Freescale Semiconductor
Application Note

Document Number: AN3608
Rev. 0, 04/2009

I²C Applications on the MC56F801x/MC56F802x3x with Processor Expert

by: John L. Winters, Applications Engineering
Donnie Garcia, Systems Engineering
MSG Applications

This application note describes the use of Processor Expert to easily generate an I²C application for the Freescale MC56F801x series as well as the Freescale MC56F802x3x series. Included is source code for both family's projects as well as description of the simple hardware connections needed to complete the projects. The note can be used as a guide to ease migration of I²C applications from the MC56F801x series to the MC56F802x3x series.

The application that is used for illustration tests the EEPROM device, M24C64, as a slave device using the Freescale DSC.

Contents

1 Introduction

Section 2, "MC56F8013 Interfacing to M24C64 Over I²C Bus," introduces the project constructed to test an M24C64 EEPROM using the MC56F8013 device. Section 3, "MC56F8037 Interfacing to M24C64 Over I²C Bus," discusses the similar project for the MC56F8037 device.

freescale semiconductor

The application chosen is from the Processor Expert HWbean test library for the intI2C. It has been modified to use the more recent EEPROM devices with 16-bit memory addresses, rather than the devices with 8-bit memory addresses included in the stationery. These changes are annotated in this note in the main program.

Then, the similar project constructed for the MC56F8037 device in section 3 will be shown. These projects will use the same daughter card (breadboard) and have slightly different software drivers for the I2C of the DSC.

Finally, in Section 4, "Comparing Applications — MC56F801x vs. MC56F802x3x," we will show that the low-level software structure differs only slightly between the two projects.

The software for these two projects is available with the application note. The hardware is simple and may be constructed easily on breadboard from the descriptions provided. It is also available for loan from the author.

2 MC56F8013 Interfacing to M24C64 Over I2C Bus

This section addresses the MC56F8013 application. Section 3, "MC56F8037 Interfacing to M24C64 Over I2C Bus," addresses the corresponding application for the MC56F8037. Although the lowest level software drivers differ, it will be shown that the user's application code can remain the same.

2.1 System Description

The system consists of:

- An MC56F8013 device mounted on a generic fanout board. The purpose of the fanout board is to supply access to each pin of the device in the form of triple row headers at 1/10 inch spacing. The middle row pins of the three rows carry the signal. The other rows allow connection to power or ground. Because the ground pins are near the signal pins, twisted-pair wiring was used to convey signals from the adjacent pins on the fanout board. Power is bench at 3.3 V.
  Hint: Be sure to connect to PTB0/SCL and PTB1/SDA.
- The system also consists of software generated by a code generator included with Code Warrior, Processor Expert. The software runs a test of the memory device over the I2C bus.

A scope was connected to the signals on the I2C bus, SCL, and SDA to observe bus characteristics. The startup of the test program is shown at two different time scales in Figure 1 and Figure 2. In these figures, the SDA line is green and the SCL line is yellow. In both figures, the leftmost item of interest is the START condition (indicated by a green rectangle in Figure 2; stop is red rectangle), if the SDA goes HI to LO while the SCL is HI. The screen is too small to display the complete memory test. This is simply the beginning of the memory test; the memory test comprises the essence of the code example under study.

The eight bits following the START condition comprise a byte of data. Because the most significant bit (MSB) is transmitted first, it is a simple matter to read the eight bits off the figure. This is made easier by realizing that the data is to be read with the SCL signal high. So you should look at the tops of the yellow signal and see if the green signal is above or below these tops to determine HI or LO.

HI is one, and LO is zero. Using this simple scope technique, we read the first eight data bits sent by the master as 10110000. The first seven of these bits are the slave being addressed. The eighth bit is the

direction bit. Because it is zero, this means, per the specification, that the master will write data to the addressed slave. Do not confuse the slave address with an address in a particular device, such as an EEPROM.

The slave being addressed is 1011000, or 0x58, or decimal 88. As can be seen below in the code example, this address is an intentional wrong address, a trick to reset the I2C devices. There is no reset line on these devices, but this very useful trick makes it possible to reset them anyway. In the figure, there is a white line over the illegal address.

The bit following the eight data bits (7-bit I2C device address plus 1 direction bit, which indicates master to slave, or write to memory) is the acknowledge ("A") bit. The SDA bit is not driven low by the slave to acknowledge the byte. "A" is zero. This is because the address was bad — no device was there to recognize that address. There is then a stop condition, where the green SDA line cuts up through the yellow SCL line. Then there is some delay prior to the initiation of another command by the master. This second command is addressed to slave 0x50, the actual address of the EEPROM. It is this command that is the first command to the device. There is a second white line over the valid address, 0x50. Note that the A bit after the second address is acknowledged, which means that the EEPROM is there to take the command that follows. So from there, the data that is sent to the EEPROM (as seen in Figure 2) is 0x00, 0x00, 0x01, 0x02.

The first data byte is the address byte for the EEPROM. The application is writing to address zero of the EEPROM. In this example it is an 8-bit address EEPROM. The next bytes — 1, 2, and 3 — go into the first and second locations of the EEPROM. Many more bytes follow than can be shown on this diagram. Later in the test, the memory will be read back to see if this simple count is contained in the memory.

