not exceed $10K\,\Omega$.

The programmable gain amplifer's gain can be set as one of the following values: 1 (default value), 2, 4, 8, 16, 24, 25, 48,or 50) shown in Table 6.8. There four power settings: minimum, low, medium (default value) or high. The power settings affect the PGA's response time with low power resulting in the slowest response time and high power the fastest. Vref_Input is a

[58]Or the equivalent of 3 megavolts/second!

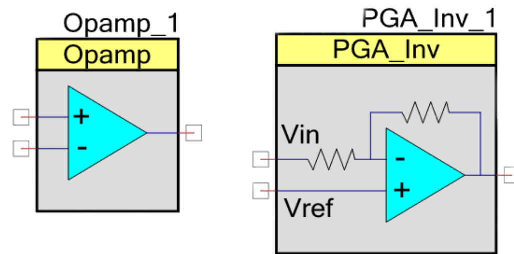


Figure 6.26: PSoC Creator's graphical representation of an OpAmp and a PGA.[37]

Table 6.8: PGA Gain Settings

Gain Setting	Gain Value
PGA_GAIN_01	1
PGA_GAIN_02	2
PGA_GAIN_04	4
PGA_GAIN_08	8
PGA_GAIN_16	16
PGA_GAIN_24	24
PGA_GAIN_25	25
PGA_GAIN_32	24
PGA_GAIN_48	48
PGA_GAIN_50	50

parameter that can be set either as *Internal Vss* which sets

$$V\text{ref_input} = Vss \tag{6.139}$$

i.e., the reference input is set to an internal ground, or as *External* which sets

$$V\text{ref_input} = External \tag{6.140}$$

for cases in which the reference input is to be connected to an arbitrary reference signal.

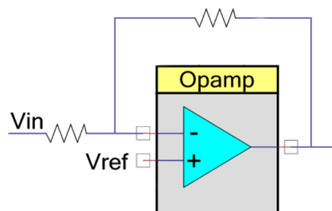


Figure 6.27: PSoC3/5 operational amplifier configured as an inverting, variable gain OpAmp using external components. [37]

```
Example 6.6: Sample C source program for initializing and starting a PGA:

#include <device.h>
void main()
{
PGA_1_Start();
PGA_1_SetGain(PGA_1_GAIN_24);
PGA_1_SetPower(PGA_1_MEDPOWER);
}
```

The PGA is constructed from a generic SC/CT block. The gain is selected by adjusting two resistors, R_a and R_b, that are shown in Figure 6.28. R_a may be set to either 20K or 40K ohms. R_b may be set between 20K and 1000K ohms, to generate the possible gain values selectable in either a parameter dialog in PSoC Creator, or via the *SetGain* function which selects the proper resistor values for the selected gain. The OpAmp component can be configured as either

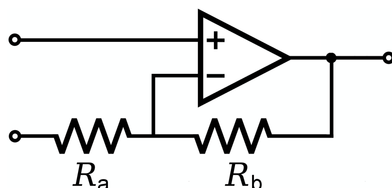


Figure 6.28: PGA internal gain resistors.

a follower or as an OpAmp that can be used in conjunction with external components. It can be used to drive loads that are less than 10K and provide a maximum driving current of 25 ma. When used as a follower, the negative input is inaccessible.

6.7.2 Continuous-Time Unity Gain Buffer

The CT, unity gain buffer is, as shown in Figure 6.29, simply an OpAmp with the inverting input connected to the output and used when an internally generated signal is being used with high output impedance, e.g., a voltage DAC driving a load, or an external, high impedance source impedance driving a significant on-chip load, such as the continuous-time Mixer.

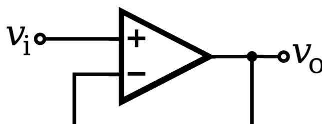


Figure 6.29: An OpAmp configured as a unity gain buffer.

6.7.3 Continuous-Time, Programmable Gain Amplifier

The programmable gain amplifier (PGA) is a continuous-time OpAmp, configured as shown in Figure 6.30, with selectable taps for the input and feedback resistors. It is selectable by setting MODE[2:0] bits in the SC[0,,3]_CR0 register to '110'. The PGA can be implemented as either a positive or negative gain topology, or as half of a differential amplifier. Gain is selected by

setting bit [5] '1'(SC_GAIN) in the SCL[0..3]_CR1 register. If SC_GAIN is set to one, then the configuration is non-inverting with a gain of $(1 + \frac{R_{FB}}{R_{in}})$. If SC_GAIN is set to zero then the configuration is inverting with a gain of $-\frac{R_{FB}}{R_{in}}$. As shown in Figure 6.31 it is possible to create a differential amplifier by connecting two PGAs as shown. A low impedance external resistor R_{LAD} are used to reduce gain error. The differential amplifier's gain is given by

$$v_{o+} - v_{o-} = A(vi+ - v_{i-}) \tag{6.141}$$

and he common mode voltage of the output is also the common voltage input, viz.,

$$VCM = \frac{(v_{i+} - v_{i-})}{2} \tag{6.142}$$

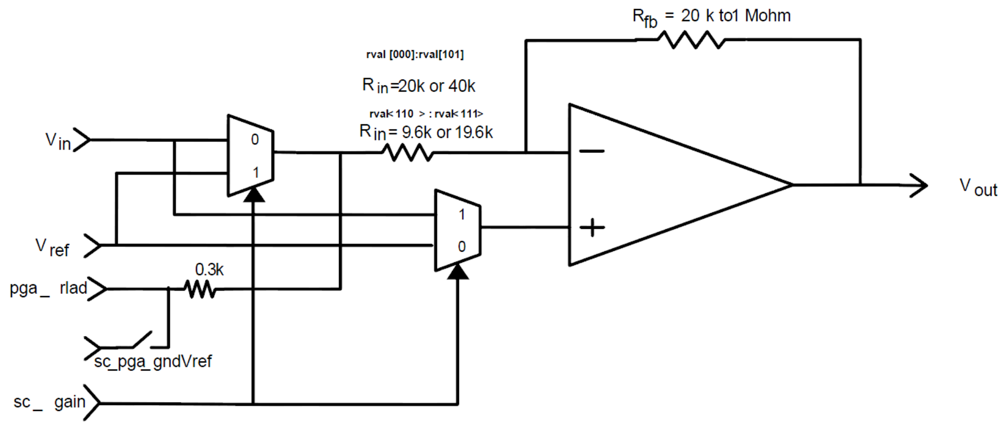


Figure 6.30: CT PGA configuration. [37]

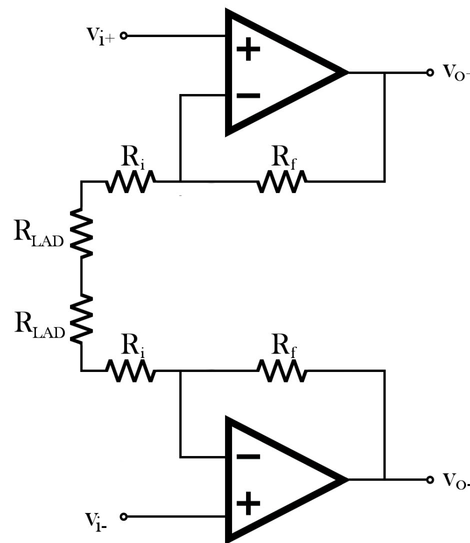


Figure 6.31: Differential amplifier constructed from two PGAs.

6.7.4 Continuous-Time Transimpedance Amplifier

PSoC3/5's transimpedance amplifier is a continuous-time OpAmp with a dedicated and selectable feedback resistor. The TIA configuration is selected by setting the MODE[2:0] bits in the SC[0..3]_CR0 register to '001'. The output of the transimpedance amplifier is a voltage that is proportional to the input current. The conversion gain is determined by the feedback resistor value, R_{fb}, so that:

$$v_o = v_{ref} - (i_i)R_{fb} \tag{6.143}$$

The output voltage, v_o, ie referenced to v_{ref} which can be a routed reference. The value of the feedback resistor, R_{fb}, can be selected programmatically as one of eight values over a range from $20k\,\Omega$ to $1.0\,M\Omega$, as shown in Table 6.9.

Table 6.9: Transimpedance Amplifier feedback resistor values

SC_RVAL[2:0]	Nominal RFB (kΩ)
000	20
001	30
010	40
011	80
100	120
101	250
110	500
111	1000

The inverting input shunt capacitance resulting from parasitic capacitances introduced by the analog global routing and at the input pin can adversely affect stability and therefore an internal shunt capacitance is employed to assure that the TIA remains stable. The feedback capacitance is set by the SC_REDC[1:0] bits in the SCL[0..3]_CR2 register bits [3:2] and the SCR[0..1]_CR2 register, bits [3:2], a shown in Table 6.10.

Table 6.10: TIA Feedback Capacitance Settings.

SC_REDC[1:0]	C_{FB}(pF)
00	0.00
01	1.30
10	0.85
11	2.15

6.8 PSoC3/5 Comparators

Comparators make it possible to make fast comparisons between two voltages particularly with respect to other methods, such as using an ADC. In some applications a DAC is connected

to the negative input to allow the reference voltage to be varied programmatically to allow the comparator to be "adjustable". The positive input is connected to the voltage that is being compared to a reference value. In this case the output goes high when the voltage being compared to the reference voltage is greater than the reference voltage. The output of the comparator can be sampled in software, or digitally routed to another component.

PSoC3/PSoC5 have four comparators configured as shown in Figure 6.32. The configuration of the inputs to the comparators is controlled by the CMPx_SW0, CMPx_SW2, CMPx_SW3, CMPx_SW4, and CMPx_SW6 registers.

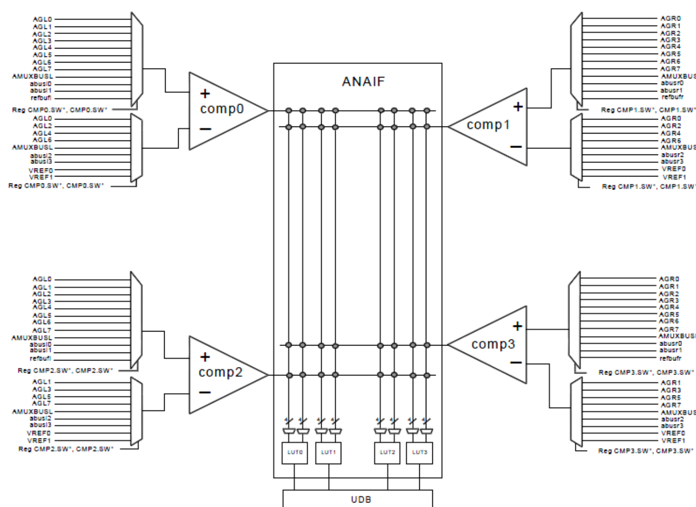


Figure 6.32: Comparator block diagram.

Inputs to the positive terminal can me made from the analog globals, analog locals, the analog mux and the comparator reference buffer. Inputs to the negative input can be made from analog globals/locals/mux and the voltage reference.

6.8.1 Power Settings

The PSoC3/5 comparators can operate in one of three power modes, viz., fast, slow and ultra low power which are selected by the power mode select bits SEL[1:0] in CMPx_CR, the comparator control register. Power modes differ in response time and power consumption. Power consumption is maximum in the fast mode and minimum in the ultra low power mode. The three speed levels are provided to enable a comparator to be optimized for either speed or power consumption.

Inputs to the comparators are via muxes whose inputs include analog globals (AGs), the local analog bus (ABUS), the Analog Mux Bus (AMUXBUS), and precision references. The output from each comparator is routed through a synchronization block to a two-input Lookup Table (LUT). The output of the LUT is routed to the UDB Digital System Interface (DSI). The comparator can also be used to wake-up the device from sleep. An "x" used with a register name denotes the particular comparator number (x = 0 to 3). Connection to the positive input is from analog globals, analog locals, analog mux bus, and comparator reference buffer. Connection to the negative input is from analog globals, analog locals, analog mux bus, and voltage reference.

Comparator output can be passed through an optional glitch filter[59]. The glitch filter is enabled by setting the filter enable (FILT) bit in the control (CMPx_CR6) register. The output of the comparator is stored in the CMP_WRK register and can be read over the PHUB interface.

PSoC3/5 comparators have the following features:

- Low input offset
- Low power mode
- Multiple speed modes
- Output routable to digital logic blocks or pins
- Selectable output polarity
- User controlled offset calibration
- Flexible input selection
- Speed power tradeoff
- Optional 10 mV input hysteresis
- Low input offset voltage (<1 mV)
- Glitch filter for comparator output
- Sleep wake-up

Four LUTs allow logic functions to be applied to comparator outputs. The LUT logic has two inputs:

- Input A is selected using MX_A[1:0] bits in LUT control (LUTx_CR1:0) register
- Input B is selected using MX_B[1:0] bits in LUT Control (LUTx_CR5:4) register

The logic function implemented in the LUT is selected using control bits (Q[3:0]) in the LUT Control register (LUTx_CR). The bit settings for various logic functions are given in Table 6.11.

The output of the LUT is routed to the digital system interface of the UDB array. From the digital system interface of the UDB array, these signals can be connected to other blocks in the device, or to an I/O pin.

The state of the LUT output is indicated in the LUT output (LUTx_OUT) bit in the LUT clear-on-read sticky[60] status (LUT_SR) register and can be read over PHUB interface. The LUT interrupt can be generated by all four LUTs and is enabled by setting the LUT mask (LUTx_MSK) bit in the LUT mask (LUT_MSK) register.

[59]Glitch filters are employed to remove transients, i.e., "glitches" in the output of a comparator.
[60]A sticky bit is a bit in a register that retains its value after the event that caused it value has occurred.

Table 6.11: Control Words for LUT

Control Word (Binary)	Output (A and B are LUT Inputs)
0000	False(`0')
0001	A AND B
0010	A AND (NOT B)
0011	A
0100	(NOT A) AND B
0101	B
0110	A XOR B
0111	A OR B
1000	A NOR B
1001	A XNOR B
1010	NOT B
1011	A OR (NOT B)
1100	NOT A
1101	(NOT A) OR B
1110	A NAND B
1111	TRUE (`1')

6.8.1.1 Hysteresis

As discussed previously, hysteresis helps to avoid excessive toggling of the comparator output when the signals are noisy in applications that compare signals that are very close to each other in terms of sign and magnitude,. The 10 mV hysteresis level is enabled by setting the hysteresis enable (HYST) bit in the control (CMPx_CR5) register.

6.8.1.2 Wake-Up from Sleep

The comparator can run in sleep mode and the output can be used to wake-up the device from sleep. Comparator operation in sleep mode is enabled by setting the override (PD_OVERRIDE) bit in the control (CMPx_CR2) register.

6.8.1.3 Comparator Clock

The comparator output changes asynchronously but it can be synchronized with a clock. The clock source can be one of the four digitally-aligned analog clocks or any UDB clock. Clock selection is done by the mx_clk bits [2:0] of the CMP_CLK register. The selected clock can be enabled or disabled by setting or clearing the clk_en (CMP_CLK [3]) bit. Comparator output synchronization is optional and can be bypassed by setting the bypass_sync (CMP_CLK [4]) bit.

6.8.1.4 Offset Trim

Comparator offset is dependent on the common mode input voltage to the comparator. The offset is factory trimmed for common mode input voltages 0.1V and Vdd - 0.1V to less than 1

mV. If the the common mode input range at which the comparator is to operate is known a priori, a custom trim can be done to reduce the offset voltage further.

6.9 PSoC3/5 Mixers

PSoC3 and PSoC5 provide two types of "mixers"[61], viz., continuous and sampled. These components are single-ended and not intended to function as "precision" mixers. The continuous-time configuration is suitable for multiplying and up-mixing. The discrete-time configuration (sampled) has sample-and-hold capability and is appropriate for sampled- or down-mixing. The continuous-time mixer uses input switches to toggle between the inverting and non-inverting inputs of a programmable gain amplifier for which the gains are 1 and -1, respectively. If a fixed local oscillator is used as a sampling clock, the mixer can be used to perform frequency conversion of a signal.

The PSoC3/5 mixer has the following features:

- Power settings are adjustable.
- Continuous-time up-mixing[62] with input frequencies up to 500 kHz and sample clock rates up to 1 MHz. Discrete time samples and hold mixing with input frequencies up to 1 MHz and sample clock rates to 4 MHz.
- Selectable reference voltages.

6.9.1 Basic Mixing Theory

Before proceeding some basic mixing concepts need to be introduced. The term "mixing" in the present context refers to the mixing, or more accurately stated, multiplying of two signals. Given two signals such as

$$y_1 = A_1 sin(\omega_1 t + \phi_1) \tag{6.144}$$

$$y_2 = A_2 sin(\omega_2 t + \phi_2) \tag{6.145}$$

The product, Y, is given by

$$Y = y_1 y_2 = A_1 A_2 [sin(\omega_1 t + \phi_1)][sin(\omega_2 t + \phi_2)] \tag{6.146}$$

but,

$$sin(u)sin(v) = \frac{1}{2}\Big[cos(u - v) - cos(u + v)\Big] \tag{6.147}$$

and therefore,

$$Y = \frac{A}{2}\Big[cos[(\omega_1 - \omega_2)t + \Phi_1 - cos[(\omega_1 + \omega_2)t + \Phi_2]\Big] \tag{6.148}$$

where $A = A_1 A_2, \Phi_1 = \phi_1 - \phi_2$ and $\Phi_2 = \phi_1 + \phi_2$. Thus the result of mixing two sine wave signals is to produce the sum and difference of the the two signals in terms of frequency which together with the two original signals, results in the presence of four signals. Note that the resulting sum and difference signals have also undergone a phase shift. Any of the resulting signals can, if necessary, be subsequently removed, e.g., by filtering.

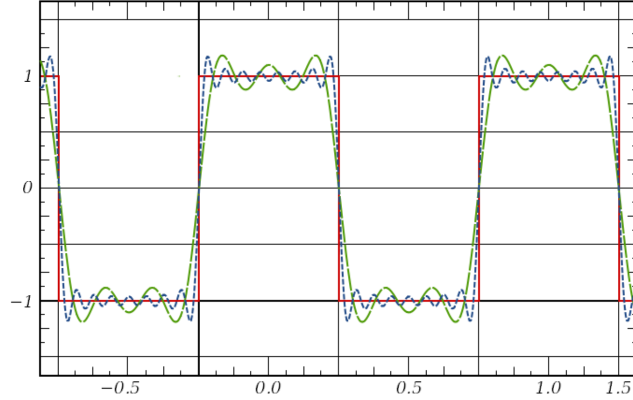


Figure 6.33: An example of cosine harmonics used to represent a square wave.[47]

If y_2 is a square wave, then it can be expressed as a trigonometric partial sum by the following:

$$y_2 = \frac{4}{\pi}\left[cos(\omega_2 t) - \frac{cos(3\omega_2 t)}{3} + \frac{cos(5\omega_2 t)}{5} - \cdots\right] \tag{6.149}$$

as illustrated in Figure 6.33.[63]

If y_1 is defined as:

$$y_1 = cos(\omega_i t) \tag{6.150}$$

Then,

$$Y = y_1 y_2 = \left[A_1 cos(\omega_1 t)\right]\frac{4}{\pi}\left[cos(\omega_2 t) + \frac{cos(3\omega_2 t)}{3} + \frac{cos(5\omega_2 t)}{5} - \ldots\right] \tag{6.151}$$

so that,

$$Y = y_i y_{clk} = \frac{2A_1}{\pi}\left[cos(\omega_- t) - \frac{cos(3\omega_{3-} t)}{3} + \frac{cos(5\omega_{5-} t)}{5} - \ldots\right] \tag{6.152}$$

$$+\frac{2A_1}{\pi}\left[cos(\omega_+ t) - \frac{cos(3\omega_{3+-} t)}{3} + \frac{cos(5\omega_{5+} t)}{5} - \ldots\right] \tag{6.153}$$

[61] Mixing in the present context is in actuality the multiplication of two signals resulting in the production of four output signals, viz., the sum, difference and two mixed signals.

[62] Up-mixing refers to mixing to signals and producing a signal at a frequency which is the sum of the two mixed signal frequencies. Similarly, down-mixing produces a signals whose frequency is the difference of the frequencies of the two mixed signals.

[63] The "ringing effects" seen at the corners of this square wave are known as the Gibbs' or Gibbs-Wilbraham phenomenon and was first observed by Wilbraham in 1848 [46] and subsequently "rediscovered" by Gibbs in 1898 who provided a much more rigorous mathematical foundation for it. This effect is commonly encountered in the course of processing digital signals, e.g., to the series.

where,

$$\omega_{+} = \omega_{clk} \quad (6.154)$$
$$\omega_{3+} = 3\omega_{clk} + \omega_i \quad (6.155)$$
$$\omega_{5+} = 5\omega_{clk} + \omega_i \quad (6.156)$$
$$\cdots \quad (6.157)$$
$$\omega_{-} = \mid \omega_{clk} - \omega_i \mid \quad (6.158)$$
$$\omega_{3-} = \mid 3\omega_{clk} - \omega_i \mid \quad (6.159)$$
$$\omega_{5-} = \mid 5\omega_{clk} - \omega_i \mid \quad (6.160)$$
$$\cdots \quad (6.161)$$
$$(6.162)$$

Thus in this case, in addition to the sum and difference frequencies for v_i and v_{clk} the third, fifth and all additional higher-order, odd harmonics are present in the output.[64] The unwanted harmonics can be removed by suitable filtering.

6.9.2 PSoC3/5 Mixer API

The PSoC3/5 mixer has an API consisting of three function calls:

- **void Mixer_Start(void)** powers up the Mixer. Performs all of the required initialization for the mixer and enables power to the block. The first time the routine is executed, the input and feedback resistance values are configured for the operating mode selected in the design. When called to restart the mixer following a **Mixer_Stop()** call, the current component parameter settings are retained.
- **void Mixer_Stop(void)** powers down the Mixer. This does not affect mixer type or power settings.

and

- **void Mixer_SetPower(uint8 power)** - Set drive power to one of the following four levels.
  Mixer_MINPOWER -Lowest active power and slowest reaction time.
  Mixer_LOWPOWER - Low power and speed.
  Mixer_MEDPOWER - Medium power and speed.
  Mixer_HIGHPOWER - Highest active power and fastest reaction time.