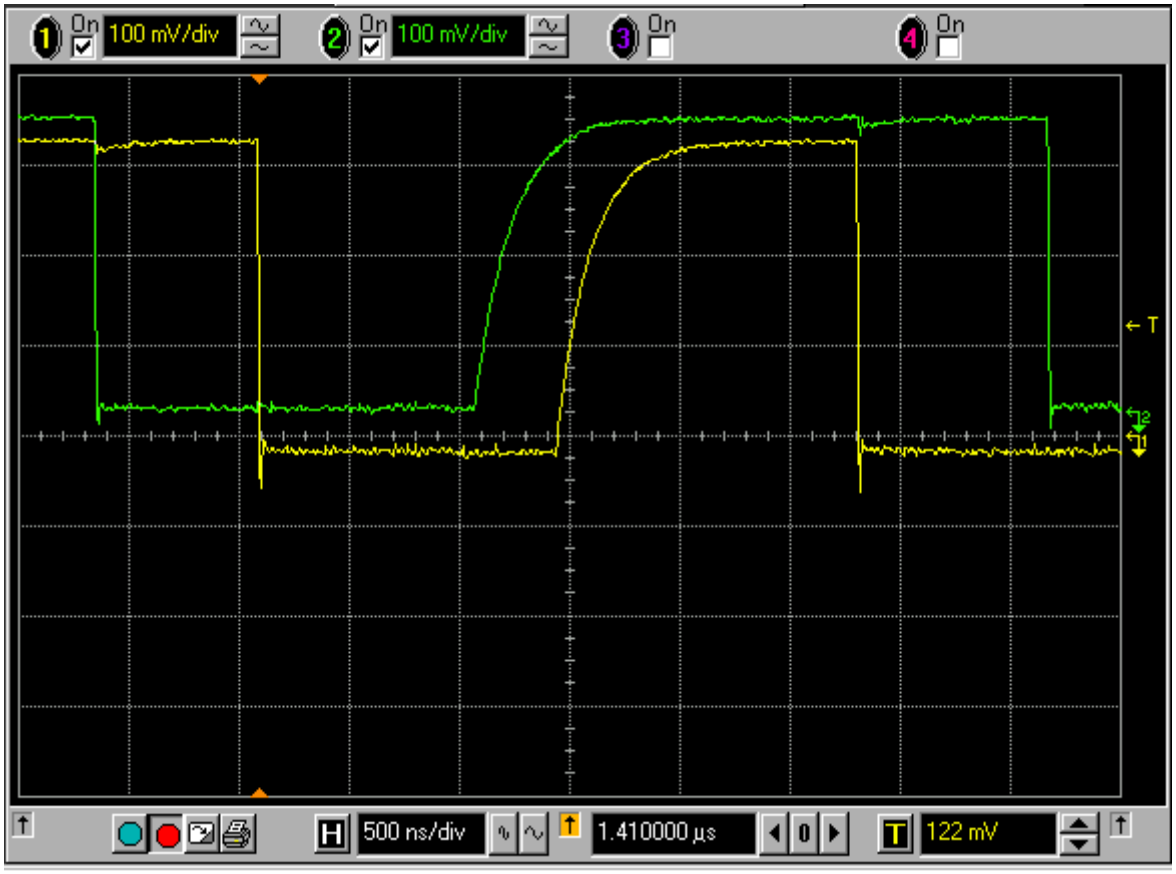
1 On 100 mV/div
2 On 100 mV/div
3 On
4 On
← T
H 500 ns/div
1.410000 µs
0
T 122 mV

Figure 1.

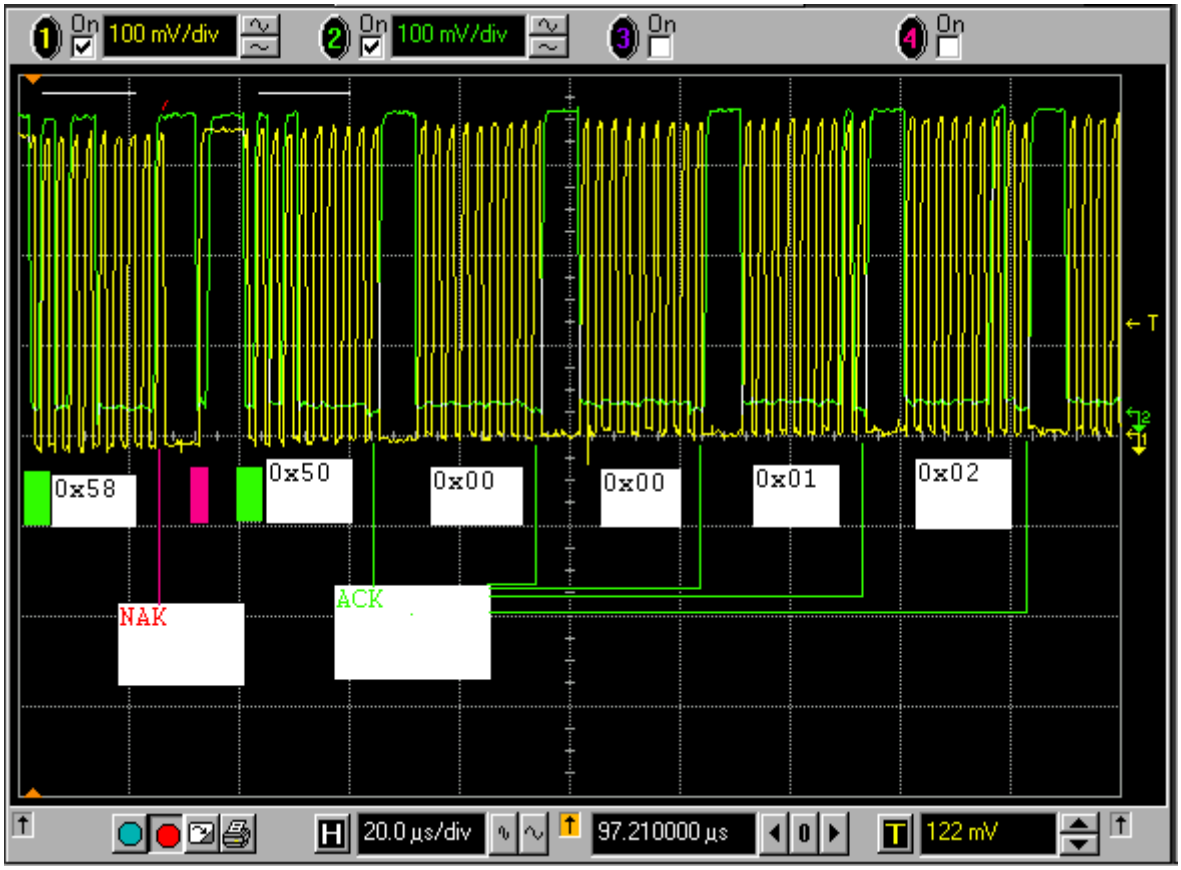

Figure 2.

2.2 Hardware

The hardware consists of the MC56F8013 mounted on a generic fanout board and a breadboard connected to the fanout board with twisted-pair wiring. The breadboard contains the M24C64 EEPROM device. The daughter card wiring is shown in Figure 3 and Figure 4 (viewed from underneath board).