6.9.3 Continuous-Time Mixer

As shown in Figure 6.34, the OpAmp is configured as a PGA that uses the lo_clock input signal to toggle between an inverting uniy gain PGA and a non-inverting unity gain buffer. The output signal includes frequency components at ($F_{clk} \pm F_{in}$) plus terms at odd harmonics of the LO frequency $\pm$ the input signal frequency: $3 * F_{clk} \pm F_{in}$, $5 * F_{clk} \pm F_{in}$, $7 * F_{clk} \pm F_{in}$, etc. The continuous-time mode is preferable for "up-conversion" since it provides much higher conversion gain than the sampled mixer. In order to assure optimal performance the value for F_{clk} should meet the Nyquist criteria[65], viz.,

$$F_{clk} > 2F_{out} \quad (6.163)$$

[64] Note that the v_i and v_{clk} frequencies are not present in the output.

[65] Simply stated, the Nyquist criteria states that a sampled, band-limited, analog signal can be completely reconstructed if the sampling rate is twice the highest frequency component in the original analog signal.

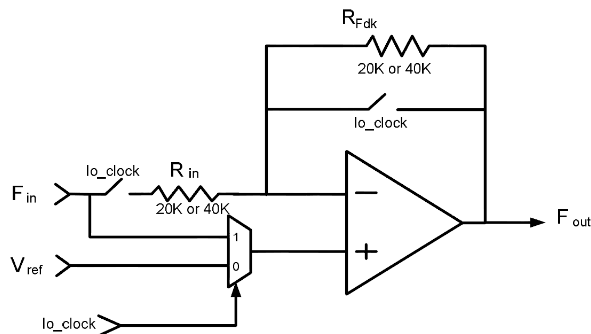


Figure 6.34: PSoC3/5 configuration for a CT mixer. [37]

An example of an implementation of the CT mixer input and output waveforms is shown in Figure 6.35 and is based on a CT block.

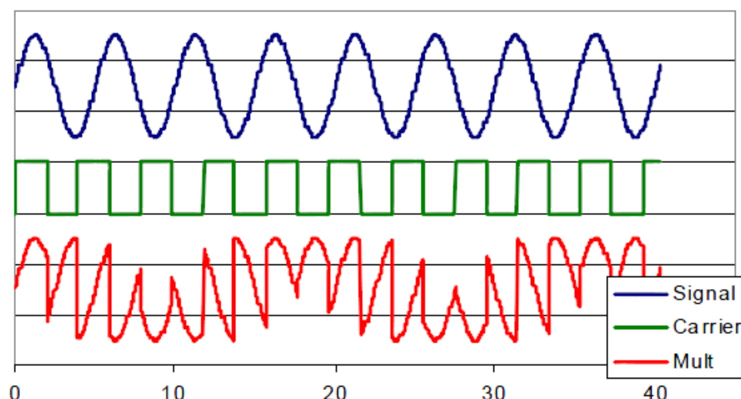


Figure 6.35: An example of CT Mixer input and output waveforms. [37]

6.9.4 Sampled-Mixer

Before beginning a discussion of the sampled-mixer it will be necessary to discuss two of the types of encoding used in digital systems, viz.,

- **Non-Return to Zero (NRZ-L)** - two distinct voltage levels are used to represent zeros and ones.x A voltage level of zero is not used to represent the binary value zero. Typically a positive value is used for one and a negative value for zero with both being of the same absolute value.
- **Non-Return to Zero Inverted (NRZI)** - any transition to high or low represents the a binary value of one, the absence of a transition represents a binary value of zero.

The sampled-mixer provided in PSoC3/5 is basically a NRZ sample and hold circuit[66] with very fast response. Unlike the CT mixer which has an upper frequency limit of 4 MHz, the sampled

[66]Sample and hold circuits are used to sample a time dependent signal and then hold that sample for a period of time to allow certain operations to be carried out with respect to the sample. A common method of "holding" is to store the sample via a capacitor that is capable of holding the sample for the time required without degrading it, i.e., with sufficiently low leakage current.

mixer can accept input frequencies as high as 14 MHz. The output of the sampled-mixer can be used as input to an internal ADC[67] via analog routing, or in conjunction with an external device such as a ceramic filter[68].

As mentioned previously, the sampled mixer shown in Figure 6.36 is used primarily for down-

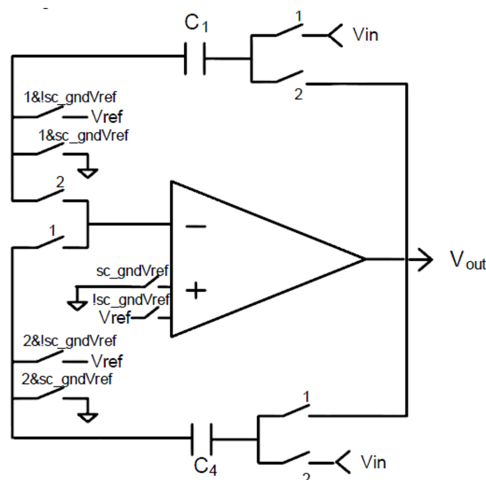


Figure 6.36: Sampled (Discrete-Time) sample and hold mixer. [37]

conversion which can be accomplished by removing the undesired products resulting from mixing the input frequency and the sample clock. The NRZ sample and hold functionality is based on alternately selecting one of two capacitors as the the integrating capacitor. Thus one capacitor, either C_1 or C_4, serves as the integrating capacitor, while the other is used to sample the input signal. This configuration is designed such that the input signal is sampled at a rate less than the input signal frequency and the integration of each new value occurs on the rising edge of f_{clk}.

If $f_{clk} > f_{in}/2$, then

$$f_{out} = \mid f_{in} - f_{clk} \mid + \text{aliasing components} \tag{6.164}$$

If $f_{clk} < f_{in}/2$, then

$$f_{out} = \mid f_{in} - N f_{clk} \mid \tag{6.165}$$

for the largest integer value of N such that $N f_{clk} < f_{in}$

For example, if the desired down-converted frequency is 500 kHz and the input frequency is 13.5 MHz, Equation (6.165) is satisfied for values of $N = 7$ and $f_{clk} = 2$ MHz. Examples for N=1 and N=3 are shown in Figures 6.37 and 6.38, respectively.

[67] If the output is routed to an internal ADC both the mixer and the ADC must employ the sample sample clock.

[68] For example the 455 kHz Murata Cerafil. 455 kHz is a standard frequency used in receivers as part of the intermediate frequency (IF) stage.

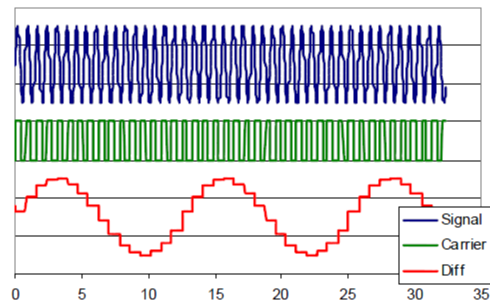


Figure 6.37: Sampled mixer waveforms for N=1. [37]

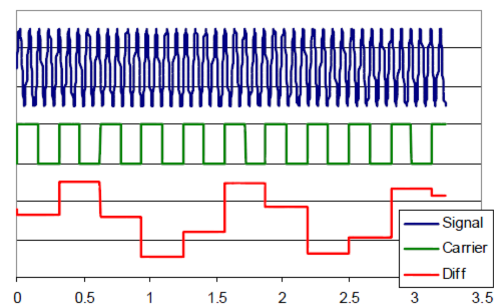


Figure 6.38: Sampled mixer waveforms for N=3. [37]

Example 6.6: C source code for implementing a mixer that employs an internal local oscillator

```
#include <device.h>
#include "Mixer_1.h"
#include "lo_clk.h"
void main()
{
/* Setup Local Oscillator Clock */
lo_clk_Enable();
lo_clk_SetMode(CYCLK_DUTY);
/* API Calls for Mixer Instance */
Mixer_1_Start();
Mixer_1_SetPower(Mixer_1_HIGHPOWER);
while (1)
{
}
}
```

6.10 Filters

Almost all embedded systems are confronted with the potential for noise altering the response and/or performance of the system. Various techniques exist for dealing with noise, depending on the source, type of noise, amplitude, spectral composition, etc. While the best cure for noise is to avoid it, the reality is that it is almost always present and must be dealt with directly. Analog filters are often used for this purpose and consist of two fundamental types, viz., passive and active. Digital filters are also used but typically require that the analog signal to be filtered first be converted to a digital format, processed and then converted back to analog form. Although digital filters do have truly outstanding characteristics they also involve what can be significant and perhaps prohibitive overhead in some applications that may not be acceptable in some application. At very high frequencies, digital filters are less attractive than their analog counterparts for a variety of reasons.

6.10.1 Ideal Filters

Filters provide a method for separating signals, e.g., as in the case of an amplitude modulated or frequency carrier[69], allow signals to be restored by removing unwanted signals/noise, and restore a signal that may have been otherwise altered. Filters may be based on combinations of RLC components, rely on mechanical resonances, employ the piezoelectric effect, utilize acoustic wave techniques, etc. , depending on the operating environment the application, frequency range and the desired filter characteristics.

Ideal filters can be characterized as:

- **Lowpass (LPF)** - "passes" all frequencies below a certain frequency with no change in amplitude and specifiable phase shift.
- **Bandpass (BPF)** - "passes' all frequencies above a given frequency and below an upper frequency with no change in amplitude and specifiable phase shift.
- **Highpass (HPF)** - "passes" all frequencies above certain frequency with change in amplitude and specifiable phase shift.
- **Notch (NPF)**)- "blocks" frequencies within a specified range.
- **Allpass** (APF) - "passes" all frequencies and alters only the phase, e.g., unity gain at all frequencies. The phase shift at the corner frequency for all frequencies is 90°. This type of filter is often used to match phase, introduce a delay and creating a 90°degree phase shift for certain types of circuits. (See Figure

Ideal filters have a number of important characteristics that should be kept in mind when designing a filter for a particular application, viz., no attenuation in the passband, sharp cutoff and complete attenuation in the stop band. Such filters are referred to a "brickwall" filters and are represented graphically as shown in Figure 6.39.
Thus a filter is considered ideal if

$$| H(\omega) | = \begin{cases} 1, & \text{if } \omega \text{ is in the passband} \\ 0, & \text{if } \omega \text{ is in the stopband} \end{cases} \tag{6.166}$$

and,

$$\angle H(\omega) = \begin{cases} -\omega\tau, & \text{if } \omega \text{ is in the passband} \\ 0, & \text{if } \omega \text{ is in the stopband} \end{cases} \tag{6.167}$$

[69] There are of course a variety of techniques for demodulating signals of which filters represent but one, e.g. mixing techniques, a topic discussed in section 6.9.

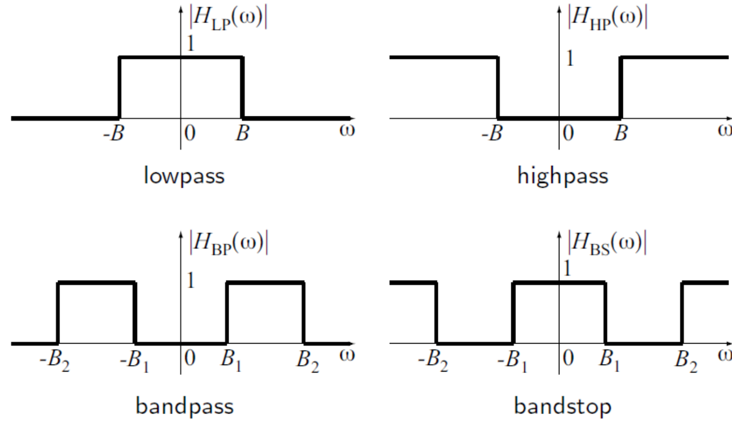


Figure 6.39: "Brickwall" Transfer functions for ideal filters.

where τ is a positive constant. The phase shift for ideal filters is shown in Figure 6.40.

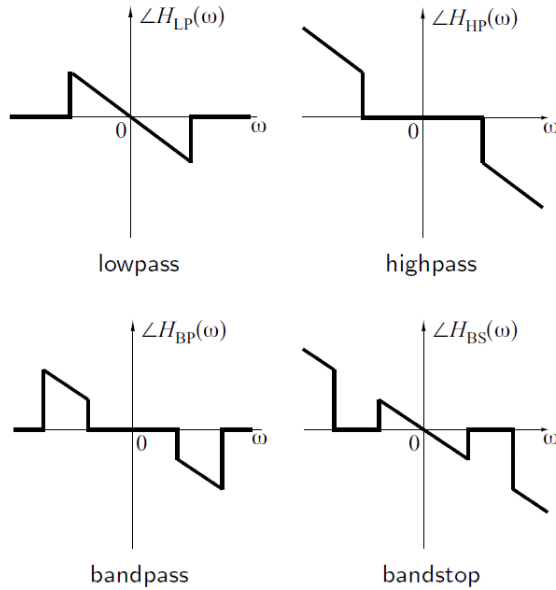


Figure 6.40: Phase as a function of frequency for ideal filters.

6.10.2 Bode Plots

Filter design has been greatly facilitated by the availability of computer programs designed to handle the mathematical complexities. Their ability to display the results of the associated calculations significantly simplifies the designers task. A common graphical characterization of a filter is the so-called Bode Plot[70]. A Bode plot is a graphical representation of a transfer function that represents a linear, time-invariant system in which the abscissa is usually the log of the

[70] This technique was introduced by Hendrik Bode in 1938 at Bell Laboratories where he worked as an engineer.

frequency and the ordinate is the log of the system's gain. A typical Bode plot is shown in Figure 6.41.

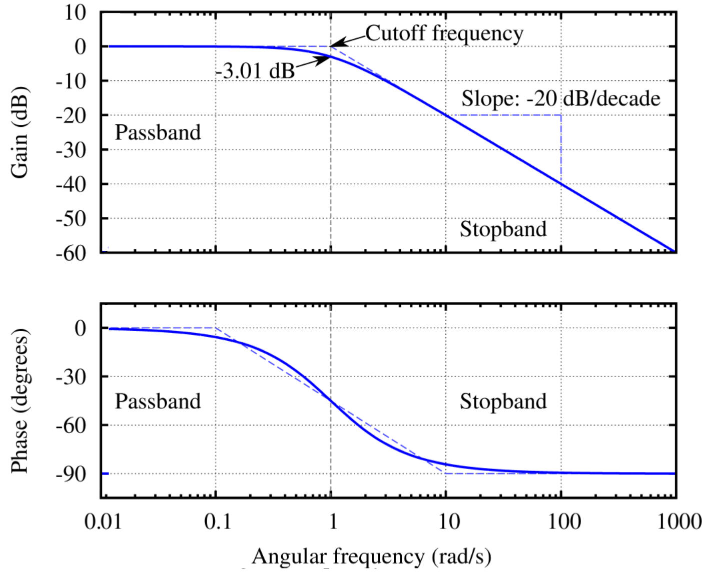


Figure 6.41: Bode plot characteristics for a 1st-order Butterworth filter.

6.10.3 Passive Filters

Passive filters typically consist of combinations of resistors, capacitors and in some cases inductors[71] configured in various ways to provide the type of filter needed and the desired filter characteristics. Passive filters do not power supplies. However, passive filters are affected by changes in the components' capacitance, resistance and inductance as a result of humidity, temperature, aging, vibration, etc., because any such variation can seriously, and adversely, alter a filter characteristics. Also, at low frequencies the physical size of components, given the inverse relation between physical size and operating frequencies, can be a problem. However, even though passive filters are inherently stable, unlike their active filter counterparts which can oscillate, they tend to be more linear than active filters, and can be designed to handle arbitrarily large voltages, currents and frequencies. Active filters employing solid state devices, such as operational amplifiers, tend to be frequency, current and voltage limited.

One of the simplest forms of passive filter consists of a combination of resistor and capacitor as shown in Figure 6.42. Although this type of filter is simplicity itself it is instructive to carry out a brief analysis in order to illustrate some important aspects of filters.

Treating this circuit as a voltage divider and defining $s = \sigma + j\omega$ leads to the result that:

$$v_r = \frac{sRC}{1 + sRC} v_i(s) \tag{6.168}$$

and,

$$v_c = \frac{1}{1 + sRC} v_i(s) \tag{6.169}$$

[71] Inductors are often not used because of size, cost and other considerations.

therefore the transfer functions for the resistor and capacitor are given by:

$$H_r(s) = \frac{v_R}{v_i} = \frac{sRC}{1 + sRC} \tag{6.170}$$

$$H_c(s) = \frac{v_R}{v_i} = \frac{1}{1 + sRC} \tag{6.171}$$

Note that assuming that the excitation is steady state, then $s = j\omega$, so that when:

$$s = -\frac{1}{RC} \tag{6.172}$$

Equations (6.170) and (6.171) become infinite and Equation (6.172) is said to be a"pole'[72]' for both transfer functions. In addition Equation(6.170) is zero when $s = 0$ which is referred to as a "zero" of the resistor's transfer function.

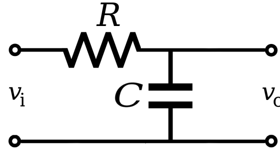


Figure 6.42: A very simple, passive, lowpass filter.

Gain is defined as the ratio of output voltage to input voltage and therefore for the capacitor:

$$A_C = \left|H_C\right| = \frac{1}{\sqrt{1 + (wRC)}^2} \tag{6.173}$$

and for the resistor:

$$A_R = \left|H_R\right| = \frac{\omega RC}{\sqrt{1 + (wRC)}^2} \tag{6.174}$$

Notice that as expected the capacitor gain approaches zero, but the resistor gain approaches unity, as the frequency increases. Thus the RC combination shown in Figure 6.42 functions as a lowpass filter if the output is taken across the capacitor and as a highpass filter if the output is taken across the resistor. Such filters are used, but because of the resistive component it does degrade the signal can making them unsuitable for some applications. One of the problems with this type of filter is that either resistive or reactive loads will change the characteristics of this filter and therefore must be take into account in designing the filter. An operational amplifier can be used to address both of these types of degradation, but has the undesirable feature of increasing the amplitude of any noise present at the input terminals[73]. The phase shift introduced by the resistor and capacitor are given by:

$$\Theta_r = tan^{-1}(\frac{1}{\omega RC}) \tag{6.175}$$

and,

$$\Theta_c = tan^{-1}(-\omega RC) \tag{6.176}$$

[72]A pole is defined as

[73]Filters employing operational amplifiers are referred to "active" filters because they require power supplies.

Active filters require external power, and may cost more, but may also be of much smaller size and offer better characteristics and performance than their passive counterparts. However, active filters can introduce noise into a system[74], if not carefully designed and implemented. This can arise as a result of noise introduced through the power supplies, introduction of noise as a result of the use of operational amplifiers, etc.

6.10.4 Analog Active Filters

Analog active[75] filters are characterized by a number of factors including the:

1. number of stages or sections employed in a cascaded fashion[76],
2. cutoff frequency, i.e., the point at which the response of the filter falls below 3 dB,
3. response of the filter in the stop band,
4. gain as a function of frequency in the passband,
5. phase shift in the passband,
6. degree of ringing, if any,
7. rate of roll-off,

and

8. transient response

Lowpass filters of the four types shown in Figure 6.43 are commonly used and of these the Butterworth[77] filter is sufficient for most lowpass applications. Items 2-6 of this list can be determined by examining the Bode plots of the type shown in Figure 6.41 for the phase and gain of a filter. A fifth type, the Bessel filter, is very flat in the passband but rolls off more slowly than Butterworth,Elliptic and Chebyshev filters.

6.10.4.1 Sallen-Key Filters (S-K)

PSoC3 does not provide explicit internal support for analog LPF, BPF, HPF and BSF because the switched-capacitance block designed for PSoC3 has been optimized for other configurations. However, PSoC3/5 do have operational amplifiers that can be used in conjunction with external components to provide analog filtering. A popular design methodology in such cases was suggested by Sallen-Key [39]. This type of filter is referred to as a voltage-controlled, voltage source, or VCVS filter. It has gained widespread popularity in part because of its relative simplicity, ability to employ conventional operational amplifiers, excellent passband characteristics, relatively low cost, requires few components and the fact that multiple S-K filters can be cascaded without significant signal degradation.

Assuming the generic configuration of the unity gain Sallen-Key filter shown in Figure 6.44 the transfer function for this type of unity gain filter is given by:

$$\frac{v_o}{v_i} = \frac{Z_3 Z_4}{Z_1 Z_2 + Z_4(Z_1 + Z_2) + Z_3 Z_4} \tag{6.177}$$

[74]One potential noise source for active filters are the power supplies required for the operational amplifier(s).

[75]An active filter is one which is a combination passive components and components capable of adding gain . The latter therefore requires input power.

[76]The advantage of cascading filters is that it increases the overall effective "order" of the filter which in turn increases the roll-off at a rate of 6 dB/octave times the equivalent order of the cascaded filters. One heuristic often employed to determine the order of each filter stage is to count the number of storage elements in each stage, e.g., the number of capacitors, which is usually the same as that stage's order.

[77]In 1930, S Butterworth published an important paper describing his design methodolgy for what became known as the Butterworth filter. He wound wire around cylinders that were 1.25 inches in diameter and 3 inches long to create resistors and inductors and placed capacitors inside the cylinders to complete the filter.

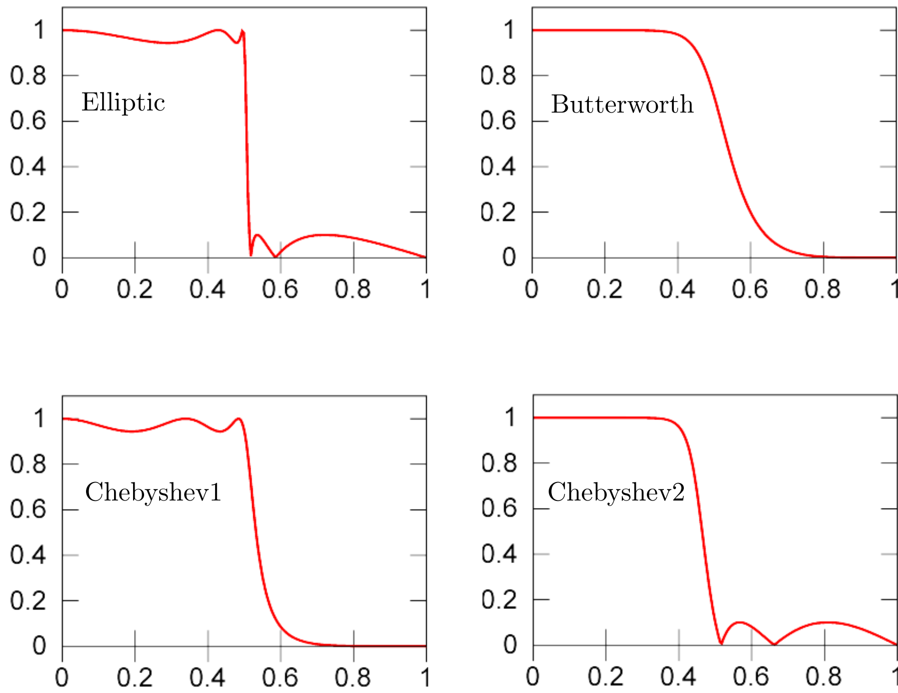


Figure 6.43: Normalized graphs of common 5th-order lowpass filter configurations.

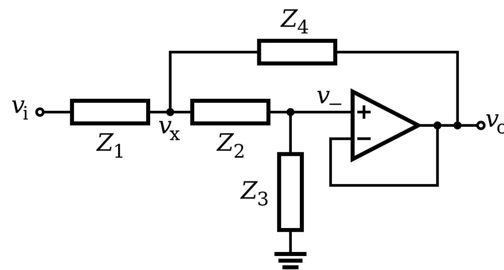


Figure 6.44: The generic form of the Sallen-Key filter.

By choosing different impedances for Z_1, Z_2, Z_3, Z_4 the Sallen-Key topology can be transformed into a filter exhibiting highpass, bandpass and lowpass characteristics. The nominal frequency limit of a filter is referred to as the "corner" frequency which is the frequency at which the input signal is reduced to 50% of it power[78] just prior to the output terminals.[79] Beyond that point the attenuation is usually referred to in terms of dB/octave[80] or dB/decade.

A typical lowpass filter response curve is shown in Figure 6.45. As shown the "cutoff fre-

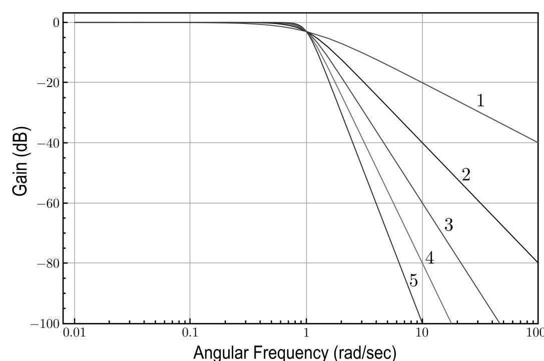


Figure 6.45: Bode plot of an nth order Butterworth filters for n = 1-5.

quency" is defined as the point at which the response curve is -3 dB down and in this particular case the response curve falls off at the rate of -20 dB/decade. As can be seen from this figure, the order of the filter dramatically effects the rate of roll of from the passband to the stopband.

6.10.4.2 Sallen-Key Unity-Gain Lowpass Filter

As shown in Figure 6.46, a Sallen-Key lowpass filter can be configured by setting:

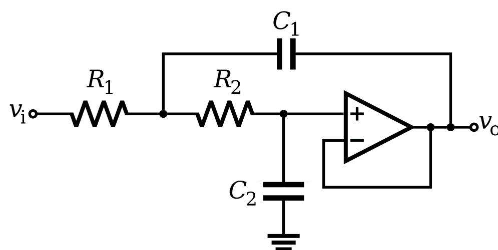


Figure 6.46: A Sallen-Key unity-gain lowpass filter.

$$Z_1 = R_1 \tag{6.178}$$
$$Z_2 = R_2 \tag{6.179}$$
$$Z_3 = -\frac{j}{\omega C_1} = \frac{1}{sC_1} \tag{6.180}$$
$$Z_4 = -\frac{j}{\omega C_2} = \frac{1}{sC_2} \tag{6.181}$$

[78] The so-called "half power point".

[79] This is also referred to as the "- 3dB down" point.

[80] An octave implies a doubling of frequency, e.g., -6 dB/octave implies that the signal is reduced by 50% if the frequency is doubled. Decade refers to a change in frequency by a factor of ten.

and after rearranging,

$$H(s) = \frac{\frac{1}{R_1R_2C_1C_2}}{s^2 + \left[\frac{R_1+R_2}{R_1+R_2C_1}\right]s + \frac{1}{R_1R_2C_1C_2}} \tag{6.182}$$

which is in the form of the transfer function of a second order unity-gain lowpass filter, i.e.,

$$H(s) = \frac{\omega_c^2}{s^2 + 2\zeta\omega_c s + \omega_c^2} \tag{6.183}$$

If ω_c^2 is defined as

$$\omega_c^2 = \frac{1}{R_1R_2C_1C_2} = \omega_0\omega_n \tag{6.184}$$

then

$$f_c = \frac{1}{2\pi\sqrt{R_1R_2C_1C_2}} \tag{6.185}$$

and,

$$2\zeta = \frac{1}{Q} = \frac{\sqrt{R_1R_2C_1C_2}}{R_1C_1 + R_2C_1} = \left[\frac{1}{R_1} + \frac{1}{R_2}\right]\frac{\sqrt{R_1R_2C_1C_2}}{C_1} \tag{6.186}$$

After rearranging, Equation (6.186) becomes

$$Q = \frac{\sqrt{R_1R_2C_1C_2}}{C_2(R_1 + R_2)} \tag{6.187}$$

The Q or selectivity value value determines the the height and width of the frequency response and is defined for bandpass filter as

$$Q = \frac{f_c}{f_H - f_L} = \frac{f_c}{Bandwidth} \tag{6.188}$$

and

$$f_c = \sqrt{f_H f_L} \tag{6.189}$$

which is the geometric mean for the -3 dB points of f_H and f_L. The reader may be puzzled by the meaning of Q for a lowpass filter given how it is defined. In the case of a Butterworth lowpass filter[81] it is used as a measure of the filters response, i.e. under-, critically- or highly-damped. Because R_1, R_2, C_1 and C_2 are independent variables, it is possible to simplify Equations 6.185 and 6.187 by choosing R_2 and and C_2 to be integer multiples of R_1 and C_1, respectively.

6.10.4.3 Sallen-Key Highpass Filter

Similarly, the configuration for a Sallen-Key 2nd-order highpass filter is shown in Figure 6.47. In this case the transfer function for the S-K bandpass filter is given by

$$H(s) = \frac{s^2}{s^2 + \frac{R_1}{R_1R_2\left(\frac{C_1C_2}{C_2+C_2}\right)}s + \frac{1}{\left[R_1R_2\left(\frac{C_1C_2}{C_1+C_2}\right)\right](C_1+C_2)}} \tag{6.190}$$

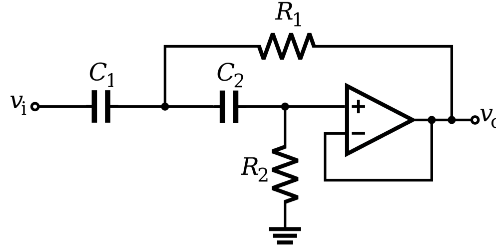


Figure 6.47: A Sallen-Key highpass filter.

and the general for for a 2nd-order high pass filter is given by

$$H(s) = \frac{s^2}{s^2 + 2\zeta\omega_c s + \omega_c^2} \tag{6.191}$$

and therefore

$$Q = \frac{\sqrt{R_1 R_2 C_1 C_2}}{R_1(C_1 + C_2)} \tag{6.192}$$

and

$$f_c = \frac{1}{2\pi\sqrt{R_1 R_2 C_1 C_2}} \tag{6.193}$$

6.10.4.4 Sallen-Key Bandpass Filter

Similarly, the transfer function for the bandpass filter shown in Figure 6.48 is given by

$$H(s) = \frac{\frac{R_a+R_b}{R_a}\frac{s}{R_1C_1}}{s^2 + \left[\frac{1}{R_1C_1} + \frac{1}{R_2C_1} + \frac{1}{R_2C_2} - \frac{R_b}{R_aR_fC_1}\right]s + \left[\frac{R_1+R_2}{R_1R_fR_2C_1C_2}\right]} \tag{6.194}$$

The denominator is of the general form for bandpass filters and expressed as:

$$H(s) = \frac{G\omega_n^2 s}{s^2 \; + \; 2\xi\omega_0 s \; + \omega_0^2} \tag{6.195}$$

where G is the so-called inner gain[82] of the filter and given by

$$G = \frac{R_a + R_b}{R_a} \tag{6.196}$$

and the gain at the peak frequency[83] is given by

$$A = \frac{G}{G - 3} \tag{6.197}$$

and the center frequency by

$$f_0 = \frac{1}{2\pi}\sqrt{\frac{R_f + R_1}{R_1 R_2 R_f C_1 C_2}} \tag{6.198}$$

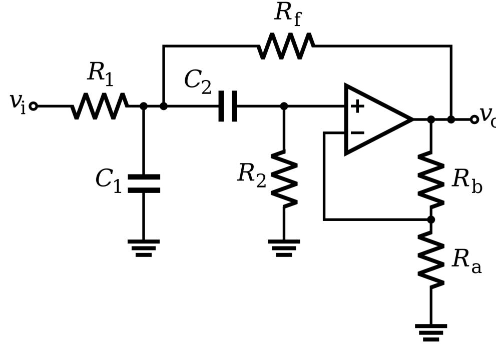


Figure 6.48: A Sallen-Key bandpass filter.

Setting

$$C_1 = C_2 \tag{6.199}$$
$$R_2 = \frac{R_1}{2} \tag{6.200}$$
$$R_a = R_b \tag{6.201}$$

yields

$$G = 2 \tag{6.202}$$
$$A = 2 \tag{6.203}$$

$$f_0 = \frac{1}{2\pi}\sqrt{\frac{R_f + R_1}{R_1^2 C_1^2 R_f}} = \frac{1}{2\pi R_1 C_1}\sqrt{1 + \frac{R_1}{R_f}} \tag{6.204}$$

6.10.4.5 An Allpass Filter

The phrase "allpass filter" is in some respects a misnomer because such filters pass all frequencies at constant gain. The important characteristic of this particular type of so-called filter is that its phase response varies linearly with respect to frequency. This makes allpass filters useful R and C form a lowpass filter as shown previously and therefore its transfer function is given by

$$H(s) = \frac{1}{1 + sRC} \tag{6.205}$$

The current into the negative feedback loop is given by

$$\begin{aligned}
\frac{v_i - v_{-input}}{R_f} &= \frac{v_i - v_i H(s)}{R_f} \\
v_{+input} - i_f R_f &= v_i H(s) - \left[\frac{v_i - viH(s)}{R_f}\right] R_f \\
&= [2H(s) - 1] v_i \\
&= \left[\frac{2}{1 + sRC}\right] v_i = \left[\frac{1 - sRC}{1 + sRC}\right] v_i
\end{aligned} \tag{6.206}$$

[81] A 4th order Butterworth filter having a Q of .707 is maximally flat in he passband.
[82] This is the gain determined by the negative feedback loop.
[83] Note that if the gain is $\leqslant 3$ the circuit will oscillate.

and therefore

$$| H | = 1 \tag{6.207}$$

and

$$\angle\text{H} = -2\tan^{-1}(\omega RC) \tag{6.208}$$

which for

$$\omega RC = 1 \Rightarrow \angle H = -90^\circ \tag{6.209}$$

which means that gain is independent of frequency and phase that is dependent on frequency.

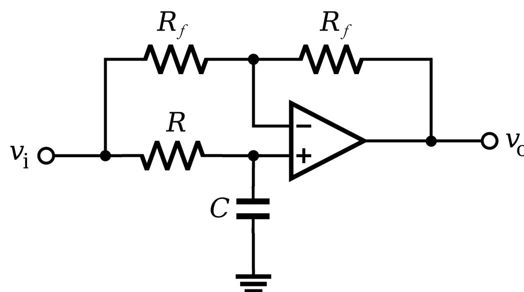


Figure 6.49: A simple first-order allpass filter.

6.10.5 Digital Filters

The conventional wisdom that digital techniques are universally superior to that of their analog counterparts is not always correct. In principle, the concept of converting all incoming signals to digital form and then carrying out whatever programmatic operations may be required, on those signals, to determine what actions should, or should not be taken, if any, would seem to be the best approach.

Digital filters certainly have much to offer when compared to their analog counterpart in terms of significantly better performance in terms of passband ripple, greater stopband attenuation, greatly reduced design times, better signal to noise characteristics[84], less nonlinearity but are not necessarily the answer for all embedded systems. In cases for which the responsiveness of the system is a primary concern the processing overhead, i.e., latency, associated with digital filters can in some cases make them inapplicable. Digital filters are generally more complex than analog filters, have good EMI and magnetic noise immunity, very stable with respect to temperature and time, provide excellent repeatability, but do not generally speaking offer the dynamic range of analog filters or have the capability operate over as wide a frequency range of a comparable analog filter.

Since digital filters are discrete time devices, difference equations are often used to model their behavior, e.g,

$$y_n = -a_1 y_{n-1} - a_2 y_{n-2} - \cdots - a_N y_{n-N} + b_0 x_n + \cdots + b_{n-M} \tag{6.210}$$

$$= -\sum a_k y_{n-k} + \sum b_k x_{n-k} \tag{6.211}$$

[84]Digital filters do introduce some noise in terms of quantization noise, as a result of the conversion from analog to digital and digital to analog of filtered signals, etc.

where a_k and $b_k \in \mathbb{Z}$.

In the z-domain, the transfer function for a LTI IIR digital filter is of the general form given by:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-1} + b_3 z + \ldots + b_n z^{-n}}{1 + a_1 z^{-1} + a_1 z^{-1} + a_1 z + \ldots + a_m z^{-m}} \tag{6.212}$$

If n>m then the filter is said to be a nth order filter and conversely, if m>n, then the filter is said to be an mth order filter.

Digital filters are modeled in terms of adders, multipliers and positive and negative delays.

The are two fundamental types of digital filters:

- Finite Impulse Response (FIR) filters that are nonrecursive, stable, linear with respect to phase, are relatively insensitive to coefficient quantization errors and depend on either the difference of contiguous samples or weighted averages. The latter serves as a lowpass filters and the former as a highpass filter. The impulse function for an FIR filter is of finite duration. Such filters can be expressed mathematically as

$$y[n] = \sum_{k=0}^{M} b_k [n-k] \tag{6.213}$$

  where M is the number of feed forward taps.

  MATLAB provides support for window-based FIR filters by providing

  1. b = fir1(n,Wn)
  2. b = fir1(n,Wn,'ftype')
  3. b = fir1(n,Wn,window)
  4. b = fir1(n,Wn,'ftype',window)
  5. b = fir1(...,'normalization')

  for windowed linear-phase FIR digital filter design.

- Infinite Impulse Response (IIR) filters have impulse responses of infinite duration and can be expressed mathematically as

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=1}^{M} b_k x[n-k] \tag{6.214}$$

  where N,M are the number of feedforward taps and feedback taps, respectively, and a_{k} is the kth feedback tap.[85] Because the output of an IIR filter output is a function of both the previous M outputs and N inputs, it is a recursive filter whose impulse response is of infinite duration. IIR filters typically require fewer numbers of multiplications than than their FIR counterparts, can be used to create filters with characteristics of analog filters but are sensitive to coefficient quantization errors.[86]

  In addition to butter, cheby1, cheby2, ellip and bessel which represent a complete filter design suite, MatLab support for IIR filters includes:

  1. buttord

[85] If $a_k = 0$, then this expression reverts to that of the FIR filter.

[86] Because digital filters introduce quantization errors the positions of poles and zeros in the complex plane can shift which is referred to as coefficient quantization error.

2. cheb1ord,
3. cheb2ord

and,

4. ellipord

6.10.6 Finite Impulse Response (FIR) Filters

Finite impulse filters, of the type shown in Figure 6.50 are causal (non-recursive), inherently

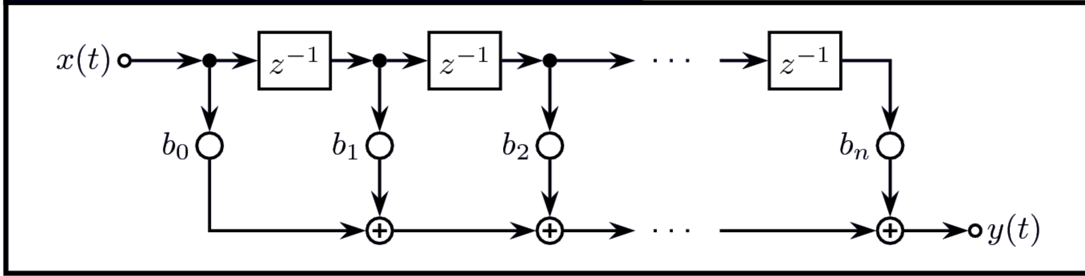


Figure 6.50: A generic Finite Impulse Response filter of order n.

stable (BIBO), do not need feedback, are relatively insensitive to coefficient quantization errors, and capable of providing the same delay for all components of the input signal[87]. In addition, FIR filters are further characterized by the fact that their impulse response is finite. FIR filters are quantifiable in terms of following linear constant-coefficient difference (LCCD) equation:

$$y[n] = b_0x[n] + b_1x[n-N] + \ldots + b_Nx[n-N] = \sum_{k=0}^{N} b_kx[n-k] \tag{6.215}$$

where N is the number of "taps",[88] y[n] is the output at the discrete time instance n and similarly x[n] is each of the input samples. Note that this type of filter does not depend on previous values of y. To determine the impulse response of this particular configuration,

$$x[n] = \delta[n] \tag{6.216}$$

so that Equation (6.215) becomes

$$y[n] = b_0\delta[n] + b_1\delta[n-N] + \ldots + b_N\delta[n-N] = \sum_{k=0}^{N} b_k\delta[n-k] = b_n \tag{6.217}$$

One of the most commonly encountered types of FIR filers is known as a "Moving Average Filter" which can be either LP, BP, or HP. It is based on a concept of averaging a number of samples to produce each of the output values, i.e.,

$$y[n] = \frac{1}{M}\sum_{k=0}^{M-1} x[n+k] \tag{6.218}$$

where M is the number of samples in the average. As simple as this technique is, it turns about to be an excellent method for removing random noise while retaining a sharp step response.

[87]A very important consideration for audio and video applications.

[88]The number of taps is a measure of the number of terms in Equation (6.215) which is $(N+1)$ and N is the order of the filter. The filter coefficients, b_j, are referred to as the jth feedforward taps

6.10.7 Infinite Impulse (IIR) Response Filters

The infinite Impulse Response or IIR filter has an impulse response which is infinite in duration[89]. The transfer function for an IIR filter is of the form:

$$H(z) = \frac{p_0 + p_1 z^{-1} + p_2 z^{-2} + \ldots + p_M z^{-M}}{d_0 + d_1 z_{-1} + d_2 z^{-2} + \ldots + d_N z^{-N}} \tag{6.219}$$

In terms of a difference equation, it is defined by a set of recursion coefficients and the following equation:

$$y[n] = -\sum_{k=1}^{M} a_k y[n-k] + \sum_{k=1}^{N} b_k x[n-k] \tag{6.220}$$

The transfer function for IIR filters for which m

$$\tag{6.221}$$

where a_k is the kth feedback tap which depends on previous outputs, M is the number of feedback taps and N is the number of feedforward taps. The a_k coefficients are referred to as the recursive or "reverse" coefficients, and the b_k coefficients are called the "forward" coefficients. Therefore unlike its FIR counterpart, the IR filter output is a function of the previous outputs and inputs which is a characteristic common to all IIR structures and is responsible for the infinite duration of the impulse response. Note that if $a_k = 0$, then Equation (6.220) becomes identical to Equation (6.215).

6.10.8 Digital Filter Blocks (DFBs)

PSoC3/5 have support for filter components called Digital Filter Blocks (DFBs)[90] that have two separate filtering channels. The DFB has its own multiplier and accumulator which supports 24-bit x 24-bit multiplication and a 48-bit accumulator. This combination is used to provide a Finite Impulse Response (FIR) filter with a computation rate of approximately one FIR tap for each clock cycle.

The DFB features include:

- data alignment support options for I/O samples,
- one interrupt and two DMA request channels,
- three semaphore bits programmatically accessible,
- two usage models for block operation and streaming,
- cascading of 2-4 stages per channel with each stage having their own filter class, filter type, window type, # of filter taps[91],center frequency and bandwidth specifications.

and,

- two streaming data channels.

The DFB is implemented as a 24-bit, fixed point, programmable, limited scope, DSP engine as shown in Figure 6.51. The DFB supports two streaming data channels, where programming

[89] Since the IIR filter is a "recursive" filter and has the property that its impulse response is in terms of exponentially decaying sinusoids and therefore infinitely long. Of course, in real world systems at some point the responses fall below the roundoff noise level, and thereafter may safely be ignored.