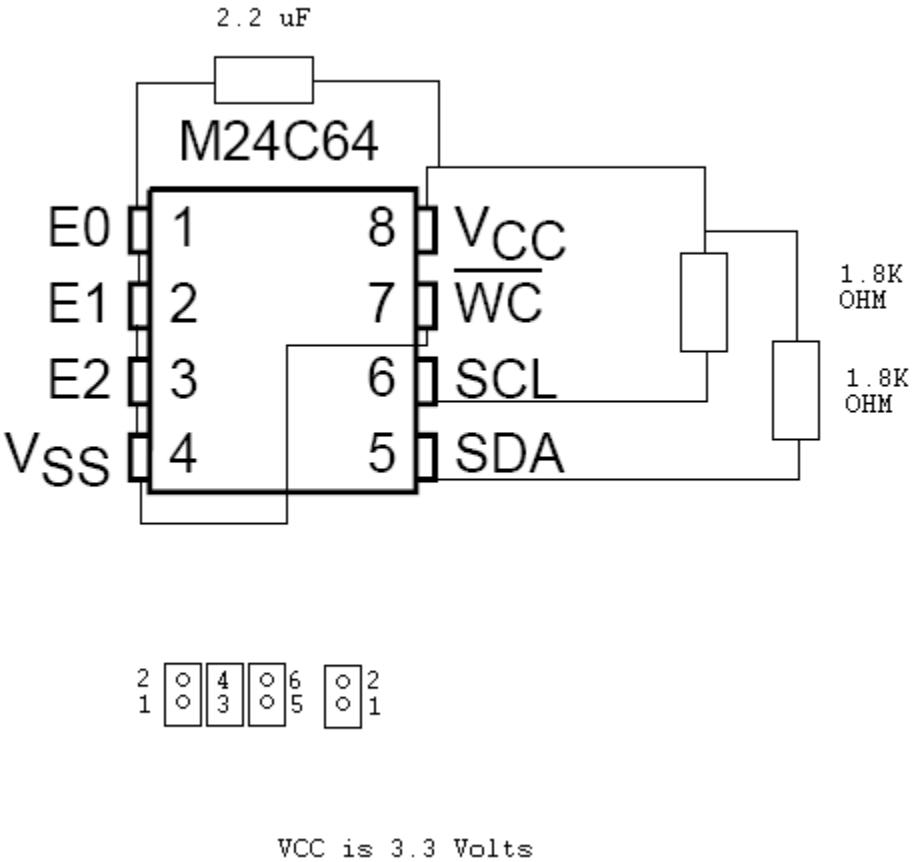
2.2 uF
M24C64
E0
E1
E2
VSS
1
2
3
4
8
7
6
5
VCC
WC
SCL
SDA
1.8K OHM
1.8K OHM
2 1
4 3
6 5
2 1
VCC is 3.3 Volts

Figure 3.

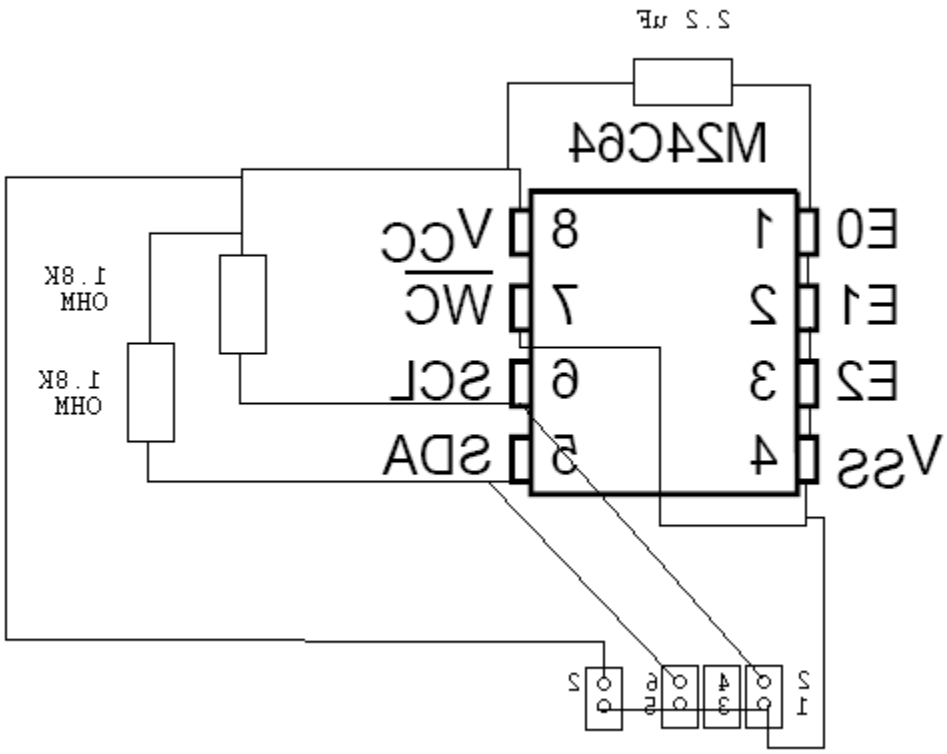

Figure 4.

To wire the daughter card to the demo card, three twisted-pair wires are used. Note that there are two header blocks on the daughter card: Signals (on the bottom left in Figure 3) and Power. These are constructed of standard 1/10” stick headers on perforated bread board. Hand point-to-point soldering was used. Three twisted pairs, with female sockets to fit the stick headers, were used. Each was about two feet long. There is one twisted pair for each type of power, SCL and SDA.

- The Power header has two pins, and is connected to:
  - Power twisted pair, to power the daughter card from the fanout board:
    - Pin 1 (0 V)
    - Pin 2 (3.3 V)
- The Signal header has six pins, but 3 and 4 are not used. This leaves room for the connectors at the ends of the twisted pairs. The two twisted-pair wires and the pins they connect to are:
  - SCL twisted pair
    - Pin 1 (0 V)
    - Pin 2 SCL — clock for the I2C interface
  - SDA twisted pair
    - Pin 5 (0 V)
    - Pin 6 SDA — data for the I2C interface

One wire of each pair is connected to 0 V on the fanout card. Power, SCL, and SDA signals are connected at the fanout card to the other ends of the twisted pair wires.

2.3 Software

Processor Expert is part of the Code Warrior IDE for DSC. Processor Expert is a C code generator. The code base from which these examples are generated is an integral part of Processor Expert stationery. The examples may be instantiated by first installing Processor Expert on a PC by installing Code Warrior for DSC, currently at version 8.2. The code may then be generated by selecting “new” from the file tab of the IDE. This will cause a window called “New” to appear.

After “new” is selected, choose “Processor Expert Examples Stationery.” After “Processor Expert Examples Stationery” is selected, fill in a project name of your choice in the Project Name field of the “New” window. Then select “OK.”

After “OK” has been selected, a “New Project” window will appear. In the “New Project” window, expand “DemoApplications.” After expanding “DemoApplications,” expand “HWBeans.” “HWBeans” contains demo code for each peripheral. After expanding “HWBeans,” expand the family, either MC56F801x for the MC56F8013, or MC56F803x for the MC56F8037.

After expanding the desired family, select IntI2C (Internal I2C, using the internal I2C peripheral logic on the DSC SOC (system on chip). Select OK. Processor Expert is now poised to generate code. To make it generate the base code example, from the Processor Expert tab select “generate code” for your project name. The code generated will be used to communicate with an I2C EEPROM 24xx08.

This device addresses its internal memory with an 8-bit address. As part of this note, we will show how to modify this code for the 16-bit address used by the device we target in this note, the M24C64.