[90] Only one filter component can be incorporated in a design at a time.

[91] Filter taps are limited to a maximum of 128 taps for each channel.

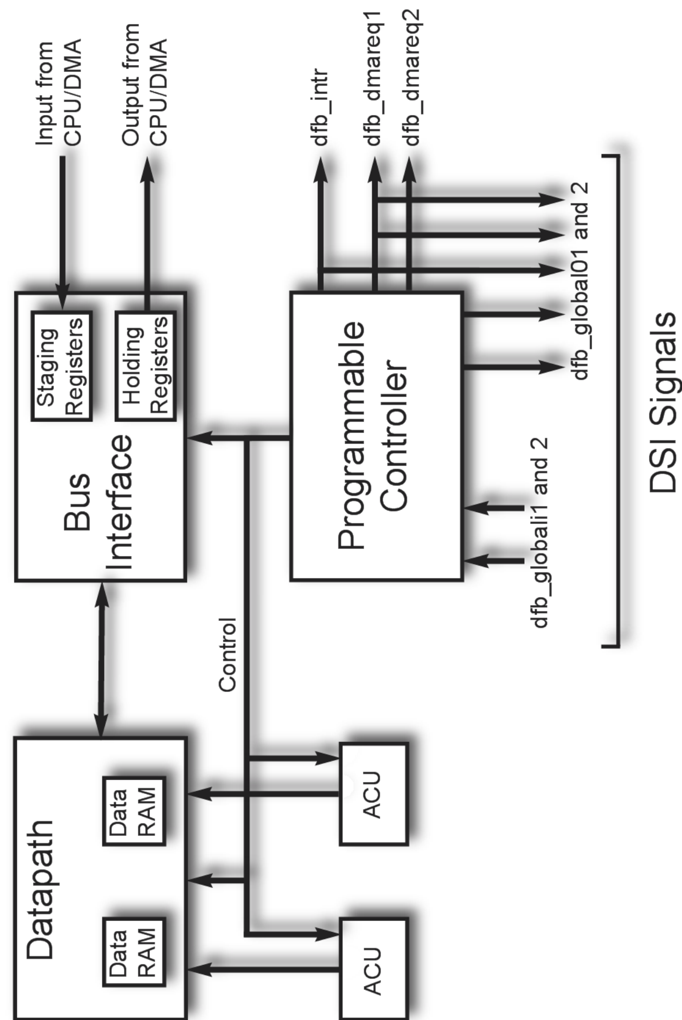


Figure 6.51: The Digital Filter Block Diagram.

instructions, historic data and filter coefficients, and results are stored locally with new periodic data samples received from the other peripherals and blocks through the PHUB interface. In addition, the system software can load sample and coefficient data in/out of the DFB data RAMs, and/or reprogram them for different operations in block mode. This allows for multi-channel processing, or deeper filters than supported in local memory. The block provides software configurable interrupt (DFB_INTR_CTRL) and two DMA channel requests (DFB_DMA_CTRL). Three semaphore bits are available for system software to interact with the DFB code (DFB_SEMA).

Data movement is typically controlled by the system DMA controller, but it can also be moved directly by the CPU. The typical usage model is for data to be supplied to the DFB over the system bus, from another on-chip system data source such as an ADC. The data typically passes through main memory, or is directly transferred through DMA. The DFB processes this data and passes the result to another on-chip resource, such as a DAC, or main memory, via DMA on the system bus. Data movement in, or out of, the DFB is typically controlled by the system DMA controller, but can be moved directly by the CPU.

The DFB consists of subcomponents, viz., a

1. controller,
2. bus interface,
3. Datapath,

and,

4. an Address Calculations Units (ACUs).

The DFB's programmable controller has three memories[92] and a relatively small amount of logic and consists of a RAM-based state machine, RAM-based control store, program counters and "next state" control logic, as shown in Figure 6.51. Its function is to control the address calculation units and the Datapath, and to communicate with the bus interface to move data in, and out of, the Datapath.

The Datapath subblock is a 24-bit fixed point, numerical processor containing a Multiply and Accumulator (MAC), a multi-function Arithmetic Logic Unit (ALU), sample and Coefficient/ Data RAM (Data RAM is shown in Figure 30-1) as well as data routing, shifting, holding, and rounding functions. The Datapath block is the calculation unit inside the DFB.

The addressing of the two data RAMs in the Datapath block are controlled by the two (identical) Address Calculation Units (ACUs), one for each RAM. These three sub-functions make up the core of the DFB block and are wrapped with a 32-bit DMA-capable AHB-Lite Bus Interface with Control/Status registers.

These three sub-functions make up the core of the DFB block and are wrapped with a 32-bit DMA-capable AHB-Lite Bus Interface with Control/Status registers. The Controller consists of a RAM-based state machine, a RAM-based control store, program counters, and next state control logic. Its function is to control the address calculation units and the Datapath, and to communicate with the bus interface to move data in and out of the Datapath.

Proprietary assembly code and an assembler allow the user to write assembly code to implement the data transform the DFB should perform. Alternatively, a "wizard" is provided to facilitate digital filter design for both FIR and FII filters. The wizard allows the designer to set either one or two data stream channels, referred to as Channel A and Channel B, of a filter component that is passing data either in, or out, using DMA transfers or register writes, via

[92]The code that embodies the data transform function of the DFB resides in these memories.

firmware. and an integral co-processor. The filter has 128 taps that determine the frequency responses of the filter. Either channel can be configured to produce an interrupt in response to receiving a data-ready event which in turn enables the interrupt output.

6.10.9 PSoC3/5 Filter Wizard

PSoC Creator includes a powerful wizard for configuring digital IIR and FIR filters. The wizard shown in Figure 6.59 provides a graphical representation of the filter by displaying the response

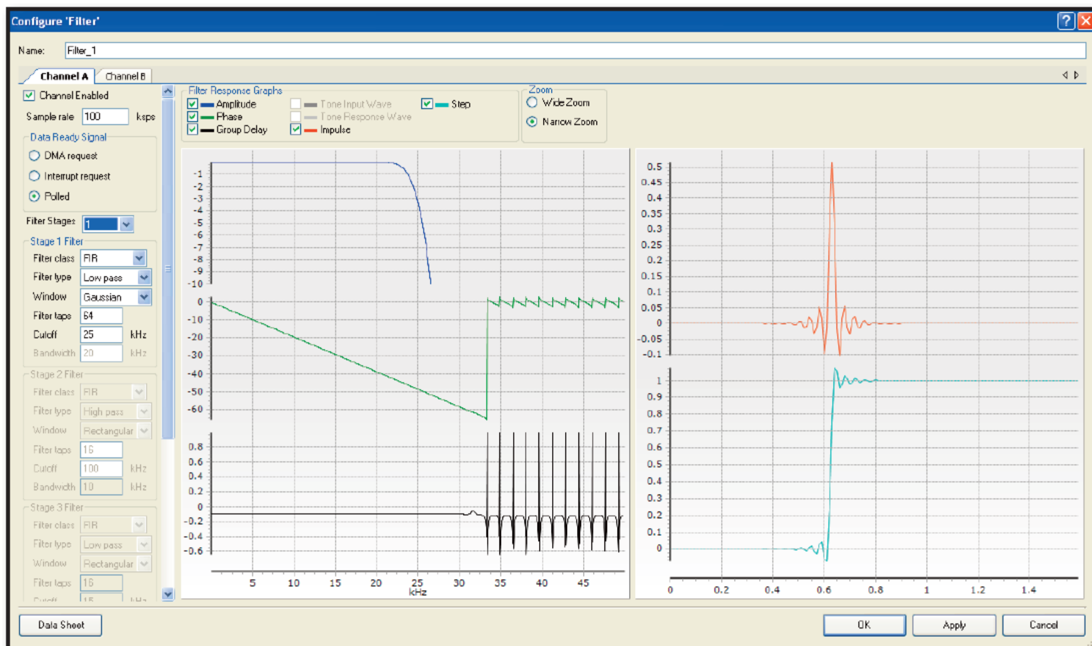


Figure 6.52: PSoC3/5's Filter Configuration Wizard.

for each of the following in a color-coded format:

1. **Amplitude** - gain is displayed graphically as a function of frequency.
2. **Phase** - phase is displayed graphically as a function of frequency.
3. **Group Delay** - occurs when phase as a function of frequency is non-linear. If the frequency components of a signal are propagated through a device with no Group Delay, then the components experience the same time delay. If some frequencies are traveling faster than others then there is group delay and distortion results.
4. **Impulse Response** - the impulse response completely characterizes the filter.
5. **Tone Input Wave** - an input signal (sinusoid) is shown graphically for a bandpass filter
6. **Tone Response Wave** - the response to the tone input into a bandpass filter is shown graphically.

Several different types of "windows"[93] are supported that offer various combinations of band width transition, pass band ripple and stop band attenuation characteristics:

[93]Windows, or more properly, windows functions are functions that are defined as zero outside of a specified interval.

1. **Rectangular** - large pass band ripple, sharp roll-off and poor stop band attenuation. Rarely used because of the large ripple effect as a result of the Gibb's[94] phenomenon.

$$w(n) = 1 \tag{6.222}$$

2. **Hamming**[95] - smoothed pass band, wider transition band and better stop band attenuation than

$$w(n) = 0.54 - 0.46\, cos\left[\frac{2\pi n}{(N-1)}\right] \tag{6.223}$$

3. **Gaussian** - Wider transmission band, but greater stop band attenuation and smaller stop band lobes than Hamming.

$$w(n) = exp\left[-\frac{1}{2}\left(\frac{\frac{2n}{N-1}-1}{\sigma}\right)^2\right] \qquad \sigma \leq 0.5 \tag{6.224}$$

4. **Blackman** - provides a steeper roll-off than its Gaussian counterpart, but similar stop band attenuation although larger lobes in the stop band.

$$w(n) = a_0 - a_1 cos\left[\frac{2\pi n}{(N-1)}\right] + a_2 cos\left[\frac{4\pi n}{(N-1)}\right] \tag{6.225}$$

Windows are often employed to deal with undesirable behavior taking places at the edges of a filter's characteristics. They are introduced either by multiplication in the time domain, or by convolution in the frequency domain.

The sample data rate is the design rate but the operational rate is determined by the data source driving the filter. There are no decimation or interpolation stages and therefore the sample rate is the same throughout each channel. The maximum sample for a channel is:

$$f_{sMax} = \frac{Clk_{bus}}{ChannelDepth + 9} \tag{6.226}$$

where Clk_{bus} is the bus clock speed and Channel Depth is the total number of taps used for a given channel.

If both channels are used then Equation (6.226) becomes:

$$f_{sMax} = \frac{Clk_{bus}}{ChannelDepth_A + ChannelDepth_B + 19} \tag{6.227}$$

The filter component can alert the system of availability of data either via a DMA request that is specific to each channel. or through an interrupt request shared between the two channels, or the status register can be polled to check for new data ready. Although the output holding register is doubled buffered, it is important to remove the data from the output before it s overwritten. Each channel can have as many as four cascaded[96] stages assuming that sufficient resources are available. The cutoff frequency parameter is used to set the 'edge" of the passband frequencies for Lowpass, Highpass and Sinc4 filters

[94] The best known Gibb's phenomenon is the so-called ringing effect observed and the leading and trailing edges of a square wave.

[95] This should not be confused with Hanning Window which is defined as $w(n) = 0.5(1 - cos[(2\pi n)/M)$ for $0 \leq n \leq M$ and zero for all other values of n. It is sometimes referred to as the "raised cosine" window.

[96] Cascading of stages refers to the use of multiple filters which are interconnected so that the output of one filter stage becomes the input of another filter stage.

The filters center frequency is defined as the arithmetic mean of the upper and lower cutoff frequencies for the bandpass stop and pass filters:

$$f_c = \frac{f_u + f_l}{2} \tag{6.228}$$

and the bandwidth (BW) is defined as:

$$BW = f_u - f_l \tag{6.229}$$

The wizard allows the designer to:

- view graphical representations of the frequency response, phase delay, group delay, and impulse and step responses,
- select from 1 to 4 filter stages,
- zoom in and out to provide a view of the filter's responses with either a linear or logarithmic frequency scale, respectively, over a frequency range from DC to the Nyquist frequency.
- enable or disable the filter cascade's response to a positive step function,
- enable or disable the filter cascade's response to to a unit impulse,
- select a tone[97] input wave at the center frequency of the bandpass
- implement both FIR and FII filters

This function is capable as acting as a filter that

6.10.9.1 Sinc Filters

The normalized sinc function, shown in Figure 6.53 is defined as:

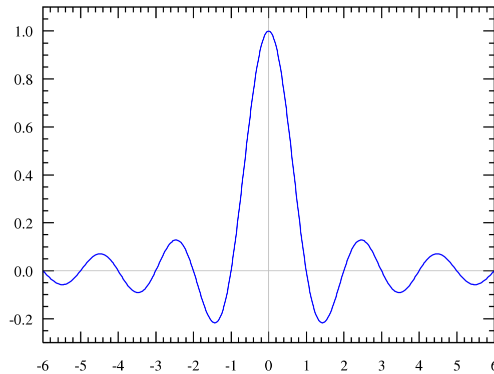


Figure 6.53: The "normalized"sinc function.

$$sinc(x) = \frac{sinc(\pi x)}{\pi x} \tag{6.230}$$

and can be used as the basis for a sinc-based lowpass digital filter with a linear phase characteristic. A related function, shown in Figure **??** called the rect function is defined as

$$rect(t) = \sqcap(t) = \begin{cases} 0 & \text{if } |t| > \frac{1}{2} \\ \frac{1}{2} & \text{if } |t| = \frac{1}{2} \\ 1 & \text{if } |t| < \frac{1}{2}. \end{cases} \tag{6.231}$$

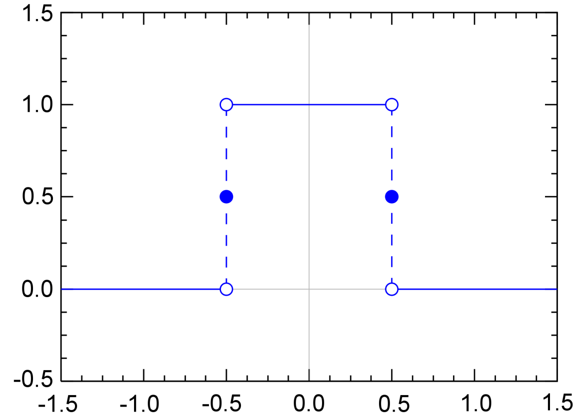


Figure 6.54: The rectangular or Rect function,

These two functions are related by the fact that Fourier Transform of the rect function is the sinc function and the inverse Fourier Transform of rect is the sinc function, i.e., given that the Fourier Transform and its inverse are defined by

$$\mathcal{F}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt \tag{6.232}$$

and

$$\mathcal{F}^{-1}(t) = \int_{-\infty}^{\infty} F(f)e^{i\omega t}d\omega \tag{6.233}$$

It follows that

$$\mathcal{F}(\omega) = \int_{-\infty}^{\infty} \frac{sin(\omega t)}{\pi\omega}dx = \mathrm{rect}(\omega) \tag{6.234}$$

$$\mathcal{F}^{-1}(t) = \int_{-\infty}^{\infty} rect(t)e^{i2\pi ft}df = \frac{1}{\sqrt{2\pi}}sin\left[\frac{\omega t}{2\pi}\right] \tag{6.235}$$

Thus the impulse function for a rect function in the frequency domain is the sinc function in the time domain and conversely a rect function in the time domain is mapped to a sinc function in the frequency domain. If the sinc function is "convolved" with with an input signal, theoretically an ideal lowpass filter could be realized. However, the fact that the sinc function extends to $\pm\infty$ presents a problem. One approach is to just cutoff all of the points on the sinc curve beyond a certain point and the examine the Bode plot to determine the resulting effect.

This technique can lead to undesirable ripple in the passband and outside of the passband as a result of the steepness of the truncated ends of the sinc function. and outside the passband. Another possibility is to employ so-called window functions, e.g., the Blackman or Hamming windows, that are multiplied by the truncated sinc function. This results in steeper roll-off and less ripple in the passband and stop bands. If stopband attenuation is a major concern the Blackman window should be employed but it will result in some degradation of the roll-off. If roll-off is the primary concern then the Hamming window is the better choice.

[97]The tone is a sine wave whose frequency is the center frequency of the bandpass filter.

6.11 Data Conversion

Embedded systems by necessity are required to carry out various operations with digital and analog data. Input of digital data can be provided by external communications channels, digital sensors or other digital sources. Analog inputs often have to be converted to equivalent digital data to permit numerical and logic processing, storage in memory, etc. In addition, digital data may have to be converted to its analog equivalent to allow it to control external devices such as motors, other actuators, etc., or for other purposes. Thus analog-to-digital and digital-to-analog conversion is an important capability for many embedded systems.

6.12 Analog to Digital Conversion

Since the world is essentially analog, it is not surprising to find that embedded systems make extensive use of analog-to-digital and digital-to-analog techniques in order to get the real world into the computational domain. Although different architecturally one is rarely to be found without the other nearby. Arguably much of the world at increasingly finer grain levels appears not continuous but discrete, embedded systems are typically dealing with an environment full of continuous sources some of which of necessity must be monitored by the embedded system. Digital-to-analog converters, of necessity, must bridge the gap between the discrete-value environment of the digital domain and that of the continuous value[98], to allow embedded systems to communicate and to some extent control external processes.

Analog-to-digital converters are presented with continuous-valued inputs, continuous-time signals and expected to provide digital equivalents in the form of discrete values and discrete times to what has become ever increasing degrees of resolution.

6.13 Basic ADC Concepts

There are a number of important and very fundamental concepts involved in the use and deployment of analog-to-digital converters including:

- *Aliasing* is the introduction of spurious signals as a result of sampling at at a rate below the Nyquist criteria, i.e., a rate less than the high frequency component in the input signal.
- *Resolution* refers to the number of quantization levels of an ADC, e.g., an 8-bit ADC has a resolution of 256.
- *Dither* refers to the addition of a small amount white noise to low level, periodic signals signal prior to conversion by an ADC. The addition of noise as shown in Figures 6.55 and 6.56. Note that in Figure 6.56 the noise added before the analog-to-digital conversion is subtracted at the output and is referred to as "subtractive dithering".

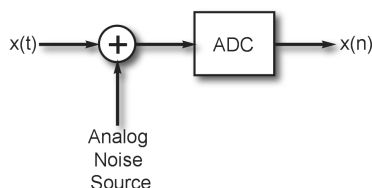


Figure 6.55: Simple dither application using a analog noise source.

[98]Often after some lowpass filtering.

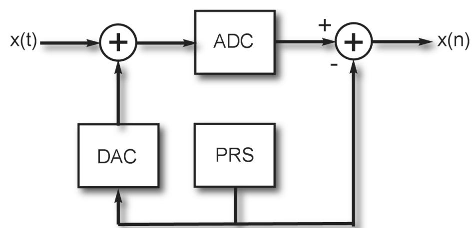


Figure 6.56: Dither application using a digital noise source

- *Sampling rate* refers to the number of samples per unit time. It is usually chosen to be at least twice the highest frequency component in the signal being sampled.
- *Oversampling* is the process of collecting more samples than would other wise be needed to accurately reproduce a signal that reduces the in-band quantization noise by a factor equal to the the square root of the oversampling ratio, e.g., reducing the noise by a factor of two, increases the effective processing gain of 3dB. Keep in mind that we are only talking about broadband noise here. Other sources of noise and other errors cannot simply be removed by oversampling.
- *Undersampling* is a technique used in conjunction with ADCs that allows an it to function as a mixer. Thus a high frequency signal can be input and the output of the ADC is a lower frequency. However, this technique does require digital filtering in order to recover the signal of interest.
- *Decimation* refers to the use in oversampling applications and the subsequent discarding of samples after conversion in a way that does not significantly alter the accuracy of the measurement.
- *Quantization* is the process of breaking a continuous signal into discrete samples or quantum.
- *Quantization error* is defined as the arithmetic difference between an actual signal and its quantized, digital value.
- *Dynamic range* is the range between the noise floor and the maximum output level.

6.13.1 Delta-Sigma ADC

The delta-sigma[99] modulator emerged from the early development of pulse code modulation technology and as originally developed in 1946. However, it remained dormant until 1952 when it appeared again in various publications including a related patent application. Its appeal was the fact that it could offer increased data transmission since it could transmit the changes, i.e., "deltas", in value between consecutive samples, instead of transmitting the actual values of the sample. A comparator is used as a one-bit ADC and the output of the comparator was then converted to an analog signal, using a 1-bit DAC, the output of which was then subtracted from the input signal after it had passed through an integrator. Delta sigma modulators rely on techniques known as "over-sampling" and noise-shaping in order to provide the best performance. A greatly simplified example of a Delta-Sigma modular is shown in Figure 6.57

In this example the input voltage, v_i, is added to the output of the single-bit, digital-to-analog converter and the sum is then integrated and the output applied to input of the comparator. The output of the comparator is either high, or low, i.e., one or zero, depending on whether the output

[99]The literature refers to both "delta-sigma" and "sigma-delta" modulators. Purists argue that the proper name is delta sigma since the signal passes through the delta phase prior to the sigma phase. Not withstanding, the distinction is comparable to that between Tweedle Dee and Tweedle Dum.

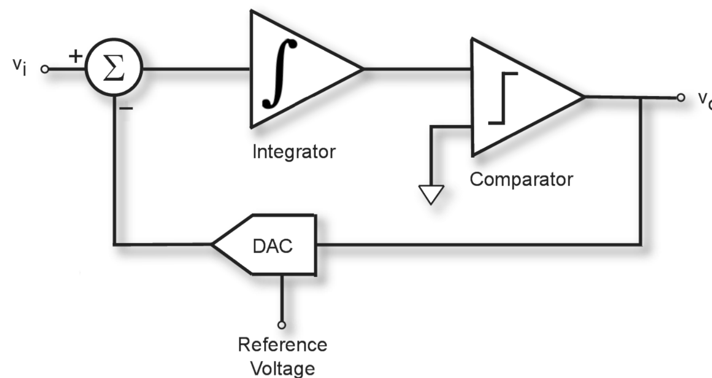


Figure 6.57: An example of a First-Order, ΔΣ modulator.

of the integrator output is ⩾ zero or negative, respectively. The output of the comparator is then provided to the input of the DAC and the process is repeated. The output of the DAC is ± the reference voltage.

Example 6.7 - As a quantitative, illustrative example consider the following:

Let the reference voltage be +/- 2.5 volts and assume that the input voltage
is 1 volt. Then when the process begins the output of the DAC is zero so
that 1+ 0 = 1 which when integrated becomes 1 and the comparator outputs
a one which is then applied to the DAC with the result that the DAC output
becomes 2.5 volts. When this is summed with the input the total is -1.5 vo
t which is integrated to produce an output from the integrator Of -.5 volt.
The comparator outputs a zero and the process continues.

Table 6.13 shows the results of the process outlined in Example 6.7. Figure 6.58 shows the configuration of a 2nd-order Delta Sigma modulator. A graphical representation of the various associated signals of a Delta Sigma modulator with a sinusoidal input is shown in Figure 6.60.

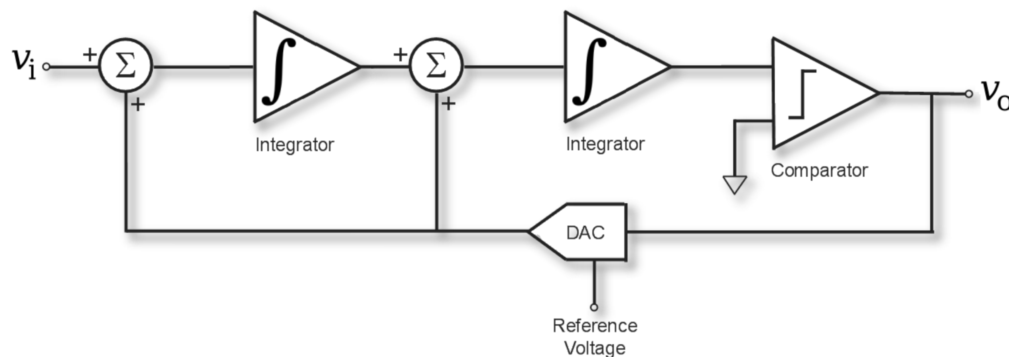


Figure 6.58: An example of a Second-Order, Delta Sigma modulator.

6.13.2 PSoC3/5 Delta Sigma Converter

The architecture of PSoC3/5 includes a very high SNR/resolution Delta Sigma ADC that employs oversampling, noise shaping, averaging and decimation. A Delta Sigma Analog-to-Digital Converter (ADC) has two main components: a modulator and a decimator. The modulator converts the analog input signal to a high data rate (oversampling), low resolution (usually 1 bit) bitstream, the average value of which gives the average of the input signal level. This bitstream is passed through a decimation filter to obtain the digital output at high resolution and lower data rate. The decimation filter is a combination of downsampler and a digital low pass (averaging) filter that averages the bitstream to get the digital output.

Features of the PSOC3/% Delta Sigma converter include:

- 12- to 20-bit resolution
- An optional input buffer with RC lowpass filter
- Configurable gain rom 0.25 to 256
- Differential/single-ended inputs
- Gain ad offset correction
- Incremental continuous modes
- Internal and external reference options
- Reference filtering for low noise

The PSoC3/5 uses a 3rd-order modulator with a high impedance front end buffer followed by a bypassable RC filter. The modulator sends out a high data rate bitstream in thermometric format. The output of the modulator is passed on to the analog interface that converts the thermometric output to twos complement (4 bit) and passes it on to the decimation filter. The decimation filter converts output of the modulator into a lower data rate, high resolution output.

The input impedance of the modulator is too low for some applications and therefore higher input impedance, low noise, independent buffers have been provided for each of the differential inputs. These buffers can be bypassed/powered down by setting DSM_BUF0[1], DSM_BUF1[1] and/or $DSM_BUF0[0]$, $DSM_BUF1[0]$, respectively. The buffers have adjustable gains $(1, 24or8)$ determined by $DSM_BUF1[3:2]$. The buffers can operate either in a level shifted mode to allow the input level to be shifted above zero and rail-to-rail when the input is rail-to-rail. input to the buffers can be from analog globals, analog locals, the analog mux bus, reference voltages and V_{ssa}.

The PSoC3/5 Delta Sigma modulator consists of three active, OpAmp-based integrators (INT1, INT2 and INT3), an active summer, a programmable quantizer and switched-capacitor feedback DAC as show in Figure 6.59. The three active integrators function as a 3rd-order modulator whose transfer function together with the quantizer provide highpass noise shaping. Increasing the order of the modulator, improves the highpass filter response and lowers noise present in the signal frequency band. The three integrators and quantizer stages are followed by an active summer. The analog input and the output of all three opamp stages are then summed. The summer output is quantized by a quantizer that is programmable to output 2, 3, or 9 levels. The DAC connects the quantizer output back to the first stage OpAmp input. It is this feedback DAC that ensures that the average of the quantizer output is equal to the average input signal level.

The quantization level can be set as 2,3, or 9. The lowest level provides the best linearity and the highest provides the best SNR. The number of quantization levels is configured in

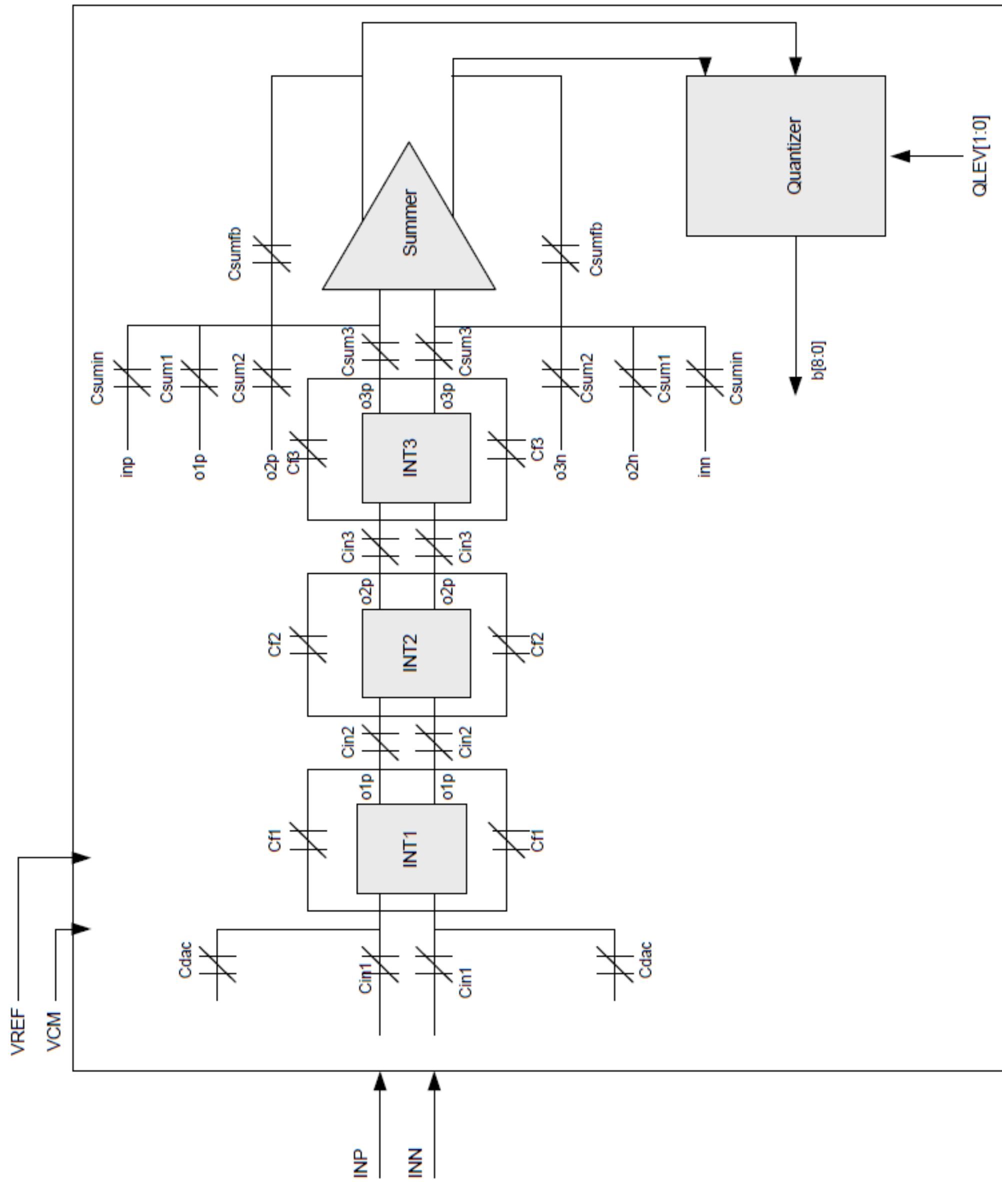


Figure 6.59: Delta Sigma modulator block diagram.

DSM_CR0[1:0] register bits. The quantizer output is stored in the register DSM_OUT1. The quantizer outputs data in a format referred to as the thermometric[100] and is illustrated by the pattern of output levels shown in Table 6.12.

Table 6.12: Quantizer Output Data

Level	Quantizer Output Data
2 Level Quantizer	
Level 1	00000000
Level 2	11111111
3 Level Quantizer	
Level 1	00000000
Level 2	00001111
Level 3	11111111
9 Level Quantizer	
Level 1	00000000
Level 2	00000001
Level 3	00000011
Level 4	00000111
Level 5	00001111
Level 6	00011111
Level 7	00111111
Level 8	01111111
Level 9	11111111

6.13.3 Successive Approximation Register ADC

As shown in Figure 6.61, there are four components that make up a Successive Approximation Register ADC (SAR):

1. A voltage DAC that converts the SAR output to an analog voltage which can then be used to compare with the input voltage.
2. A comparator that compares the analog input to the DAC output.
3. A successive approximation register which, based on the output of the comparator, provides the appropriate input to the DAC.
4. A Track and hold circuit that holds an input value constant during the conversion at which point it loads another sample of the input.

PSoC3/5 have 8 bit voltage DACs and therefore the SAR is limited to 8-bit resolution. Although PSoC3/5 do not currently have explicit support for a SAR, it is possible to construct a SAR based on the resources that are available in PSoC3/5[30]. SAR logic must set or reset a given bit based on the output of the comparator.

In a typical implementation, this operation is repeated 8 times until the SAR generates an “end of conversion” signal and latches the data. The VDAC can accept data from the DAC bus and therefore data can be transferred directly from the SAR to VDAC without incurring CPU overhead. However, it is necessary to generate a strobe using SAR logic when data is available on the DAC bus so that the VDAC will produce a corresponding output voltage.

[100]In the thermometric format, the number of ones increases from LSB to MSB as the quantization level increases.

Table 6.13: Delta Sigma Example Output.

Iteration	Input	DAC Output	Sum	Integrator Output	Comparator Output	Mean Voltage Output
0	1	0	0	0	0	0
1	1	0	1	1	1	2.5
2	1	2.5	-1.5	-0.5	0	0
3	1	-2.5	3.5	3	1	0.83
4	1	2.5	-1.5	1.5	1	1.25
5	1	2.5	-1.5	0	1	1.5
6	1	2.5	-1.5	-1.5	0	0.83
7	1	-2.5	3.5	2	1	1.07
8	1	2.5	-1.5	0.5	1	1.25
9	1	2.5	-1.5	-1	0	0.83
10	1	-2.5	3.5	2.5	1	1
11	1	2.5	-1.5	1	1	1.14
12	1	2.5	-1.5	-0.5	0	0.83
13	1	-2.5	3.5	3	1	0.96
14	1	2.5	-1.5	1.5	1	1.07
15	1	2.5	-1.5	0	1	0.94
16	1	2.5	-1.5	-1.5	0	1.03
17	1	-2.5	3.5	2	1	1.11
18	1	2.5	-1.5	0.5	1	0.92
19	1	2.5	-1.5	-1	0	1
20	1	-2.5	3.5	2.5	1	1.07
21	1	2.5	-1.5	1	1	0.91
22	1	2.5	-1.5	-0.5	0	0.98
23	1	-2.5	3.5	3	1	1.04
24	1	2.5	-1.5	1.5	1	1

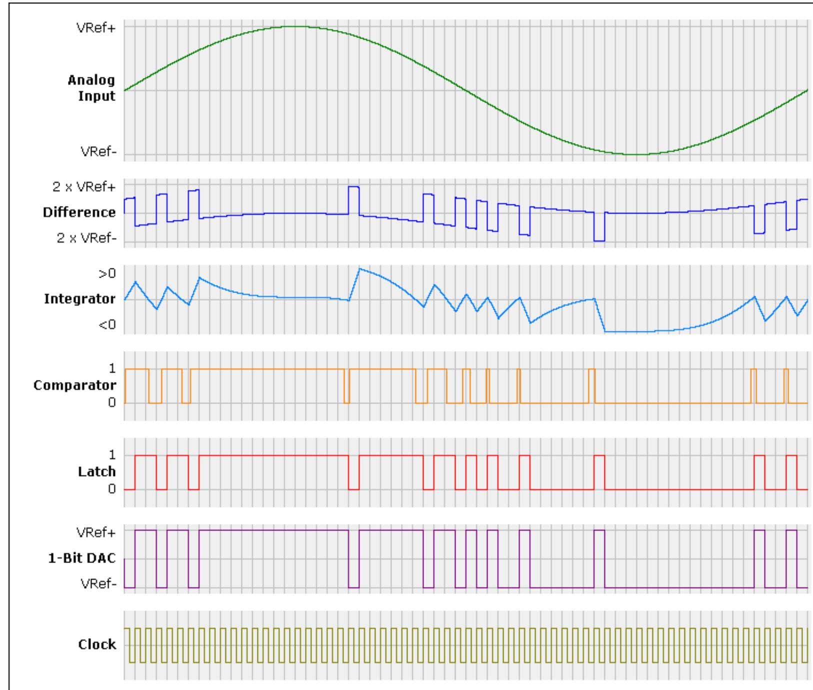


Figure 6.60: First order, Delta-Sigma modulator signals with sinusoidal input.

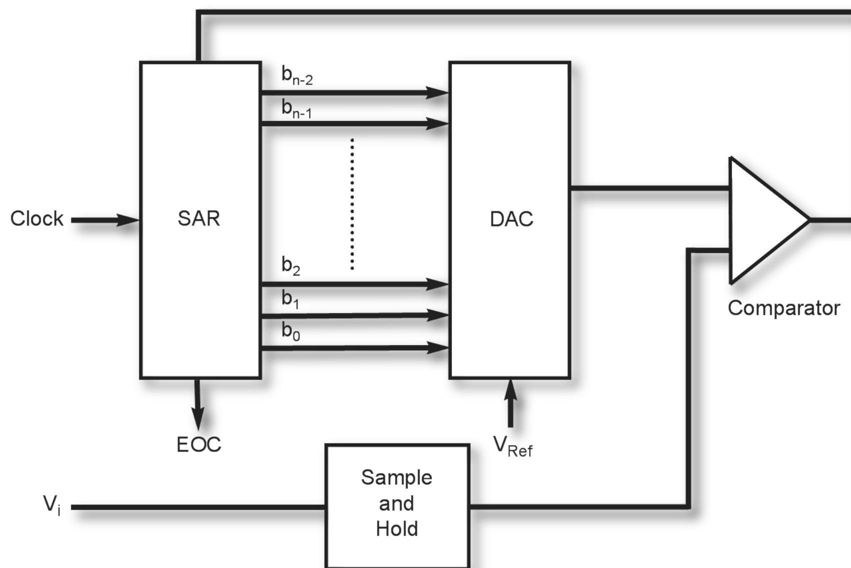


Figure 6.61: A schematic diagram of a SAR ADC.

The limitation on speed of conversion are set primarily by:

1. The speed with which the comparator is able to resolve differences between v_{i} and the output of the DAC.
2. The DAC's settling time which is a function of the settling time for the MSB.
3. Overhead introduced by the latency of the various components of the SAR ADC. Such factor include the sample and hold acquisition time, sample and hold settling time, EOC recognition time by the CPU, etc.

The change of state of the comparator signals that the binary representation of the input signal has been found and that the data can then be accessed programmatically.

6.13.4 Analog MUX

A multiplexer, or Mux, is a device that allows one, or more, inputs to be switched, and/or combined, programmatically to one, or more, outputs. These inputs/outputs may be either digital or analog, respectively. Typically muxes are controlled by digital signals consisting of one or more binary inputs. Depending on the "address" represented by the the binary inputs one of the input sources is connected to the output of the mux, as shown in Figure 6.62.

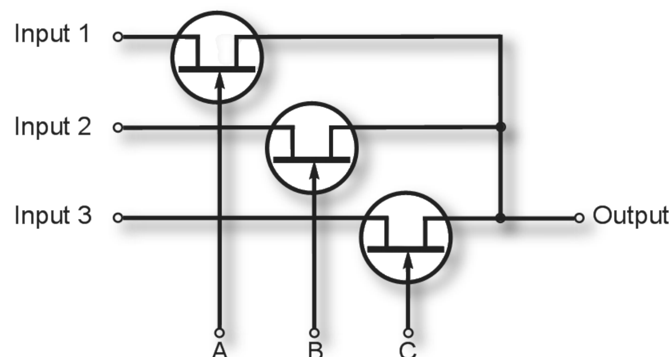


Figure 6.62: A simple analog multiplexer.

The analog multiplexer, or AMux, is a passive device that can combine multiple analog signals, or multiple pairs thereof, in a single signal, or pair, to allow the output signal to be routed to a single input of some other component. The AMux also allows multiple input simultaneous input connections to be routed to a single connection. The AMux employs individual switches that connect blocks to analog busses, and analog busses to pins.

Unlike most hardware multiplexers, the AMux is a collection of independent switches that are controlled by firmware, and not by hardware. This makes AMux much more flexible than other types of multiplexers because it allows more than one signal at time to be connected to the common output signal. Note that in the "Differential Mode" the firmware will not allow the differential signals to be connected to each other and instead treats such cases as two parallel multiplexers controlled by the same signal.

6.13.4.1 Allowable Input/Output Connections

There are various types of input/output connections that are supported by the AMux:

- aN (Analog) - the AMux supports 2 - 32 analog inputs, inclusive.
- bN (Analog)[101] - the paired inputs (aN,Bn) are only used when the Mux Type parameter is set to “Differential”.
- y (Analog) - This is a required connection and is the output of the AMux.
- x (Analog) - the “x” signal is the output connection when using the AMux in a differential mode. Its output is determined by the “void AMux_Select(void)” function.

In setting up an AMux, certain parameters must be set in order to achieve the desired configuration, i.e.:

- **Channels** - this parameter specifies the number of single or paired inputs and may have a value of 2-32, inclusive.
- **MuxType** - this parameter determines whether a single input per connection (Single)[102]. or dual input per connection (Differential) is to be used. If two, or more, input signals have different signal references, the “Differential” mode must be used. This mode is often employed when the output of the mux is connected to an ADC with a differential input.

6.13.4.2 The AMux API

The application programming interface or API for the AMux provides programmatic access to various routines that allow the designer to configure the AMux. By default, PSoC Creator assigns the instance name “AMux_1” to the first instance of AMux. The API function calls for AMux are:

- **void AMux_n_Start** - disconnects all channels.[103]
- **void AMux_n_Stop** - disconnects all channels.[104]
- **void AMux_n_Select(uint8 chan)** - disconnects all other channels and then connects the selected channel (chan) signal.
- **void AMux_n_FastSelect(uint8 chan)** - disconnects the last connection made with either FastSelect or Select function calls and then connects the “init8 chan”.[105]
- **void AMux_n_Connect(uint8 chan)** - connects the given channel to the common signal without affecting any previous channel connection.
- **void AMux_n_Disconnect(chan)** - disconnect only the specified channel from the output.
- **void AMux_n_DisconnectAll(void)** - disconnect all channels.

[101] This type of I/O may be hidden on the symbol under the conditions listed in the description of that I/O.
[102] Single refers to cases in which each input signal is referenced with respect to a common signal, e.g., V_ssa
[103] With respect to AMux function calls, there are no return values, side effects or parameters to specify, unless otherwise noted.
[104] The Stop API call is not really required but is provided for “compatibility’ reasons.
[105] If the Connect function was used to select a channel prior to calling FastSelect, the channel selected will not be disconnected which is useful when parallel signals need to be connected.

6.13.5 Analog/Digital Virtual MUX

PSoC3/5 have support for analog/digital "virtual" muxes which are analogous to hardware muxes in that they connect a selected input to an output. However, unlike their hardware counterpart, virtual muxes are can not be dynamically controlled. They can be used at the schematic level to chose from a variety of different sources, e.g., to select from a number of different clock sources. The actual connection to be made is selected at build time. The default number of inputs[106] is two with a maximum of sixteen and the selected input Virtual muxes do not consume any resources but merely connect a pre-defined input to the output.

6.13.6 PSoC3/5 Delta Sigma ADC (ADC_DelSig)

The Delta-Sigma ADC provided in PSoC3/5 supports resolutions from 8-20 bits, continuous mode operation, an adjustable sample rate (10-375,000 sps), a high input impedance input buffer and selectable input buffer gain all of which makes it ideal for sampling signals over a wide range of frequencies. Whether used to sample input from strain gauges, thermocouples or other forms of high precision but low amplitude sensors, the ADC_DelSig is designed to spread the quantization noise across a sufficiently wide spectrum to allow it to be moved out of the input signal's bandwidth and the filtered by a lowpass filter.