The software presented:

- First the “bean” level, or highest level in terms of what parameters go into the code generator (Processor Expert) to generate the C code.
- Second, the code will be presented at the C level, starting with the main program, and descending down through the call hierarchy to the driver code:
  - main top level C routine supplied by application designer
  - Events.c - high level user-supplied events routine, supplied by application designer
  - I2C1.c Low level code generated by Processor Expert in C language, based on GUI inputs

2.3.1 Processor Expert Bean Parameters

The I2C peripheral is served by an I2C hardware bean. The beans have properties, methods, and events. Each bean is configured by a GUI, and then generates the I2C1.c program as well as the skeleton code for the Events.c program. Interrupt handlers from within I2C1.c manage the interrupts for the I2C peripheral and call user written Events. In these Events the user can, for example, keep track of statistics on the bus, as is done in this application.

2.3.1.1 I2C1 Internal I²C Hardware Bean Properties

The properties of the I²C bean have been selected via the GUI interface for this application, as shown in the left side of Figure 5. The bean's properties determine how the DSC peripheral registers for the I²C peripheral will be initialized, as shown on the right side. Changes from the default are highlighted on the right in blue-green.

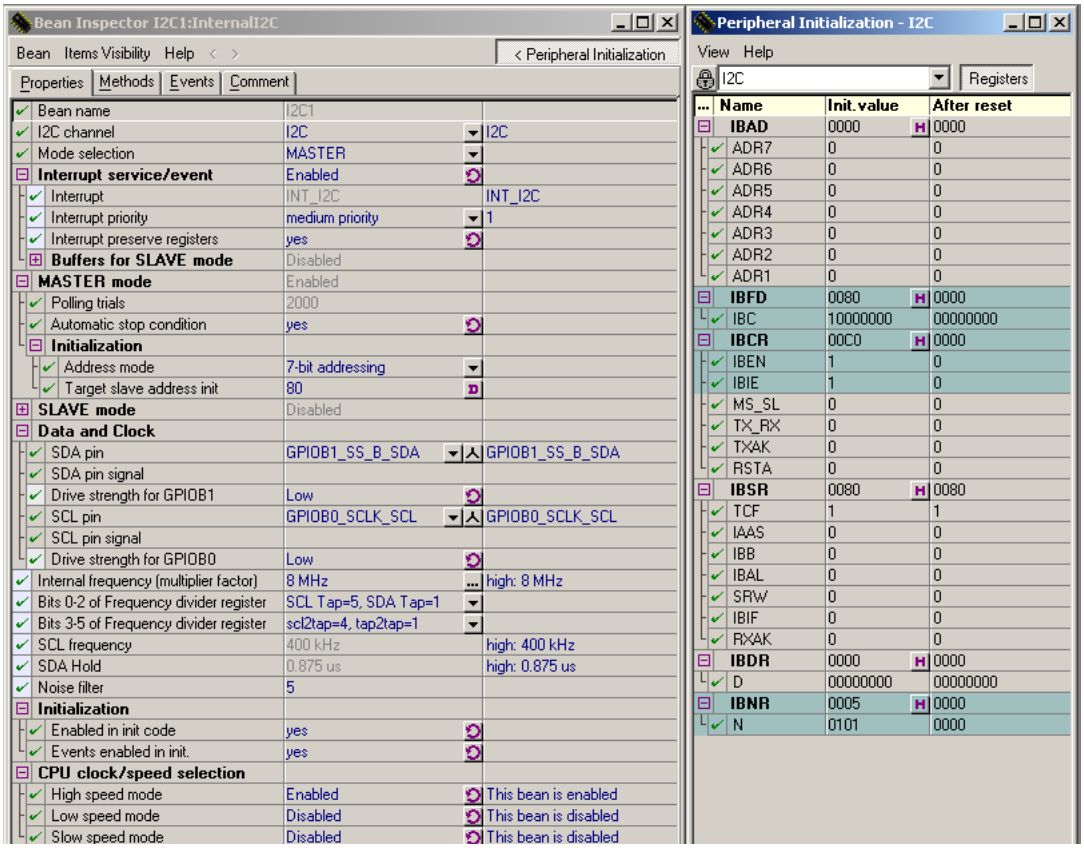

Figure 5.

2.3.1.2 I2C1 Internal I2C Hardware Bean Methods

This application only needs some of the available methods, or driver calls, for this peripheral. These are checked as shown in Figure 6.

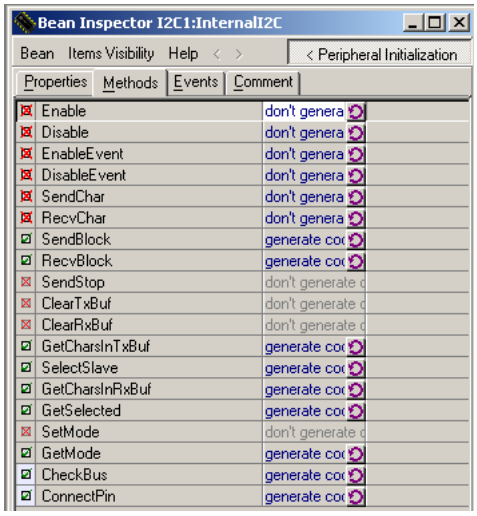

Figure 6.

2.3.1.3 I2C1 Internal I2C Hardware Bean Events

The events are called by the interrupt service routine driver code provided by Processor Expert. If the application needs to react to interrupts, it can do so by using these “hooks,” which provide a location to place needed functions.

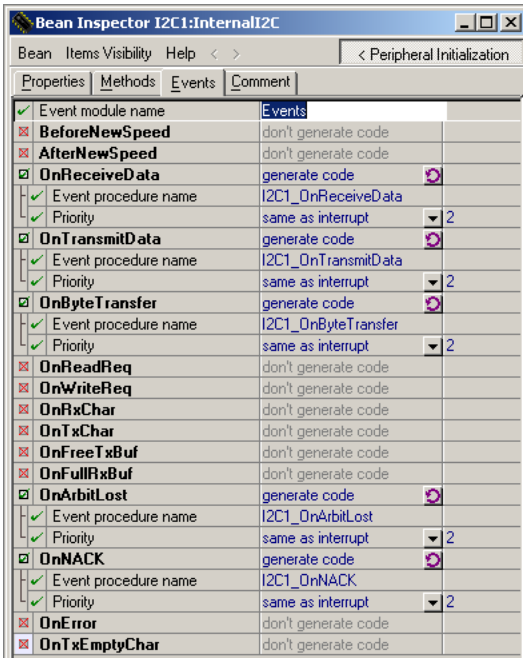

Figure 7.

2.3.2 MC56F8013 C Code with Annotation

Here we have the application code for testing the EEPROM device. Areas of the code that have been changed to accommodate the larger memory of the EEPROM device selected for this application note are highlighted in red. These changes are limited to this one file in which the main program resides. In fact, all the changes are in the main program. (These same changes were carried over to the code for the MC56F8037, but are not shown again when that device is discussed).