The Delta-Sigma ADC is an inherently three terminal device with an optional fourth and fifth pin for a start of conversion (SOC)which occurs as a result of the presence of a rising edge and an external clock source, respectively. The other three pins are positive input, negative input and end of conversion (EOC). This positive input is used for a positive analog signal input to the ADC_DelSig. The conversion result is a function of the positive minus the voltage reference, which is either negative or V_{ssa}.[107]

The ADC_DelSig's negative input functions as the reference input and the result of a conversion is a function of the positive input minus the negative input. If the ADC_INPUT_Range option is selected for this device, then the following modes are available:

0.0 +/- 1.024V (Differential) -Input +/- Vref
0.0 +/- 2.048V (Differential) -Input +/- 2Vref
0.0 +/- 0.512V (Differential) -Input +/- Vref/2
0.0 +/- 0.256V (Differential) -Input +/- Vref/4

User definable parameters for the ADC_DelSig include the following:

- **Variable power settings**: - Low, Medium or High.
- **Conversion modes**: Continuous, Fast Filter or FIR.
- **Resolution**: 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 or 20.
- **Input buffer gain**: 1, 2, 4 or 8[108] Start of conversion:
- **Start of Conversion**: Can be initiated at the hardware, or the software level.
- **Conversion Rate**: 10 to 375,000 samples second
- **Clock Source**: External or internal.
- **Input Range**:
  0.0 to 1.024 V(Single-Ended)
  0.0 to 1.024 V(Single-Ended)
  V_{ssa} to V_{dda}, (Single-Ended)

[106] The number of inputs is specified by NumInputTerminals
[107] V_{ssa} is analog ground.
[108] The input buffer gain can also be disabled.

0.0 ± 1.024V (Differential) Negative Input ± V_{ref}
0.0 ± 2.048V (Differential) Negative Input ± 2(V_{ref})
0.0 ± 0.512V(Differential) Negative Input ± $V_{ref}/2$
0.0 ± 0.256V (Differential) Negative Input ± $V_{ref}/4$

The ADC_DelSig consists of three blocks: an input amplifier, a 3rd-order Delta-Sigma modulator, and a decimator as shown in Figure 6.63. The input amplifier provides a high impedance

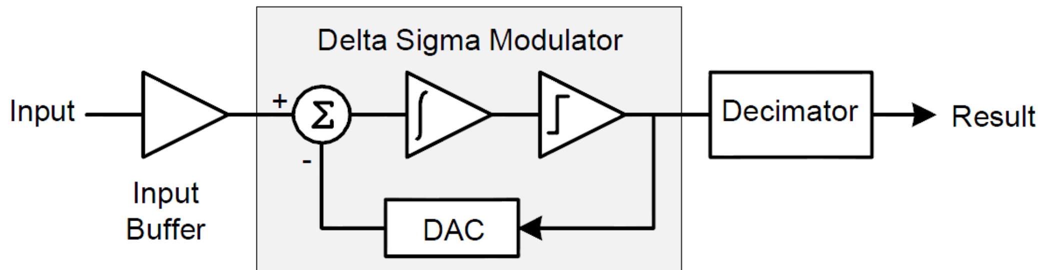


Figure 6.63: The PSoC3/5 ADC_DelSig block diagram.

input and a user-selectable input gain. The decimator block contains a 4 stage CIC decimation filter[109] and a post-processing unit. The CIC filter operates on the data sample directly from the modulator. The post-processing unit optionally performs gain, offset, and simple filter functions on the output of the CIC decimator filter. Decimation is a combination of downsampling and filtering where downsampling refers to the discarding of samples particularly in cases when oversampling[110] is employed. In some cases lowpass filtering is employed prior to to downsampling in order to remain consistent with the Nyquist criteria.

6.14 I/O Pins

In order for an embedded system to interact with the real world there must obviously be provision for hardware connections for both input and output. Microcontrollers have pins for this purpose and for PSoC3/5 the I/O pins are as important, as is their potential configurability. In addition to the pins being configurable at the schematic level, they can also be configured dynamically through program control. At the schematic level the pins component is definable as analog, digital input, digital output or bidirectional with an initial state of high or low.

Implementing embedded systems with PSoC3/PSoC5 typically requires extensive use of various types of I/O pins including:

- Analog Pins
- Digital Input Pins
- Digital Output Pins and

and,

- Digital Bidirectional Pins

PSoC3/5's Pins components can be configured into complex combinations of input, output, bidirectional, and analog I/O connections to provide both on- and off-device signals via physical I/O

[109] Cascaded integrator comb filters (CIC) is a linear-phase FIR filter and are more efficient than conventional FIR filters.
[110] Oversampling is sampling at rates greater than the Nyquist criteria.

pins. A pin component can have 1-64 pins inclusive with a default value of 1 pin. It provides access to external data via an appropriately configured physical I/O pin. It allows electrical characteristics to be associated with one or more pins. These characteristics are then used by PSoC Creator to automatically place and route the signals constrained within the component. Pins can be used from schematics and/or software. To access a Pins component from component APIs, the component must be contiguous and non-spanning. This ensures that the pins are guaranteed to be mapped into a single physical port. Pins components that span ports, or are not contiguous, can only be accessed from a schematic, or with the global per-pin APIs.[111]

An analog Pins component may also support digital input or output connections, or both, as well as, bidirectional connections, e.g., analog with digital input, analog with digital output, analog with digital input and output and analog with bidirectional digital I/O. Digital input pins can also support digital output and analog connections. Digital output pins can support digital input and analog connections. Bidirectional pins can support analog connections. When the Pins component is used in conjunction with an internal reference voltage (Vref) an SIO pin must be used, however, Vref can only be used with another digital connection, i.e. analog pins cannot be used. Digital pins can be used with an IRQ but not an analog pin. There are eight available drive modes for pin as shown in Figure 6.64 .

The drive modes for pins includes:

- Strong drive
- High impedance analog
- High impedance digital
- Open drain drives high
- Open drain drives low
- Resistive pull-up
- Resistive pull-down
- Resistive pull-up and pull-down
- Resistive pull up/pull down

The defaults for drive modes are high impedance for analog, digital and digital I/O and open drain (drives low) for bidirectional. All other

[111]#defines are created for each pin in the Pins component to be used with global APIs.

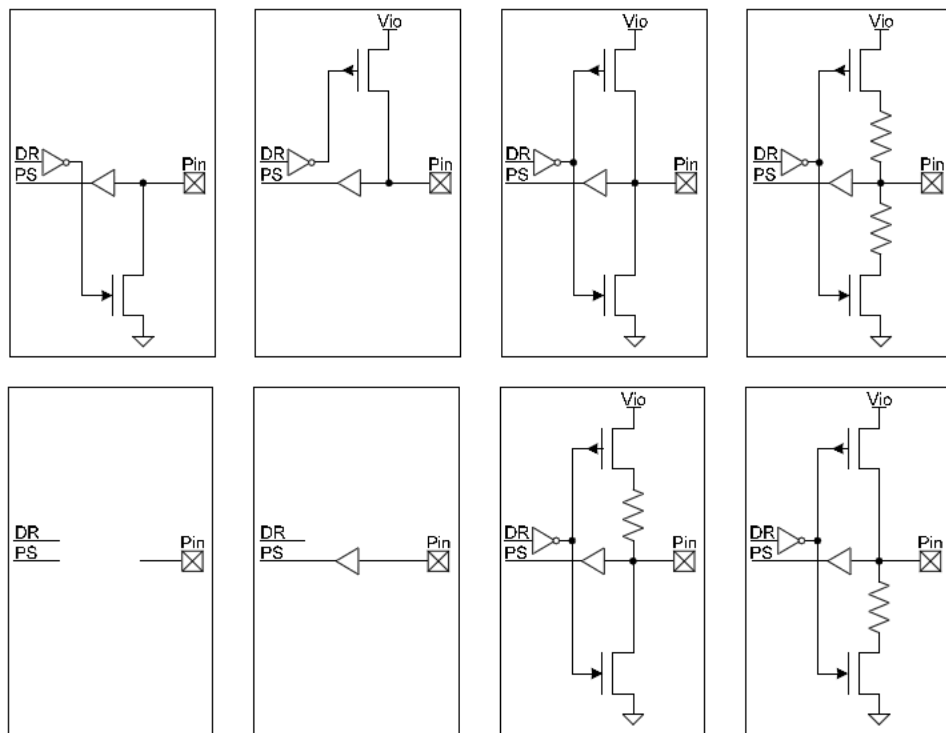


Figure 6.64: PSoC3/5 pin drive modes.

6.15 Digital to Analog Converters (DACs)

Digital to analog converters are an important component in embedded systems that allows the system to covert digital data into its analog equivalent for driving actuators, motors, switches, etc. Selection of a DAC is based on a number of factors, one of which is the desired resolution which determines the number of analog levels that the DAC is capable of producing and the N-bit resolution, where N is the power of 2 representing the number of possible output levels. Dividing the number of levels into the DAC's maximum output voltage determines the voltage step size, i.e.,

$$\text{Output Voltage Step Size} = \frac{\text{Maximum Output Voltage}}{2^N} \tag{6.236}$$

Another factor is the sampling frequency, which refers to the maximum rate of output that the DAC is capable of producing. This is an important consideration when the accuracy, or fidelity, of the output analog signal is of concern. If the Nyquist-Shannon condition is to be met then the DAC must be capable of producing analog values at a rate of at leat twice the highest frequency component to be included in the output. A third concern is the so-called monotonicity of the output from a DAC. In particular, if the output voltage is assumed to be increasing, or decreasing, each actual output step must represent a monotonic increase, or decrease, in the output. Dynamic range is also a consideration and is a function of the resolution of the DAC and the noise floor. Total harmonic distortion is a figure of merit for DACs and may need to be taken into account when selecting a DAC, depending on the application.

PSoC3/5 DACs generate either a voltage or a current output and employ a current mirror[112] architecture in which current is mirrored from a reference source to a mirror DAC. Calibration and value current mirrors are responsible for the 8-bit calibration [DACx.TR] and the 8-bit DAC value. The current is then diverted into the scaler to generate the current corresponding to the DAC value. The DAC value can either be given from the register DACx.D or from 8 lines from the UDB. This selection is made using the DACx.CR1[5] bit. The DAC is strobed to get its output to change for the input code. The strobe control is enabled by the DACx.STROBE[3] bit. The strobe sources for the DAC can be selected from the bus write strobe, analog clock strobe to any UDB signal strobe. This selection is based on the setting in DACx.STROBE[2:0].

- **Voltage (VDAC) Mode** - The current is routed through resistors according to the range and voltage across it provided as output. The output from the DAC is single-ended in both IDAC and VDAC modes.
- **Current (IDAC) Mode** - The two mirrors for the current source and sink provide output as a current source or current sink, respectively. These mirrors also provide range options in the current mode.

The outputs from the PSoC3/5 DACs is single-ended in both IDAC and VDAC modes.

6.15.1 PSoC3/5 Voltage DAC (VDAC8)

The VDAC8 is an 8-bit voltage digital-to-analog converter that can be configured in various ways depending on the application. It is controllable via hardware, software or a combination of both hardware and software. It can be employed as a fixed or programmable voltage source with:

- a CPU, DMA or UDB data source text,

[112] Current mirrors are circuits designed to accurately replicate a reference current referred to as the "golden current source", and sometimes includes scaling of the replicated current. Current mirrors can be thought of as ideal current amplifiers. Golden current sources are expected to be relatively temperature and voltage independent.

- Software, or clock-driven, output strobe,
- Two ranges: 1.020V and 4.096V full scale
- Voltage output

6.15.1.1 Input/Output Connections

When used as a VDAC, the output is an 8-bit digital-to-analog conversion voltage to support applications where reference voltages are needed. The reference source is a voltage reference from the Analog reference block called VREF(DAC). The DAC can be configured to work in voltage mode by setting the DACx.CR0 [4] register.

In this mode, there are two output ranges selected by register DACx.CR0 [3:2].

- 0V to 1.024V
- 0V to 4.096V

Both output ranges have 255 equal steps.

The VDAC is implemented by driving the output of the current DAC through resistors and obtaining a voltage output. Because no buffer is used, any DC current drawn from the DAC affects the output level. Therefore, in this mode any load connected to the output should be capacitive. The VDAC is capable of converting up to 1 Msps. However, the DAC is slower in 4V mode than 1V mode, because the resistive load to Vssa is 4 times larger. In 4V mode, the VDAC is capable of converting up to 250 ksps. The VDAC8's output can be routed to any analog compatible pin on PSoC3/5.

An 8-bit wide data signal, i.e., data[0:7], connects the VDAC8 directly to the DAC Bus. The DAC Bus may be driven by UDB based components, control registers, or routed directly from GPIO pins. Input is enabled by setting the *Data_Source* parameter to "DAC Bus". data[7:0] input should be used when hardware is capable of setting the proper value without CPU intervention and the strobe option should be set as External. For many applications this input is not required, but instead the CPU or DMA will write a value directly to the data register. In firmware, the SetRange() function or directly writing a value to the VDAC8_n_Data register (assuming an nth instance name) should be used.

In strobe input mode, the data is transferred from the VDAC8 register to the DAC on the next positive edge of the strobe signal. If this parameter is set to "Register Write"the pin will disappear from the symbol and any write to the data registers will be immediately transferred to the DAC. For audio or periodic sampling applications, the same clock used to clock the data into the DAC could also be used to generate an interrupt. Each rising edge of the clock would transfer data to the DAC and cause an interrupt to get the next value loaded into the DAC register.

The output voltage is determined by:

$$v_o = 1.020\left[\frac{value}{256}\right]\text{volts} \tag{6.237}$$

or,

$$v_o = 4.096\left[\frac{value}{256}\right]\text{volts} \tag{6.238}$$

depending upon the selected output range and $0 \leq (value) \leq 255$.

6.15.2 PSoC3/5 Current DAC (IDAC8)

When used as an IDAC, the output is an 8-bit digital-to-analog conversion current. This is done by setting the DACx.CR0 [4] register. The reference source is a current reference from the analog reference called IREF(DAC). In this mode, there are three output ranges selected by register DACx.CR0 [3:2].

- $0 - 2.048\,mA$, $8\,\mu$A/bit
- $0 - 256\,\mu A$, 1μA/bit
- $0 - 32\,\mu A$, $0.125\,\mu$A/bit

For each level, there are 255 equal steps of M/256 where M=2.048 mA, $256\mu A$, or $32\mu A$. In the 2.048 mA configuration, the block is intended to output a current into an external $600\,\Omega$ load. The IDAC is capable of converting up to 8 Msps. The user also has the option of selecting the output as either a current source, or a current sink. This is controlled by the DACx.CR1[2] register. This selection can also be made by using a UDB input. UDB control for the source/sink selection is enabled using the DACx.CR1[3] bit. Separate muxes are used for current and voltage modes.

It is possible to achieve a higher resolution current output DAC by summing the outputs of two 8-bit current DACs, each one having a different segment of the input bus for input, as shown in Figure 6.65. The range of the two DACs used partially overlap.

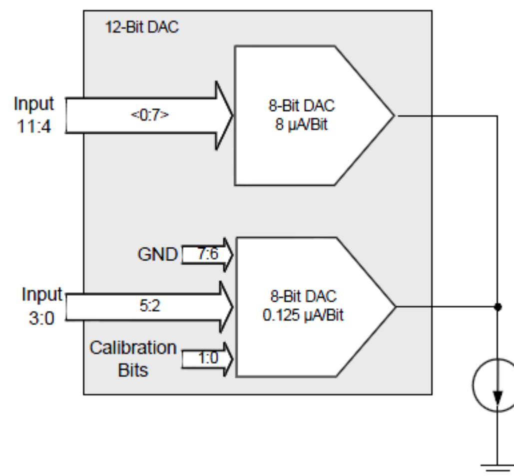


Figure 6.65: Higher resolution current DAC configuration.

For example, the implementation of a 12-bit DAC using two 8-bit DACs require: One DAC scaled to the range 0 - 2.048 mA and the second one scaled to the range $0 - 32\mu A$. The middle 4 bits of the lowest range DAC are used as inputs to the lower 4 bits. This architecture may have problems of there is a mismatch between the two DACs, and therefore, adjustment and scaling may be required. The last two bits of the LSB DAC are used for minor calibration requirements.

6.15.3 Digital Functions

PSoC3/5 provide a powerful suite of digital functions that are interoperable with their analog counterparts. In addition to digital functionality such as counters, timers, cyclic redundancy

check modules, pulse width modulators, quadrature decoders, shift registers, pseudo random sequence (PRS) generators, and precision illumination signal modulators (PrISM) a full set of logic functions such as AND, OR, (NOT), NOR, NAND, XOR, XNOR, and inverters is also provided to provide all of the basic boolean operations. Arbitrarily complex combinations of these logic functions allows the design to create logic configurations for a wide variety of situations involving PSoS3/5's analog and digital blocks. With the exception of the inverter which functions as a NOT gate all of the included logic gates have two digital inputs as a default. The inverter has a single input and a single output but the other gates can have as many as eight digital inputs (*NumTerminals*), inclusive. A second parameter *TerminalWidth* defines the number of bus connections that can be attached to the same number of discrete logic gates in parallel.

All of the digital logic gates used are converted to their VHDL[113] equivalents and reduced to a sum of products and then placed into the Universal Digital Blocks (UDB) Programmable Logic Devices (PLD). This process results in digital logic gates being automatically optimized and placed into the PSoC device. Resource usage is dependant upon the specific logic created and can not be determined prior to project compilation in PSoC Creator.

6.15.4 Gates

Logic levels for the PSoC3/5 gates are defined as:

- True = 1 = high logic level
- False = = = low logic level

The AND gate, shown symbolically in Figure 6.66 functions in the same manner as a logical AND operator, viz., the output is true when all inputs are true and is otherwise false.

Table 6.14: AND gate truth table.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

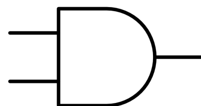


Figure 6.66: AND gates perform logical multiplication.

The OR gate, shown symbolically in Figure 6.67, functions in the same manner as a logical OR gate, viz., the output is true if any input is true and false if all inputs are false.

[113]The Very High-level design language (VHDL) was created as a hardware description language for the development of high-speed integrated circuits and has evolved to an industry standard language for describing digital systems.

Table 6.15: OR gate truth table.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

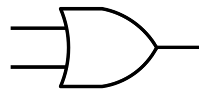


Figure 6.67: OR gates perform logical addition.

The inverter, shown symbolically in Figure 6.68, which is also referred to as a NOT gate, performs a logical inversion function, viz., the output state of the inverter is the inverse state of the input.

Table 6.16: NOT gate truth table.

Input	Output
1	0
0	1

The NAND gate, shown symbolically in Figure 6.69, is the equivalent to a logical AND gate followed by an logical inverter. If all of the inputs to the NAND gate are true, the output is false, otherwise the output is true.

The NOR gate, shown symbolically shown in Figure 6.70, functions as a logical OR gate followed by a logical NOT gate, viz., the output is true if all of the inputs are false, otherwise the output is false.

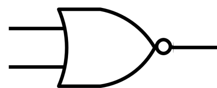


Figure 6.70: The NOR gate functions as a combination of a logical OR and NOT gates.

The XOR (exclusive-OR) gate, shown symbolically in Figure 6.71, is useful as a parity generator. It has two or more inputs and one output. As shown in the Table 6.19, the XOR's output is true when there are an odd number of true inputs. Otherwise, the output is false. The XNOR gate, shown symbolically in Figure 6.72, is an exclusive-NOR gate that functions as a logical XOR gate followed by a logical NOT gate, viz., the output is true when there is an even number of true inputs and otherwise the output is false.

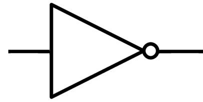


Figure 6.68: An inverter functions is a NOT gate.

Table 6.17: NAND gate truth table.

Input 1	Input 2	Output
0	0	1
0	1	1
1	0	1
1	1	0

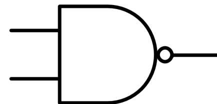


Figure 6.69: The NAND gate functions as the combination of a logical NAND and NOT gate.

Table 6.18: NOR Gate truth table.

Input 1	Input 2	Output
0	0	1
0	1	0
1	0	0
1	1	0

Table 6.19: Exclusive-OR (XOR) truth table.

Input 1	Input 2	Input 3	Output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

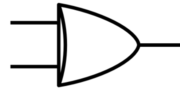


Figure 6.71: An exclusive-OR gate.

Table 6.20: Exclusive-NOR (XNOR) truth table.

Input 1	Input 2	Input 3	Output
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

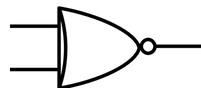


Figure 6.72: An exclusive-NOR (XNOR) gate.

6.15.5 Tri-State Buffer (Bufoe 1.10)

The PSoC3/5 Tri-State Buffer (Bufoe) component is a four terminal, non-inverting buffer with an active high output enable signal shown symbolically in Figure 6.73. When the output enable signal is true, the buffer functions as a standard buffer. When the output enable signal is false, the buffer turns off. It is used to interface to a shared bus, e.g., I^2C. Bufoe's should be used with an I/O pin and not be used in conjunction with internal logic.

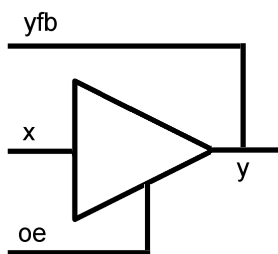


Figure 6.73: A Bufoe is a buffer with an output enable signal (oe).

The four connections are:

x -Input to the Bufoe.
oe (output enable) - The Bufoe is enabled when oe is '1' and otherwise the output is in a high impedance state (referred to as "tri-stated").

y - This connection is connected to the output of the buffer. When oe is true ('1'), this connection is an output, and y has the same value as x. When oe is false ('0'), this connection may be used as an input.

yfb (output) - This is the feedback signal from the y connection. When oe is true ('1') the yfb and y have the same value as x. When oe is false ('0'), yfb has the same value seen at y irrespective of x.