2.3.2.1 MC56F8013 Main Program

These header files are generated by Processor Expert. The I2C1.h file is associated with the I2C bean.

```
/*Including used modules for compiling procedure*/
#include "Cpu.h"
#include "Events.h"
#include "I2C1.h"
#include "TEST1.h"

/*Include shared modules, which are used for whole project*/
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
```

The counters are used to garner statistics during the course of the test and are updated by the Events in response to the I2C interrupts.

```
volatile word OnReceiveDataNum = 0;
volatile word OnTransmitDataNum = 0;
volatile word OnByteTransferNum = 0;
volatile word OnArbitLostNum = 0;
volatile word OnNACKNum = 0;

volatile word OnSendRecvDataNum;
volatile word OnErrorNum;


/* Communication with I2C EEPROM 24xx64 with 16 bit memory addresses */
```

No reset line is associated with the I2C device. It does not need one. A reset is effected here by sending an erroneous slave address. This puts the I2C device into a known state.

```
/*
Scenario:
0 Wrong slave address
```

From this point on the correct address is used, and the device catches the first valid message.

```
1 a) Send memory address (always 2 bytes) + data (1..10)
  b) Send memory address
  c) Receive 10 bytes

2 a) Send memory address + data (0xAA's)
  b) Send memory address
  c) Receive 10 bytes

*/
```

```
volatile byte Err;

void main(void)
{
  test_sRec testRec;
```

Here is a needed change to use the larger EEPROM device: there are two bytes to address memory, not one.

Do not confuse the memory address with the slave address. The slave address can choose which device to address. The memory address chooses the address within the device addressed.

```
byte Data[12]; // first two bytes are memory address
byte RecvData[10];
  word rcv;
  bool Error = FALSE;
  byte i;
  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/

  TEST1_testStart(&testRec, "InternalI2C test");
  TEST1_testComment(&testRec, "Please interconnect the pins GPIOB0_SCLK_SCL and
GPIOB1_SS_B_SDA with 24xx08 serial EEPROM.");

  Data[0] = 0; // initial memory address of zero
  Data[1] = 0; // memory address is two bytes

  /* Wrong address slave */
  OnErrorNum = 0;
  I2C1_SelectSlave(88); // 88 is 0x58 (wrong address)
  Err = I2C1_SendBlock(&Data[0],2,&rcv);
  I2C1_SelectSlave(80); // 80 is 0x50 (correct address)

  /* Send memory address + data (1..10) */
  for (i=1; i<11; i++) {
    Data[i+1] = i; // skip first two locations where address is stored.
  }
  while ((OnSendRecvDataNum == 0) && (OnErrorNum == 0)) {}
  while (I2C1_CheckBus() == I2C1_BUSOFF) {}
  OnErrorNum = 0;
  OnSendRecvDataNum = 0;
  Err = I2C1_SendBlock(&Data[0],12,&rcv);
  if ((Err) || (rcv != 12)) {
    Error = TRUE;
  }
  while ((OnSendRecvDataNum == 0) && (OnErrorNum == 0)) {}
  while (I2C1_CheckBus() == I2C1_BUSOFF) {}
  OnErrorNum = 0;
  OnSendRecvDataNum = 0;
  Cpu_Delay100US(100);
  /* Send memory address */
  Err = I2C1_SendBlock(&Data[0],2,&rcv);
  if ((Err) || (rcv != 2)) {
    Error = TRUE;
  }
  while ((OnSendRecvDataNum == 0) && (OnErrorNum == 0)) {}
  while (I2C1_CheckBus() == I2C1_BUSOFF) {}
```

```
OnErrorNum = 0;
OnSendRecvDataNum = 0;
/* Receive 10 bytes */
do {
  Err = I2C1_RecvBlock(&RecvData[0],10,&rcv);
} while (Err == ERR_BUSOFF);

while ((OnSendRecvDataNum == 0) && (OnErrorNum == 0)) {}
while (I2C1_CheckBus() == I2C1_BUSOFF) {}
if ((Err) || (rcv != 10) || (OnSendRecvDataNum != 1)) {
  Error = TRUE;
}
for (i=1; i<11; i++) {
  if (RecvData[i-1] != i) {
    Error = TRUE;
  }
}


/* Send memory address + data (0xAA's) */
for (i=1; i<11; i++) {
  Data[i+1] = 0xAA;  // skip first two locations for address
}
OnErrorNum = 0;
OnSendRecvDataNum = 0;
Err = I2C1_SendBlock(&Data[0],12,&rcv);
if ((Err) || (rcv != 12)) {
  Error = TRUE;
}
while ((OnSendRecvDataNum == 0) && (OnErrorNum == 0)) {}
while (I2C1_CheckBus() == I2C1_BUSOFF) {}
OnErrorNum = 0;
OnSendRecvDataNum = 0;
Cpu_Delay100US(100);
/* Send memory address */
Err = I2C1_SendBlock(&Data[0],2,&rcv);
if ((Err) || (rcv != 2)) {
  Error = TRUE;
}
while ((OnSendRecvDataNum == 0) && (OnErrorNum == 0)) {}
while (I2C1_CheckBus() == I2C1_BUSOFF) {}
OnErrorNum = 0;
OnSendRecvDataNum = 0;
/* Receive 10 bytes */
do {
  Err = I2C1_RecvBlock(&RecvData[0],10,&rcv);
} while (Err == ERR_BUSOFF);
while ((OnSendRecvDataNum == 0) && (OnErrorNum == 0)) {}
while (I2C1_CheckBus() == I2C1_BUSOFF) {}
if ((Err) || (rcv != 10) || (OnSendRecvDataNum != 1)) {
  Error = TRUE;
}
for (i=1; i<11; i++) {
  if (RecvData[i-1] != 0xAA) {
    Error = TRUE;
  }
}
```

```
  if (!Error) {
    TEST1_testComment(&testRec, "Stored data verification OK.");
  }
  else {
    TEST1_testFailed(&testRec, "Stored data verification.");
  }

  /* Event test */
  Error = FALSE;
  if (OnReceiveDataNum != 2) {
    Error = TRUE;
  }
  if (OnTransmitDataNum != 4) {
    Error = TRUE;
  }
  if (OnByteTransferNum != 55) {
    Error = TRUE;
  }
  if (OnArbitLostNum != 0) {
    Error = TRUE;
  }
  if (OnNACKNum != 1) {
    Error = TRUE;
  }
  if (!Error) {
    TEST1_testComment(&testRec, "Event test OK.");
  }
  else {
    TEST1_testFailed(&testRec, "Event test.");
  }

  TEST1_testEnd(&testRec);
}
```

2.3.2.2 MC56F8013 Events.c

These are the functions provided by the programmer and called by the I2C interrupt service routine in the Processor Expert driver. Events are called from the Processor Expert-supplied driver code. The skeleton code for the events is provided by Processor Expert, and the user can easily add code to respond to events as needed. The user can add any code required for the application, or no code if there is no wish for the code to be “event driven.”

```
/** ###################################################################
**     Filename  : Events.C
**     Project   : I2Cbean8013Fast
**     Processor : 56F8013VFAE
**     Beantype  : Events
**     Version   : Driver 01.03
**     Compiler  : Metrowerks DSP C Compiler
**     Date/Time : 23.06.2006, 14:21
**     Abstract  :
**         This is user's event module.
**         Put your event handler code here.
**     Settings  :
**     Contents  :
**         I2C1_OnReceiveData  - void I2C1_OnReceiveData(void);
```

```
**         I2C1_OnTransmitData - void I2C1_OnTransmitData(void);
**         I2C1_OnByteTransfer - void I2C1_OnByteTransfer(void);
**         I2C1_OnArbitLost    - void I2C1_OnArbitLost(void);
**         I2C1_OnNACK         - void I2C1_OnNACK(void);
**         I2C1_OnTxAbort      - void I2C1_OnTxAbort(void);
**         I2C1_OnStart        - void I2C1_OnStart(void);
**         I2C1_OnStop         - void I2C1_OnStop(void);
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2006
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
** ###################################################################*/
/* MODULE Events */

#include "Cpu.h"
#include "Events.h"

extern volatile word OnReceiveDataNum;
extern volatile word OnTransmitDataNum;
extern volatile word OnByteTransferNum;
extern volatile word OnArbitLostNum;
extern volatile word OnNACKNum;

extern volatile word OnSendRecvDataNum;
extern volatile word OnErrorNum;

/*
** ===================================================================
**     Event       :  I2C1_OnReceiveData (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         This event is invoked when I2C finishes the reception of
**         the data successfully. This event is not available for
**         the SLAVE mode.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
                       /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR)      */
void I2C1_OnReceiveData(void)
{
  OnReceiveDataNum++;
  OnSendRecvDataNum++;
}

/*
** ===================================================================
**     Event       :  I2C1_OnTransmitData (module Events)
**
```

```
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         This event is invoked when I2C finishes the transmission
**         of the data successfully. This event is not available for
**         the SLAVE mode.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
                       /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR)      */
void I2C1_OnTransmitData(void)
{
  OnTransmitDataNum++;
  OnSendRecvDataNum++;
}

/*
** ===================================================================
**     Event       :  I2C1_OnByteTransfer (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         This event is called when one-byte transfer (including
**         the acknowledge bit) is successfully finished (slave
**         address or one data byte is transmitted or received).
**         This event is not available for the SLAVE mode and it is
**         also provided if the interrupt service is disabled.
**         Note: It is possible to use the event for slowing down
**         communication, when slower slave needs some time for data
**         processing.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
                       /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR)      */
void I2C1_OnByteTransfer(void)
{
  OnByteTransferNum++;
}

/*
** ===================================================================
**     Event       :  I2C1_OnArbitLost (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         This event is called when the master lost the bus
**         arbitration or the device detects an error on the bus.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
```

```
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
                       /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR)      */
void I2C1_OnArbitLost(void)
{
  OnArbitLostNum++;
  OnErrorNum++;
}

/*
** ===================================================================
**     Event       :  I2C1_OnNACK (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         Called when a no slave acknowledge (NAK) occurs during
**         communication. This event is not available for the SLAVE
**         mode.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
                       /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR)      */
void I2C1_OnNACK(void)
{
  OnNACKNum++;
  OnErrorNum++;
}


/* END Events */

/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 0.00.00 [03.82]
**     for the Freescale 56800 series of microcontrollers.
**          !!! DEVELOPER VERSION FOR INTERNAL USAGE ONLY !!!
**
** ###################################################################
*/
```

2.3.2.3 MC56F8013 I2C1.c

The actual driver software generated by the tool for the I2C is shown here. There is no need for the user to modify it. Although it is possible to disable Processor Expert and make modifications, it is strongly recommended that only an expert user do so. It is better to be able to use the generated code directly, especially if development of new features is needed quickly.