6.15.6 D Flip-flop

A "D flip-flop", shown in Figure 6.74 is a bistable device that can be used to store a digital value that can be preset or reset asynchronously. It functions nominally as a three terminal device

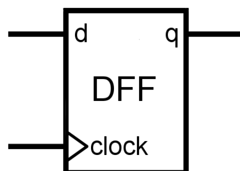


Figure 6.74: PSoC3/5's D Flipflop.

with signal input (d), clock input (clock) and output (q) and is frequently used to implement sequential logic. The D flip-flop output (q) tracks with the D flip-flop output (q) so that it can serve as a storage device.

A fourth input terminal called the *asynchronous preset* (ap) is accessible when the *PresetOr-Reset* parameter is set to Preset. The *ArrayWidth* parameter, whose default setting is '1', allows an array of D flip-flops to be created when the input or output is a bus. The *PresetOrReset*

parameter controls whether the *asynchronous preset* (ap) input or *asynchronous reset* (ar) is visible with a default of "None". All D flip-flop components in the same UDB must have the same ar or ap input. In addition D Flip Flop components in the same PLD must have the same clock signal. Resources The D Flip Flop uses one macrocell. If the *ArrayWidth* parameter is greater than 1, the D flip-flop uses a number of macrocells equal to *ArrayWidth*.

6.15.7 Digital Multiplexer and Demultiplexer

The PSoC3/5 digital multiplexer is used to select 1 of n inputs and the digital demultiplexer is used to dynamically route a signal to one of n outputs, under firmware or hardware control. The most common control method is to connect the mux select signals to a control register using a bus. The control register is then used to select the input or output for the mux/demux. Another option is to drive the select signals from hardware control logic to provide dynamic hardware routing. Tables show the truth tables for a 4-input multiplexer and a 4-output demultiplexer, respectively.

There are three parameters that control multiplexers and demuxiplexers:

NumInputTerminals determines the number of inputs of a multiplexer. The default is 4. The acceptable values are 2, 4, 8, and 16 and the corresponding select input widths are 1, 2, 3 and 4.

NumOutputTerminals determines the number of outputs of a de-multiplexer. The default is 4. The acceptable values are 2, 4, 8, and 16 and the corresponding select input widths are 1, 2, 3 and 4.

TerminalWidth is used to create an array of parallel multiplexers or de-multiplexers when the inputs and outputs are buses. It defines the bus width of the inputs and outputs and has a default value of 1. The width of the Select input is not affected by this parameter.

Table 6.21: 4-Input Multiplexer Truth Table

Select[1]	Select[2]	Input 3	Input 2	Input 1	Input 0	Output
0	0	X	X	X	0	0
0	0	X	X	X	1	1
0	1	X	X	0	X	0
0	1	X	X	1	X	1
1	0	X	0	X	X	0
1	0	X	1	X	X	1
1	1	0	X	X	X	0
1	1	1	X	X	X	1

6.15.8 Lookup Tables (LUTs)

PSoC3/5 have a lookup table component that can be used to provide any logic function with as many as five inputs and eight outputs, inclusive. Such functions are implemented by creating logic equations that are implemented in the UDB PLDs. The LUT should be used any time that a particular input combination should generate a specific set of outputs. The LUT allows an easy method of specifying the input to output relationship without having to generate specific gate level combinatorial logic. Use of the optional registered output mode allows the generation

Table 6.22: 4-Output Demultiplexer Truth Table

Select[1]	Select[2]	Input	Output 3	Output 2	Output 1	Output 0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	0	0
1	1	1	1	0	0	0

of sequential logic. State machines may also be created by registering the outputs and routing some of the outputs back to the LUT inputs. The LUT can configure all of its outputs for all of the possible input combinations. Additionally it can be configured to register the output data on the rising edge of an input clock. Because the LUT is a hardware-only block it does not have any software configuration options. The default LUT is configured with two inputs and two outputs, and the "Register Outputs" option is not selected.

The Clock input of the LUT is only available if the "Register Outputs" option is selected. All outputs will be registered on the rising-edge of this clock. Any clock in the system can be selected however it should be noted that that if any of the outputs go to an I/O they will not work correctly if the LUT is operating faster than the fastest I/O operating speed of the type of PSoC used, e.g., 33 MHz in the case of the PSoC3.

6.15.9 Logic High/Low

Logic High/Low components are provided as part of the PSoC3/5 architectures to provide constant digital values used to hard code digital inputs to in part optimize resource usage. The logic high and logic low functions are used for inputs that remain constant, e.g., for enabling timers, '1' counters, etc. Logic low is defined as '0' and logic High as '1'.

6.15.10 Registers

PSoC3/5 provide two types of very special registers, i.e., control and status. The former is used to control/interact with a module and the latter is used when the firmware needs status information about a module. Thus the status register allows the firmware to read digital signals and the control register can be used as a configuration register to allow the firmware[114] to specify the desired behavior of the digital system. The status register has a clock input and eight connections for status input, $status_0 - status_7$. The number of inputs depends on the *NumInputs* parameter and the firmware queries the input signals by reading the status register. The firmware sets the values of the output terminals for the control register by writing to it. The number of outputs depends on the *NumOutputs* parameter represents the number of output terminals (specified as 1-8) with a default value of 8.

The Bit0Mode Bit7Mode parameters are definable in PSoC Creator and used to set specific bits of the Status Register to be held high after being registered, until a read is executed which

[114]Note: The terms firmware and software are used throughout this text interchangeably and it is left as an exercise for the reader to determine which if either is more appropriate within a given context.

also clears all of the registered values. The settings are: *Transparent* and *Sticky* (Clear on Read). By default, a CPU read of this register will transparently read the state of the associated routing net. This mode can be used for a transient state that is computed and registered internally in the UDB.

In the *Sticky Status, with Clear on Read* mode, the associated routing net is sampled on each cycle of the status and control clock and if the signal is high in a given sample, it is captured in the status bit and remains high, regardless of the subsequent state of the associated route. When the CPU firmware reads the status register, the bit is cleared. Clearing of the status register is independent of the mode and will occur even if the block clock is disabled, it is based on the bus clock and occurs as part of the read operation.

```
Example 6.5: Sample C source code for reading/writing from/to the
status/control registers.

include <device.h>
void main()
{
uint8 value;
value = Status_Reg_1_Read();
}

#include <device.h>
void main()
{
uint8 value;
Control_Reg_1_Write(0x3E);
value = Control_Reg_1_Read();
}
```

6.15.11 PSoC3/5 Counters

PSoC3/5 architecture includes counters and timers which are important in most embedded systems. These counters are capable of counting up, down or up-and-down and are configurable to allow them to operate as 8, 16, 24 or 32-bit counters. Options include compare out and capture input. Additionally, enable and reset inputs can be synchronized with other PSoC components and the period of a count is programmable.

Counters are particulary useful in situations that require the "counting" of events and if required capturing the current count value for programmatic use or to compare an output for hardware synchronization and.or signaling. In the simplest configuration Counters count either up or down and utilize a single input from either other components internal to PSoC, or from an I/O pin. A "count event" occurs with each rising edge of the input and continues until the terminal count is reached at which point the Counter is "reloaded". In the case of a "down" counter the terminal count when the Counter reaches zero and subsequently the Counter is reloaded with the Period value. Counters also have optional "capture" functions that allow the current count to be captured for comparison, or for software processing.

Up/down counters are similar to up and down counters, but there are some important differences. One configuration provides a count input and a direction input. When active a 1 on the

up and down input forces the counter to increment by one on a rising edge of the count input, a 0 on the up and down input causes the counter to decrement by one on a rising edge of the count input. The other configuration provides an up count input and a down count input. The counter will increment or decrement based on which respective count input had a rising edge. This version of the counter requires an additional oversample clock input while all other versions do not. On counter underflow and overflow, flags are set and the period reloaded allowing glitch proof counter expansion in firmware. During each clock cycle, the optional compare output compares the current count to the compare value. The compare mode is configurable to all the standard Boolean comparison modes providing several waveform options. The compare output provides a logic level that may be routed to I/O pins and to other component inputs.

An optional capture input copies the current count value into a storage location on a rising edge. Firmware can be used to read the capture value at any time without timing restrictions as long as the capture FIFO has room. The Capture FIFO allows storage for a maximum of 4 capture values. The enable and reset inputs allow the Counter to be synchronized with other internal or external hardware. The Counter enable signal may be generated by a software API, the hardware compare input or the AND of both. For the hardware Enable input the counter only counts while the Enable input is high. A rising edge on the reset input causes the counter to reset its count as if the terminal count was reached. If the reset input remains high the counter will remain in reset. An interrupt can be programmed to be generated under any combination of the following conditions: when the counter reaches the terminal count, the comparator output is asserted, or a capture event has occurred.

In the default mode the counter counts the number of rising edge events on the count input. The counter can also be used as a clock divider by supplying a clock signal to the count input and using the compare, or terminal count, outputs as the divided clock output. Furthermore, the counter can be employed as a frequency counter by using a known period on the enable input of the counter while counting the signal to measure on the count input. After the enable period, the counter will contain the number of rising edges measured during that period allowing calculation of the input frequency. The up and down counter may be used to measure complementary events such as the output of a quadrature decoder to measure a sensors position. A timer component is is a better choice for timing the length of events, measuring the interval of multiple rising and/or falling edges, or for multiple capture events. Another option is to use a PWM when multiple compare outputs are involved because of the support a PWM provides for center alignment, output kill and deadband[115] outputs.

The input and output connections for PSoC3/5's counter component includes a clock input which defines the oversample clock rate required to increment on upCnt, or decrement on dwnCnt, or alternativey cause neither an upCnt or dwnCnt. The count input is the input connection for the signal to be counted. The counter value is either incremented or decremented depending on the the assigned direction or pin usage selected for the *Clock Mode* parameter. The reset input resets the counter to the starting value. For the "Up Counter" configuration, the starting value is zero and For "Down Counter", "Count Input and Direction" and "Clock With UpCnt & DwnCnt" configurations, the starting value is set to the current period register value.

6.15.11.1 UDB Implementation of a Counter

When the UDB mode is selected for a Counter component:

- The *Resolution* parameter defines the bit-width resolution of the counter. This value may

[115]Deadband refers to a signal range for which nothing happens is often employed to prevent oscillation of a device or "hunting". A deadband is analogous to mechanical backlash in a gear system.

be set to 8, 16, 24 or 32 for maximum count values of 255, 65535, 16777215, and 4294967295 respectively.,

- The *Compare Mode* (software option) parameter configures the operation of the Compare output signal which is the status of a compare between the compare value parameter and current counter value. It defines the initial setting loaded into the control register which can be updated at any time to re-configure the compare operation of the counter.
  1. *Less Than*: The Counter value is less than the compare value
  2. *Less Than Or Equal To*: The Counter value is less than or equal to the compare value
  3. *Equal To*: The Counter value is equal to the compare value
  4. *Greater Than*: The Counter value is greater than the compare value
  5. *Greater Than Or Equal To*: The Counter value is greater than or equal to the compare value
  6. *Software Controlled*: The compare mode can be set during runtime with the SetCompareMode() API call to any one of the 5 compare modes listed above.
- The *Clock Mode* can be up counter, down counter, count input and direction, and count with upCnt and dwnCnt. This parameter configures the desired clocking and direction control method. The value is an enumerated type and can be set to any of the following options:
  1. *Count Input + Direction*: The counter is a bi-directional counter counting up while the up_ndown input is high on each rising edge of the input clock and counting down while up_ndown is low on each rising edge of the input clock.
  2. *Clock with UpCnt DwnCnt*: The counter is a bi-directional counter incrementing the counter by 1 for each rising edge on the upCnt input and decrementing the counter by 1 for each rising edge of the dwnCnt input.
  3. *Up Counter*: The counter is an up counter only configured to increment on any rising edge of the input clock signal while the counter is enabled.
  4. *Down Counter*: The counter is a down counter only configured to decrement on any rising edge of the input clock signal while the counter is enabled.
- The *Period* parameter defines the max counts value (or rollover point) for the counter. This parameter defines the initial value loaded into the period register which can be changed at any time by the software with the Counter_WritePeriod() API. The limits of this value are defined by the Resolution parameter. For 8, 16, 24 and 32-bit Resolution parameters the maximum value of the Period value is defined as $(2^8) - 1$, $(2^{16}) - 1$, $(2^{24}) - 1$, and $(2^{32}) - 1$ or $255, 65535, 16777215, and 4294967295$ respectively. When Clock Mode is configured as "Clock with UpCnt & DwnCnt" or "Count Input and Direction" the counter is set to the period at start and any time the counter overflows at all 0xFF or underflows at all 0x00.
- The *Capture Mode* parameter configures the implementation of the capture input. This value is an enumerated type and can be set to any of the following values:
  1. *None*: No capture implemented and the capture input pin is hidden
  2. *Rising Edge*: Capture the counter value on any rising edge of the capture input Falling *Edge*: Capture the counter value on any falling edge of the capture input
  3. *Either Edge*: Capture the counter value on any edge of the capture input
  4. For the *Software Controlled* mode, the mode is set at runtime by setting the *Compare Mode* bits in the control register Counter_CTRL_CAPMODE_MASK with the enumerated capture mode types defined in the Counter.h header file.
- The *Enable Mode* parameter configures the enable implementation of the counter. This value is an enumerated type and can be set to any of the following options:

1. *Software*: The Counter is enabled based on the enable bit of the control register only.
2. *Hardware*: The Counter is enabled based on the enable input only.
3. *Software And Hardware*: The Counter is enabled if, and only if, both the input and the control register bits are active.

- The *Reload Counter* parameters allow the counter value to be reloaded when one or more of the following selected events occur. The counter is reloaded with its start value (for an up counter this is reloaded to a value of Zero, for a down counter this is reloaded to the max counts or period value). This configuration is ORd with all of the other Reload Counter parameters to provide the final reload trigger to the counter.

1. On *Capture* The counter value will be reloaded when a capture event has occurred. By default this parameter is set to false. This parameter is only shown when UDB is selected for Implementation.
2. On *Compare* - The counter value will be reloaded when a compare true event has occurred. By default this parameter is set to false. This parameter is only shown when the UDB is selected for Implementation.
3. On *Reset* - The counter value will be reloaded when a reset event has occurred. By default this parameter is set to true. This parameter is always shown, but it is only active when UDB is selected for Implementation.
4. On *TC* - The counter value will be reloaded when the counter has overflowed (in count up mode) or underflowed (in count down mode). By default this parameter is set to true. This parameter is always shown, but it is only active when UDB is selected for Implementation. When the clock mode is set to “Clock with UpCnt & DwnCnt” this option reloads to the period value when counter is 0x00 or all 0xFF. This configuration is OR’d with all of the other reload parameters to provide the final reload trigger to the counter.

- The Interrupt parameters allow the initial interrupt sources to be configured. These values are ORd with any of the other Interrupt parameters to give a final group of events that can trigger an interrupt. The software can re-configure this mode at any time; this parameter simply defines an initial configuration.

1. On *TC* -This option is always available; it is set to false by default.
2. *On Capture* - This option is set to false by default. It is always shown, but it is only active when UDB is selected for Implementation.
3. On *Compare* - This option is set to false by default. It is always shown, but it is only active when UDB is selected for Implementation.

and

- The *Compare Value* (Software Option) parameter defines the initial value loaded into the compare register of the counter. This value is used in conjunction with the Compare Mode parameter selected to define the operation of the compare output. This value can be any unsigned integer value from 0 to ($2^Resolution - 1$), but it must be less than the max_counts or Period value. If the value is allowed to be larger than max_counts, the compare output would be a constant 0 or 1 value and is therefore not allowed.

6.15.11.2 Clock Selection

The *Counter* component’s clock/count input can be any signal whose rising edges are to be counted. When configured to utilize the fixed function timer block in the device, the clock input to the *Counter* component has the following restrictions:

1. The clock input must be from a user-defined clock that is synchronized to the bus clock or directly from the bus clock via a clock defined using the existing clock feature, and with a source of the bus clock.
2. If the frequency of the clock matches the bus clock, then the clock must be a direct connection to the bus clock using the existing clock scheme listed earlier. A user-defined clock with a frequency that matches the bus clock will generate an error during the build process.

The Timer, Counter and PWM components share a common set of internal requirements and are therefore implemented in PSoC3/5 as fixed function blocks. When the Fixed Function implementation of a Counter, Timer or PWM is to be employed, certain limitations are imposed., viz., operation is restricted to

- 8 or 16-bits only
- Down count only
- Reload on Reset and
- Terminal Count only
- Interrupt on Terminal Count only

When configured to utilize the fixed function timer block, the clock input to the Counter component will have the following restrictions:

1. The clock input must be from a user-defined clock that is synchronized to the bus clock or directly from the bus clock (via a clock defined using the existing clock feature, and with a source of the bus clock).
2. If the frequency of the clock matches the bus clock, then the clock must be a direct connection to the bus clock (again using the existing clock scheme listed earlier). A user-defined clock with a frequency that matches the bus clock will generate an error during the build process.

Thus the default configuration of the Counter component provides the a very simple counter that increments a count value on every rising edge of the clock input. The count input is the signal whose rising edge is counted and the reset input provides a hardware mechanism for resetting the count value. Since this is configured as an up counter by default, when a reset event occurs on the reset input, the counter value is reset to zero. Terminal count indicates in real time whether the counter value is at the terminal count (Maximum value or Period). The period is programmable to be any value from 1 to $(2^{Resolution}) - 1$.

The compare output is a real time indicator that the count value compares to the compare value as defined in the compare configuration. The compare configuration is set in the control register for the component and can be set by software at any time. The default Maximum Count (Period) is set to $2^{Resolution} - 1$ and the compare value is set to 1/2 of that number. The counter increments on any rising edge clock until it rolls over at the terminal count.

A simple extension of the default configuration provides a clock divider with programmable duty cycle. If a clock input is applied to the counter clock input with the default period and compare parameter settings, the compare output will be a 50% duty cycle clock with 1/256th the frequency of the input clock. This is because the default compare configuration is less than or equal to which would have a high state on the compare output from 0 to 127 and a low signal from 128 to 255. Any even number period setting can have a 50% duty cycle if the compare value or compare configuration is changed. Adding hardware enable functionality to the basic counter allows a frequency counter function to be implemented. If the Enable input is driven by a known period signal such as a 1KHz Clock starting with a counter value of 0x00 and an up counter implementation, the frequency of an input signal is easily determined.

6.15.12 Timers

Timers are a form of counter designed to time the interval between hardware events. They are ubiquitous in embedded systems being used to determine elapsed times between events, periods of recurrent events, triggers for various types of events, elapsed time since an event or function last occurred, etc. Timers have some of the same features found in Counters and PWMs. Typical uses of a PSoC3/5 timers include recording the number of clock cycles between events, measuring the number of clock cycles between two rising edges generated for example by a tachometer sensor or the measurement of the period and duty cycle of a PWM input.

For PWM measurement, a PSoC3/5 timer is configured to start on a rising edge, capture the next falling edge and then capture and stop on the next rising edge. An interrupt on the final capture signals the CPU that all the captured values are available in the FIFO. The PSoC 3/5 timer can be used as a clock divider by driving a clock into the clock input and using the terminal count output as the divided clock output. In general, timers share many features with counters and PWMs. A counter is better used in situations that require the counting of a number of events but also provides rising edge capture input and compare output. A PWM is more appropriate for situations requiring multiple compare outputs with control features like center alignment, output kill and deadband outputs.

PSoC3/5 timers are designed to provide an easy method for timing complex, real time events with great accuracy and minimal CPU overhead. Timers of this type only count down from a predefined state defined by the "period value" which is inversely proportional to the timer's clock frequency. Therefore the minimal time interval that can be measured is determined by the timer's clock.

The maximum timer interval that can be measured is given by:

$$T_{max} = (Timer\ Clock\ Frequency)(Timer\ Resolution) \tag{6.239}$$

The Timer component provided with PSoC includes a function known as "capture". This function is an extremely useful feature in that it makes it possible to "capture" the Timer's "count" at any particular moment and save that value in a FIFO[116] storage location. The FIFO is capable of storing four such values after which the data in the FIFO will be overwritten by new "captured" data.[117] This data can be accessed programmatically, that is by the firmware, without destroying the data read from the FIFO.[118]

PSoC3/5 timers are specifically designed to provide an easy method of timing complex real time events accurately with minimal CPU intervention and may be combined with other analog and digital components to create complex peripherals. PSoC3/5 timers count only in the down direction starting from the period value and require a single clock input. The input clock period is the minimum time interval able to be measured. The maximum timer measurement interval is the input clock period multiplied by the resolution of the timer. The signal interval to be captured may be routed from an I/O pin, or from other internal component outputs. Once started, the timer operates continuously and reloads the timer period value on reaching the terminal count. The timer capture input is the most useful feature of the timer because on a capture event the current timer count is copied into a storage location. Firmware can then read the capture value at any time without timing restrictions as long as the capacity of the capture FIFO is not exceeded.

[116] FIFO = First-In First-Out

[117] The oldest data is overwritten first and therefore the newest data is returned the next time the FIFO is "read".

[118] T

However, it is important to not write to the FIFO when it is full to avoid overwriting the oldest value. If the oldest value is overwritten, the newly captured value will be returned in its place the next time the FIFO is read.It is up to the software to keep track of the amount of data that is written to the FIFO, if unwanted overwriting of its data is to be avoided.

The Capture FIFO allows storage of up to 4 capture values. The capture event may be generated by software, rising or falling edges, or all edges allowing great measurement flexibility. To further assist in measurement accuracy of fast signals an optional 7-bit counter may be used to capture every n[2..127] of the configured edge type. The trigger and reset inputs allow the timer to be synchronized with other internal or external hardware. The optional trigger input is configurable so that a rising edge, falling edge or all edges to start the timer counting. A rising edge on the reset input causes the counter to reset its count as if the terminal count was reached.