It is not recommended for a user to modify code generated by the tool. It is shown only to explain how the driver works.

```
/** ###################################################################
**     THIS BEAN MODULE IS GENERATED BY THE TOOL. DO NOT MODIFY IT.
**     Filename  : I2C1.C
**     Project   : I2Cbean8013Fast
**     Processor : 56F8013VFAE
**     Beantype  : InternalI2C
**     Version   : Bean 01.171, Driver 01.06, CPU db: 2.87.090
**     Compiler  : Metrowerks DSP C Compiler
**     Date/Time : 8/29/2007, 10:35 AM
**     Abstract  :
**          This bean encapsulates the internal I2C communication
**          interface. The implementation of the interface is based
**          on the Philips I2C-bus specification version 2.0.
**          Interface features:
**          MASTER mode
**            - Multi master communication
**            - The combined format of communication possible
**              (see "Automatic stop condition" property)
**            - 7-bit slave addressing (10-bit addressing can be made as well)
**            - Acknowledge polling provided
**            - No wait state initiated when a slave device holds the SCL line low
**          - Holding of the SCL line low by slave device recognized as 'not available bus'
**            - Invalid start/stop condition detection provided
**          SLAVE mode
**            - 7-bit slave addressing
**            - General call address detection provided
**     Settings  :
**         Serial channel              : I2C
**
**         Protocol
**             Mode                    : MASTER
**             Auto stop condition     : yes
**             SCL frequency           : 400 kHz
**
**         Initialization
**
**             Slave address           : 80
**             Bean function           : Enabled
**             Events                  : Enabled
**
**         Registers
**             Input buffer            : IBDR      [61652]
**             Output buffer           : IBDR      [61652]
**             Control register        : IBCR      [61650]
**             Status register         : IBSR      [61651]
**             Baud setting reg.       : IBFD      [61649]
**             Address register        : IBAD      [61648]
**
**         Interrupt
**             Vector name             : INT_I2C
**             Priority                : 1
**
**         Used pins                   :
**       ----------------------------------------------------------
**           Function    | On package |    Name
**       ----------------------------------------------------------
**             SDA       |     2      |  GPIOB1_SS_B_SDA
```

```
**              SCL       |     21     |  GPIOB0_SCLK_SCL
**        ----------------------------------------------------------
**     Contents  :
**         SendBlock       - byte I2C1_SendBlock(void* Ptr,word Siz,word *Snt);
**         RecvBlock       - byte I2C1_RecvBlock(void* Ptr,word Siz,word *Rcv);
**         GetCharsInTxBuf - word I2C1_GetCharsInTxBuf(void);
**         SelectSlave     - byte I2C1_SelectSlave(byte Slv);
**         GetCharsInRxBuf - word I2C1_GetCharsInRxBuf(void);
**         GetSelected     - byte I2C1_GetSelected(byte *Slv);
**         GetMode         - bool I2C1_GetMode(void);
**         CheckBus        - byte I2C1_CheckBus(void);
**         ConnectPin      - void I2C1_ConnectPin(byte PinMask);
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2006
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
** ###################################################################*/


/* MODULE I2C1. */


#include "Events.h"
#include "I2C1.h"

#define OVERRUN_ERR      1            /* Overrun error flag bit   */
#define WAIT_RX_CHAR     2            /* Wait for received char. flag bit (Master)  */
#define CHAR_IN_TX       4            /* Char is in TX buffer (Master)    */
#define CHAR_IN_RX       8            /* Char is in RX buffer     */
#define FULL_TX          16           /* Full transmit buffer     */
#define IN_PROGRES       32           /* Communication is in progress (Master) */
#define FULL_RX          64           /* Full receive buffer      */
#define MSxSL            128          /* Master x Slave flag bit  */

volatile byte I2C1_SlaveAddr;         /* Variable for Slave address */
static word InpLenM;                  /* Length of input bufer's content */
static byte *InpPtrM;                 /* Pointer to input buffer for Master mode */
static word OutLenM;                  /* Length of output bufer's content */
static byte *OutPtrM;                 /* Pointer to output buffer for Master mode */
volatile byte I2C1_SerFlag;           /* Flags for serial communication */
                                      /* Bits: 0 - OverRun error */
                                /*       1 - Wait for received char. flag bit (Master) */
                                      /*       2 - Char is in TX buffer (Master) */
                                      /*       3 - Char in RX buffer */
                                      /*       4 - Full TX buffer */
                                      /*       5 - Running int from TX */
                                      /*       6 - Full RX buffer */
                                      /*       7 - Master x Slave */

/*
** ===================================================================
**     Method      :  I2C1_Interrupt (bean InternalI2C)
**
```

```
**     Description :
**         The method services the interrupt of the selected peripheral(s)
**         and eventually invokes the beans event(s).
**         This method is internal. It is used by Processor Expert only.
** ===================================================================
*/
#define RXAK 1
#define SRW  4
#define IBAL 16
#define IAAS 64

#define ON_ARBIT_LOST 1
#define ON_FULL_RX    2
#define ON_RX_CHAR    4
#define ON_FREE_TX    8
#define ON_TX_CHAR    16
#define ON_OVERRUN    32
#define ON_TX_EMPTY   64

#pragma interrupt alignsp saveall
void I2C1_Interrupt(void)
{
  register word Status = getReg(IBSR); /* Safe status register */

  setRegBit(IBSR,IBIF);                /* Clear interrupt flag */
  if (getRegBit(IBCR,MS_SL)) {         /* Is device in master mode? */
    I2C1_OnByteTransfer();             /* Invoke OnByteTransfer event */
    if (getRegBit(IBCR,TX_RX)) {       /* Is device in Tx mode? */
      if (Status & IBSR_RXAK_MASK) {   /* NACK received? */
       clrRegBit(IBCR,MS_SL);        /* Switch device to slave mode (stop signal sent) */
        OutLenM = 0;                   /* No character for sending */
        InpLenM = 0;                   /* No character for reception */
      I2C1_SerFlag &= ~(CHAR_IN_TX|WAIT_RX_CHAR|IN_PROGRES); /* No character for sending
or reception */
        I2C1_OnNACK();                 /* Invoke OnNACK event */
      }
      else {
        if (OutLenM) {                 /* Is any char. for transmitting? */
          OutLenM--;                   /* Decrease number of chars for the transmit */
          setReg(IBDR,*(OutPtrM)++);   /* Send character */
        }
        else {
          if (InpLenM) {               /* Is any char. for reception? */
            if (InpLenM == 1)          /* If only one char to receive */
              setRegBit(IBCR,TXAK);    /* then transmit ACK disable */
            else
              clrRegBit(IBCR,TXAK);    /* else transmit ACK anable */
            clrRegBit(IBCR,TX_RX);     /* Switch to Rx mode */
            getReg(IBDR);              /* Dummy read character */
          }
          else {
            I2C1_SerFlag &= ~IN_PROGRES; /* Clear flag "busy" */
           clrRegBit(IBCR,MS_SL);    /* Switch device to slave mode (stop signal sent) */
            I2C1_OnTransmitData();     /* Invoke OnTransmitData event */
          }
        }
      }
```

```
    }
    else {
      InpLenM--;                         /* Decrease number of chars for the receive */
      if (InpLenM) {                     /* Is any char. for reception? */
        if (InpLenM == 1)
          setRegBit(IBCR,TXAK);          /* Transmit ACK disable */
      }
      else {
       clrRegBit(IBCR,MS_SL);          /* If no, switch device to slave mode (stop signal
sent) */
        clrRegBit(IBCR,TXAK);            /* Transmit ACK enable */
      }
      *(InpPtrM)++ = (byte)getReg(IBDR); /* Receive character */
      if (!InpLenM) {                    /* Is any char. for reception? */
        I2C1_OnReceiveData();            /* Invoke OnReceiveData event */
      }
    }
  }
  else {
    if (Status & IBSR_IBAL_MASK) {       /* Arbitration lost? */
      OutLenM = 0;                       /* Any character is not for sent */
      InpLenM = 0;                       /* Any character is not for reception */
     I2C1_SerFlag &= ~(CHAR_IN_TX|WAIT_RX_CHAR|IN_PROGRES); /* Any character is not for
sent or reception*/
      I2C1_OnArbitLost();                /* Invoke OnArbitLost event */
    }
  }
}

/*