PSoC3/5 timers support:

- 8-, 16-, 24- or 32-bit resolution,
- implementation as a Fixed Function or UDB device,
- a 4-deep capture FIFO, an optional capture edge counter,
- configurable hardware/software enable
- continuous or single shot running modes.

6.15.12.1 PSoC3/5 Timer I/O Connections

Timer I/O connections for a PSoC3/5 timer include:

- a clock input that determines the operating frequency of the timer,
- a capture input that copies the period counter value to a 4-sample FIFO in the UDB, or alternatively to a single sample register in the Fixed Function block,
- a capture_out output that is an indicator of when a hardware capture has been triggered
- an interrupt output that is a copy of the interrupt source a terminal count (tc) output that goes high if the current count value is equal to the terminal count (zero)[119]
- a reset input that resets the period counter, to the period value, and the capture counter. This reset function is synchronous and requires at least one rising edge of the clock.
- an enable input that enables the period counter to decrement on each rising edge of the clock. If the enable value is low, then the outputs remain active but the timer does not change states,

6.15.13 Shift Registers

Shift registers are sequential logic circuits that typically consist of cascaded flip-flops sharing a common clock with the output of one flip-flop serving as the input for the next flip-flop in the chain. They are available in a number of configurations as discrete devices, e.g.,

- serial-in, serial-out shift registers[120]
- serial-in parallel out shift registers
- parallel in serial out shift registers

[119] The terminal count output is a zero compare of the period counter value, i.e., if the period counter is zero the output will be high.

[120] Serial shift registers in their simplest form allow shifting in only one direction, i.e., from the input towards the output, often referred to as “right-shift” or “left-shift”.

- bidirectional (reversible) shift registers[121]

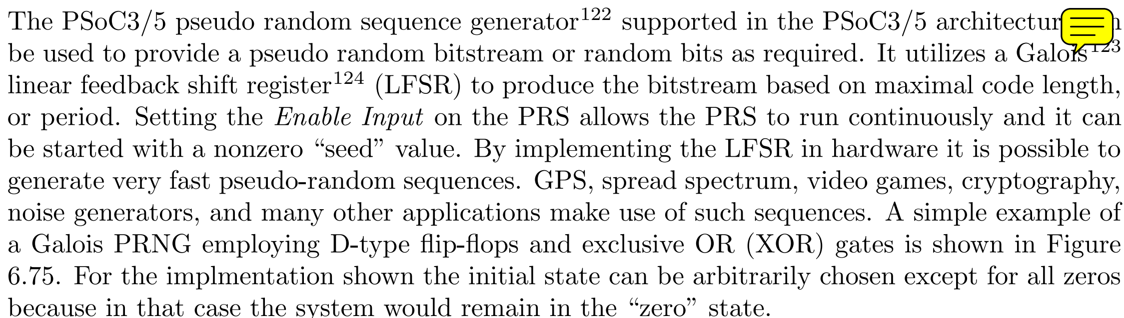
PSoC3/5's Shift Register component provides synchronous shifting of data into and out of a parallel register. The parallel register can be read or written to by the CPU or DMA. The Shift Register component provides universal functionality similar to standard 74xxx series logic shift registers including: 74164, 74165, 74166, 74194, 74299, 74595 and 74597.

In most applications the Shift Register is used in conjunction with other components and logic to create higher level application specific functionality, such as a counter to count the number of bits shifted. In general usage, the PSoC3/5 shift register functions as a 1-32 bit shift register that shifts data on the rising edge of the clock input.

The shift direction is configurable and allows a right shift where the MSB shifts in the input and the LSB shift out the output, or a left shift were the LSB shifts in the input and the MSB shifts out the output. The reset input (active high) causes the entire shift register contents to be set to all zeros. The reset input is synchronous to the clock input. The shift register value may be read by the CPU or DMA at any time.

A rising edge on the optional store input transfers the current shift register value to the FIFO from where it can later be read by the CPU. The store input is asynchronous to the clock input. The shift register value may be written by the CPU or DMA at any time. A rising edge on the optional load input transfers pending FIFO data (already written by CPU or DMA) to the shift register. The load input is asynchronous to the clock input. The Shift Register component may generate an interrupt signal on any combination of the following signals; Load, Store or Reset.

6.15.14 Pseudo Random Sequence Generator (PRS)



The PSoC3/5 pseudo random sequence generator[122] supported in the PSoC3/5 architecture can be used to provide a pseudo random bitstream or random bits as required. It utilizes a Galois[123] linear feedback shift register[124] (LFSR) to produce the bitstream based on maximal code length, or period. Setting the *Enable Input* on the PRS allows the PRS to run continuously and it can be started with a nonzero "seed" value. By implementing the LFSR in hardware it is possible to generate very fast pseudo-random sequences. GPS, spread spectrum, video games, cryptography, noise generators, and many other applications make use of such sequences. A simple example of a Galois PRNG employing D-type flip-flops and exclusive OR (XOR) gates is shown in Figure 6.75. For the implmentation shown the initial state can be arbitrarily chosen except for all zeros because in that case the system would remain in the "zero" state.

The PRS has the following features:

[121] Bidirectional shift registers allow shifts in either direction, e.g., left-to-right or right-to-left.

[122] A pseudo-random number generator does not generate a truly random sequence of values because ultimately it will repeat the sequence.

[123] Galois was a nineteenth century French mathematician who made some significant contributions to Group Theory and to the algebra of polynomials.

[124] A LFSR is a finite state machine which consists of a combination shift register and XOR function in which a seed value is placed in a shift register and is shifted one bit to the right and if the bit value shifted from the rightmost bit position is a 1, then the register is XORed with a mask, otherwise the bit register is shifted one bit position to the right again and then the process of examining the bit and determining if the register should be subjected to an XOR with the mask is repeated. This process continues for a long as required to produce the required pseudo-random bit stream. In some applications single-stepping is employed to produce individual random bit values as required. The number of bits that are generated before the sequence repeats is referred to as its "period". Maximal period LFSRs generate 2^{n-1} bits before repeating where n is the bit length of the register. A 32-bit LFSR will produce in excess of 4 billion bits before repeating. Each of the bit positions in the shift register that have an effect on the next state are referred to as "tap". The speed of the LFSR in generating pseudo random bitstreams is largely a result of the minimal use of combinational logic.

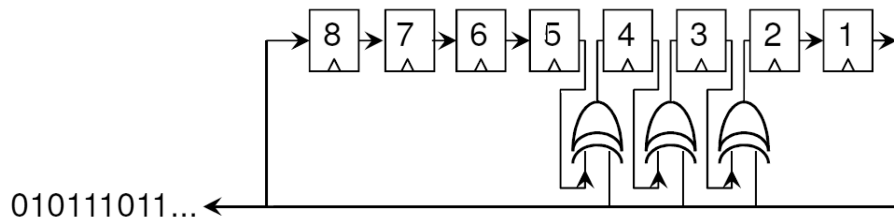


Figure 6.75: A simple example of a Galois PRNG.

- continuous, or single-step, run modes,
- an enable input for synchronized operation with other components,
- a computed pseudo-random number can be read directly from the LFSR,
- either a standard of custom polynomial/seed value can be employed,
- a serial output bit stream
- a PRS sequence lengths 2-64 bits in length

The repeating sequence of states of an LFSR allows it to be used as a divider, or as a counter when a non-binary sequence is acceptable. LFSR counters have simpler feedback logic than natural binary counters or Gray code counters, and therefore can operate at higher clock rates. However it is necessary to ensure that the LFSR never enters an all-zeros state, for example by presetting it at start-up to any other state in the sequence. The PRS has an enable input, a clock input and a serial bitstream output. The clock input is used in continuous mode only and the output is synchronized and when operated in continuous modes The PRS runs as long as the *Enable* input is held high.

6.15.15 Pulse Width Modulator (PWM)

The pulse width modulator is a component that provides user selectable pulse widths for use as single or continuous hardware timing control signals. The most common use of a PWM is to generate periodic waveforms with adjustable duty cycles. The PWM also provides optimized features for power control, motor control, switching regulators and lighting control. It can also be used as a clock divider by supplying it with a a clock input and using the terminal count, or a PWM output, as the divided clock output.

While PWMs, Timers and Counters share many capabilities each provides very specific functionality. A Counter component is used in situations that require the counting of a number of events but also provides rising edge capture input as well as a compare output. A Timer component is used in situations focused on timing the length of events, measuring the interval of multiple rising and/or falling edges, or for multiple capture events. The PSoC3/5 PWM module is provided with an Application Programming Interface (API) that allows the designer to configure the PWM in software.

PSoC3/5's PWM component provides compare outputs to generate single or continuous timing and control signals in hardware. The PWM is designed to provide an easy method of generating complex real time events accurately with minimal CPU intervention. The PWM features include

- 8 or 16 bit resolution
- Configurable Capture

- Configurable Dead-band
- Configurable Hardware/Software Enable
- Configurable Trigger
- Multiple Configurable Kill modes.

and the PWM component may be combined with other analog and digital components to create custom peripherals. The PWM generates up to 2 left- or right-aligned PWM outputs, or 1 center-aligned or dual-edged PWM output. The PWM outputs are double buffered to avoid glitches due to duty-cycle changes while running. Left-aligned PWMs are used for most general purpose PWM uses. Right-aligned PWMs are typically used only in special cases which require alignment opposite of left-aligned PWMs. Center-aligned PWMs are most often used in controlling an AC motor to maintain phase alignment. Dual-edge PWMs are optimized for power conversion where phase alignment must be adjusted.

The optional deadband provides complementary outputs with adjustable dead time where both outputs are low between each transition. The complementary outputs and dead time are most often used to drive power devices in half bridge configurations to avoid shoot-through currents and the resulting potential for damage. A kill input is also available that, when enabled, immediately disables the deadband outputs. Three kill modes are available to support multiple use scenarios. Two hardware dither[125] modes are provided to increase PWM flexibility. The first dither mode increases effective resolution by 2-bits when resources or clock frequency preclude a standard implementation in the PWM counter. The second dither mode uses a digital input to select one of the two PWM outputs on a cycle by cycle basis typically used to provide fast transient response in power converts.

The trigger and reset inputs allow the PWM to be synchronized with other internal, or external, hardware. The optional trigger input is configurable so that a rising edge starts the PWM. A rising edge on the reset input causes the PWM counter to reset its count, as if the terminal count was reached. The enable input provides hardware enable to gate PWM operation based on a hardware signal. An interrupt can be programmed to be generated under any combination of the following conditions; when the PWM reaches the terminal count or when a compare output goes high.

The clock input defines the signal to count and increments or decrements the counter on each rising or following edge of the clock. The reset input resets the counter to the period value and then normal operation continues. The enable input works in conjunction with the software enable and trigger input, if the latter is enabled.[126]

The kill input disables the PWM output(s). Several kill modes are supported all of which rely on this input to implement the final kill of the output signal(s). If deadband is implemented only the deadband outputs (ph1 and ph2) are disabled and the pwm, pwm1, and pwm2 outputs are not disabled.[127] The cmp_sel input selects either pwm1 or pwm2 output as the final output to the pwm terminal. When the input is 0 (low) the pwm output is pwm1 and when the input is 1(high) the pwm output is pwm2 as shown in the configuration tool waveform viewer.[128]

[125]Dithering is sometimes used as a method for reducing harmonic content and involves frequency modulation within a narrow band. In some applications dither is used when a PWM is being used to control a mechanical device, e.g., a valve or actuator , as method of overcoming static friction by introducing some ripple into the actuating current.

[126]The enable input will not be visible in PSoC Creator if the EnableMode parameter is set to "Software Only." This input is not available when the Fixed Function PWM implementation is chosen.

[127]The kill input is not visible if the kill mode parameter in PSoC Creator is set to Disabled. When the Fixed Function PWM implementation is chosen kill will only kill the deadband outputs if deadband is enabled. It will not kill the comparator output when deadband is disabled.

[128]The cmp_sel input is visible when the PWM mode parameter is set to Hardware Select.

The capture input forces the period counter value into the read FIFO. There are several modes defined for this input in the Capture Mode parameter.[129] When the Fixed Function PWM implementation is chosen the capture input is always rising edge sensitive.

The trigger input enables the operation of the PWM. The functionality of this input is defined by the Trigger Mode and Run Mode parameters. After the Start API command the PWM is enabled but the counter does not decrement until the trigger condition has occurred. The trigger condition is set with the Trigger Mode parameter.[130]The terminal count output is 1 when the period counter is equal to zero. In normal operation this output will be 1 for a single cycle where the counter is reloaded with period. If the PWM is stopped with the period counter equal to zero then this signal will remain high until the period counter is no longer zero. The interrupt output is the logical OR of the group of possible interrupt sources. This signal will go high while any of the enabled interrupt sources remain true.

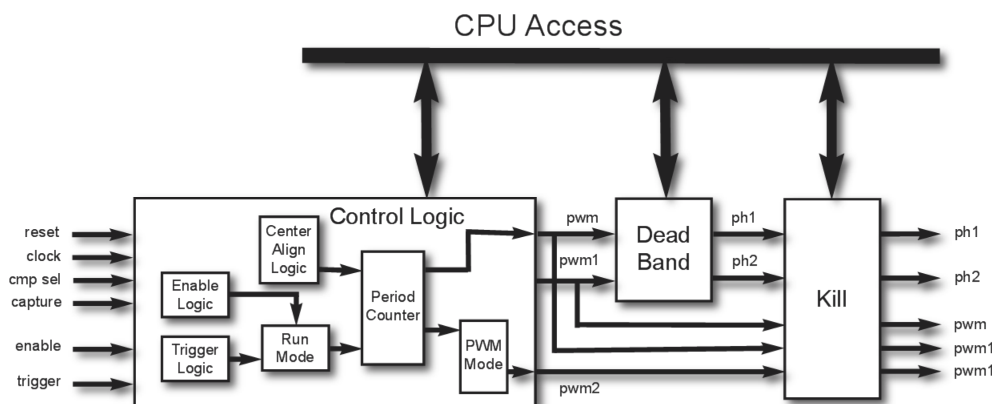


Figure 6.76: A block diagram of PSoC3/5's PWM architecture.

The pwm or pwm1 output is the first, or only, pulse width modulated output and is defined by the PWM Mode, compare modes(s), and compare value(s) as indicated in waveforms in the Configure dialog in PSoC Creator. When the instance is configured in one output, Dual Edged, Hardware Select, Center Aligned, or Dither PWM Modes, then the output pwm is visible. Otherwise the output pwm1 is visible with pwm2 the other pulse width signal. The pwm2 output is the second pulse width modulated output. The pwm2 output is only visible when the PWM Mode is set to Two Outputs.

The ph1 and ph2 outputs are the deadband phase outputs of the PWM. In all modes where only the pwm output is visible these are the phased outputs of the pwm signal which is also visible. In two output mode these signals are the phased outputs of the pwm1 signal only.[131] The bit-width resolution of the period counter is 8-16 bits with 8-bits as the default value.

6.15.16 Precision Illumination Signal Modulation (PrISM)

PSoC3/5 have precision illumination signal modulation (PrISM) components that use linear feedback shift registers (LFSRs) of the type discussed in section 6.15.14 to generate a pseudo random bit stream sequence and up to two user-adjustable, pseudo random, pulse densities.

[129]The capture input is not visible if the Capture Mode parameter is set to None.
[130]The trigger input is not visible if the trigger mode parameter is set to None.
[131]Both of these outputs are visible if deadband is enabled in 2-4 or 2-256 modes and are not visible if deadband is disabled.

ranging from 0 to 100. The PrISM runs continuously after started and as long as the Enable input is held high. Its pseudo random number generator may be started with any valid seed value excluding 0.

The result is modulation technology that significantly reduces low-frequency flicker and radiated electro-magnetic interference (EMI) which are common problems with high brightness LED designs. The PrISM is also useful in other applications requiring this capability, such as motor controls and power supplies.

6.15.17 Quadrature Decoder

PSoC3/5's Quadrature Decoder (QuadDec) component provides the ability to count transitions of a pair of digital signals. The signals are typically provided by a speed/position feedback system mounted on a motor or trackball. The signals typically called A and B are positioned 90 out-of-phase, which results in a "Gray" code output. A Gray code is a sequence of bits in which only one bit changes on each count. This is essential to avoid glitches, and it allows detection of direction and relative position. A third optional signal, named index, is used as a reference to establish an absolute position once per rotation. Quadrature Decoders are used to decode current position, velocity and direction of an object (mouse, trackball, robotic axles) inputs. It can also be used for precision measurement of speed, acceleration and position of a motor's rotor and with rotary knobs, to determine user input.

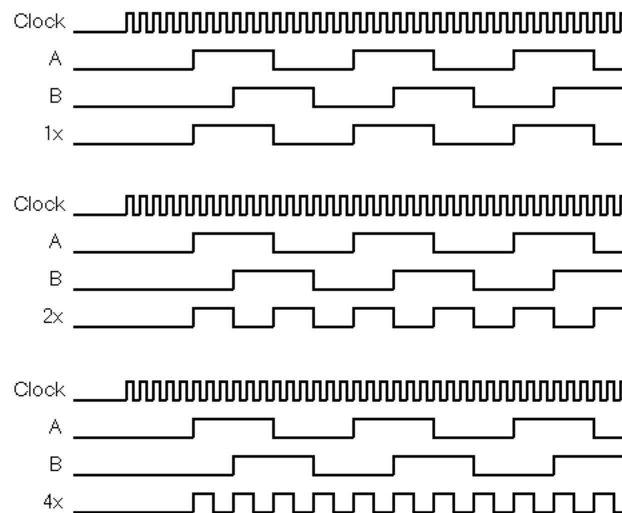


Figure 6.77: Counter resolution for 1x,2x and 4x.

The index input detects a reference position for the quadrature encoder. If an index input is provided, when inputs A, B, and index are zero, the counter is reset to zero. Additional logic is typically added to gate the index pulse. Index gating allows the counter to be reset only during one of many possible rotations, e.g., as in the case of a linear actuator that only resets the counter when the far limit of travel has been reached. This limit is signaled by a mechanical limit switch whose output is AND-ed with the Index pulse.

The clock input clock input is required for sampling and glitch filtering of the inputs. If glitch filtering is used then the filtered outputs will not change until three successive samples of the input are the same value. For effective glitch filtering, the sample clock period should be

greater than the maximum time during which glitching is expected to take place. A counter can be incremented/decremented at a resolution of 1x, 2x, or 4x the frequency of the A and B inputs as shown in Figure 6.77. The clock input frequency should be greater than, or equal to, 10x the maximum A or B input frequency.

An interrupt output is provided following the occurrence of one or more of the following events:

- counter overflow and underflow
- counter reset due to index input (if index is used)
- invalid state transition on the A and B inputs

The counter size is defined in terms of the number of bits. The counter holds the current position encoded by the quadrature encoder. A counter size should be selected that is large enough to encode the maximum position in both the positive and negative directions. The 32-bit counter implements the lower 16 bits in the hardware counter and the upper 16 bits in software to reduce hardware resource usage. Available settings include: 8, 16, or 32 bits.

A field is provided in PSoC Creator that determines whether or not to apply digital glitch filtering to all inputs. Filtering can be applied to reduce the probability of having miscounts due to glitches on the inputs. Some filtering is already done using hysteresis on the GPIOs, but additional filtering may be required. If selected, filtering is applied to all inputs. The filtered outputs do not change until three successive samples of the input are the same value. For effective filtering, the period of the sample clock should be greater than the maximum time during which glitching is expected to take place.

```
Example 6.7 Sample source code demonstrating the use of the quadrature
decoder and writing the position information to an LCD.

#include <device.h>
void main()
{
uint8 stat;
uint16 count16;
uint8 i;
CYGlobalIntEnable;
LCD_1_Start();
QuadDec_1_Start();
QuadDec_1_SetInterruptMask(QuadDec_1_COUNTER_RESET |
QuadDec_1_INVALID_IN);
stat = QuadDec_1_GetEvents();
LCD_1_Position(1, 0);
LCD_1_PrintInt16(stat);
while(1)
{
CyDelay(150);
count16 = QuadDec_GetCounter();
LCD_1_Position(2, 0);
LCD_1_PrintInt16(count16);
}
}
```

6.15.18 Cyclic Redundancy Check (CRC)

A Cyclic Redundancy Check or CRC-check as it is commonly referred to is a method of ascertaining the integrity or lack thereof of digital data that is often used when transferring data from one spatial domain to another. It is equivalent to conducting polynomial long division and retaining only the remainder. The remainder is then appended to the data and transmitted to another location. Upon arrival the division process is repeated and the transmitted remainder is compared to the locally calculated one. If they do not agree some system return a negative acknowledgment is sent back to the transmitter if the data usually causing the data to be retransmitted. This type of check is supported by PSoC3/5 and in the default configuration is used to compute the CRC value for a serial bit stream of arbitrary length that is sampled on the rising edge of the data clock. The CRC value is either reset to 0 before starting or can optionally be seeded with an initial value. Following the computation of the CRC for a particular bitstream, the computed CRC value is available. CRCs are often used for checking the integrity of stored data as well as transmitted data.

PSoC3/5's CRC component has provisions for inputting data and a clock signal with the result of the check being accessible programmatically.

```
Example 6.4 The following C source code illustrates how to computer
 a CRC using PSoC3/5's CRC component.

#include <device.h>
void main()
{
        uint32 crc_val = 0;
        uint16 crc_part1 = 0;
        uint16 crc_part2 = 0;
        uint8 i = 0;
        uint8 j = 0;

        clock_Enable();
        di_Enable();
        LCD_Start();
        CRC_Start();
        for(i=0;i<4;i++)
        {
                for(j=0;j<=13;j+=5)
                {
                        crc_val = CRC_ReadCRC();
                        crc_part2 = HI16(crc_val);
                        crc_part1 = LO16(crc_val);

                        LCD_Position(i, j);
                        LCD_PrintInt16(crc_part2);
                j+=4;
                LCD_Position(i, j);
                LCD_PrintInt16(crc_part1);
                CyDelay(500);
}
}
        for(;;){}
}
```