** ===================================================================
**     Method      :  I2C1_SendBlock (bean InternalI2C)
**
**     Description :
**         When working as a MASTER, this method writes one (7-bit
**         addressing) or two (10-bit addressing) slave address
**         bytes inclusive of R/W bit = 0 to the I2C bus and then
**         writes the block of characters to the bus. The slave
**         address must be specified before, by the "SelectSlave" or
**         "SlaveSelect10" method or in bean initialization section,
**         "Target slave address init" property. If interrupt
**         service is enabled and the method returns ERR_OK, it
**         doesn't mean that transmission was successful. The state
**         of transmission is detectable by means of events
**         (OnTransmitData, OnError or OnArbitLost). Data to be send
**         is not copied to an internal buffer and remains in the
**         original location. Therefore the content of the buffer
**         should not be changed until the transmission is complete.
**         Event OnTransmitData can be used to detect the end of the
**         transmission.
**         When working as a SLAVE, this method writes a block of
**         characters to the internal output slave buffer and then,
**         after the master starts the communication, to the I2C bus.
**         If no character is ready for a transmission (internal
**         output slave buffer is empty), the "Empty character" will
**         be sent (see "Empty character" property). In SLAVE mode
**         the data are copied to an internal buffer, if specified
```

```
**         by "Output buffer size" property.
**     Parameters  :
**         NAME            - DESCRIPTION
**       * Ptr             - Pointer to the block of data to send.
**         Siz             - Size of the block.
**       * Snt             - Amount of data sent (moved to a buffer).
**                           In master mode, if interrupt support is
**                           enabled, the parameter always returns
**                           the same value as the parameter 'Siz' of
**                           this method.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
**                           ERR_DISABLED -  Device is disabled
**                           ERR_BUSY - The slave device is busy, it
**                           does not respond by the acknowledge
**                           (only in master mode and when interrupt
**                           service is disabled)
**                           ERR_BUSOFF - Clock timeout elapsed or
**                           device cannot transmit data
**                           ERR_TXFULL - Transmitter is full. Some
**                           data has not been sent. (slave mode only)
**                           ERR_ARBITR - Arbitration lost (only when
**                           interrupt service is disabled and in
**                           master mode)
** ===================================================================
*/
byte I2C1_SendBlock(void* Ptr,word Siz,word *Snt)
{
  if (!Siz) {                          /* Test variable Size on zero */
    *Snt = 0;
    return ERR_OK;                     /* If zero then OK */
  }
  if
((getRegBit(IBSR,IBB))||(InpLenM)||(I2C1_SerFlag&(CHAR_IN_TX|WAIT_RX_CHAR|IN_PROGRES)))
{ /* Is the bus busy */
    return ERR_BUSOFF;                 /* If yes then error */
  }
  EnterCritical();                     /* Enter the critical section */
  I2C1_SerFlag |= IN_PROGRES;          /* Set flag "busy" */
  OutLenM = Siz;                       /* Set lenght of data */
  OutPtrM = (byte *)Ptr;               /* Save pointer to data for transmitting */
  setRegBit(IBCR,TX_RX);               /* Set TX mode */
  if (getRegBit(IBCR,MS_SL))           /* Is device in master mode? */
    setRegBit(IBCR,RSTA);              /* If yes then repeat start cycle generated */
  else
    setRegBit(IBCR,MS_SL);             /* If no then start signal generated */
  setReg(IBDR,I2C1_SlaveAddr);         /* Send slave address */
  ExitCritical();                      /* Exit the critical section */
  *Snt = Siz;                          /* Dummy number of really sent chars */
  return ERR_OK;                       /* OK */
}

/*
** ===================================================================
```

```
**     Method      :  I2C1_RecvBlock (bean InternalI2C)
**
**     Description :
**         When working as a MASTER, this method writes one (7-bit
**         addressing) or two (10-bit addressing) slave address
**         bytes inclusive of R/W bit = 1 to the I2C bus, then reads
**         the block of characters from the bus and then sends the
**         stop condition. The slave address must be specified
**         before, by the "SelectSlave" or "SelectSlave10" method or
**         in bean initialization section, "Target slave address
**         init" property. If interrupt service is enabled and the
**         method returns ERR_OK, it doesn't mean that transmission
**         was finished successfully. The state of transmission must
**         be tested by means of events (OnReceiveData, OnError or
**         OnArbitLost). In case of successful transmission,
**         received data is ready after OnReceiveData event is
**         called.
**         When working as a SLAVE, this method reads a block of
**         characters from the input slave buffer.
**     Parameters  :
**         NAME            - DESCRIPTION
**       * Ptr             - A pointer to the block space for
**                           received data.
**         Siz             - The size of the block.
**       * Rcv             - Amount of received data. In master mode,
**                           if interrupt support is enabled, the
**                           parameter always returns the same value
**                           as the parameter 'Siz' of this method.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
**                           ERR_DISABLED -  Device is disabled
**                           ERR_BUSY - The slave device is busy, it
**                           does not respond by an acknowledge (only
**                           in master mode and when interrupt
**                           service is disabled)
**                           ERR_BUSOFF - Clock timeout elapsed or
**                           device cannot receive data
**                           ERR_RXEMPTY - The receive buffer didn't
**                           contain the requested number of data.
**                           Only available data (or no data) has
**                           been returned  (slave mode only).
**                           ERR_OVERRUN - Overrun error was detected
**                           from last character or block receiving
**                           (slave mode only)
**                           ERR_ARBITR - Arbitration lost (only when
**                           interrupt service is disabled and in
**                           master mode)
** ===================================================================
*/
byte I2C1_RecvBlock(void* Ptr,word Siz,word *Rcv)
{
  if (!Siz) {                          /* Test variable Size on zero */
    *Rcv = 0;
    return ERR_OK;                     /* If zero then OK */
```

```
  }
  if
((getRegBit(IBSR,IBB))||(InpLenM)||(I2C1_SerFlag&(CHAR_IN_TX|WAIT_RX_CHAR|IN_PROGRES)))
{ /* Is the bus busy */
    return ERR_BUSOFF;                  /* If yes then error */
  }
  EnterCritical();                      /* Enter the critical section */
  InpLenM = Siz;                        /* Set lenght of data */
  InpPtrM = (byte *)Ptr;                /* Save pointer to data for reception */
  setRegBit(IBCR,TX_RX);                /* Set TX mode */
  if (getRegBit(IBCR,MS_SL))            /* Is device in master mode? */
    setRegBit(IBCR,RSTA);               /* If yes then repeat start cycle generated */
  else
    setRegBit(IBCR,MS_SL);              /* If no then start signal generated */
  setReg(IBDR,I2C1_SlaveAddr+1);        /* Send slave address */
  ExitCritical();                       /* Exit the critical section */
  *Rcv = Siz;                           /* Dummy number of really received chars */
  return ERR_OK;                        /* OK */
}

/*
** ===================================================================
**     Method      :  I2C1_GetCharsInTxBuf (bean InternalI2C)
**
**     Description :
**         Returns number of characters in the output buffer. In
**         SLAVE mode returns the number of characters in the
**         internal slave output buffer. In MASTER mode returns
**         number of characters to be sent from the user buffer
**         (passed by SendBlock method).
**         This method is not supported in polling mode.
**     Parameters  : None
**     Returns     :
**         ---             - Number of characters in the output
**                           buffer.
** ===================================================================
*/
word I2C1_GetCharsInTxBuf(void)
{
  return OutLenM;                       /* Return number of chars remaining in the Master
Tx buffer */
}

/*
** ===================================================================
**     Method      :  I2C1_GetCharsInRxBuf (bean InternalI2C)
**
**     Description :
**         Returns number of characters in the input buffer. In
**         SLAVE mode returns the number of characters in the
**         internal slave input buffer. In MASTER mode returns
**         number of characters to be received into a user buffer
**         (passed by RecvChar or RecvBlock method).
**         This method is not supported in polling mode.
**     Parameters  : None
**     Returns     :
**         ---             - Number of characters in the input
```

```
**                          buffer.
** ===================================================================
*/
word I2C1_GetCharsInRxBuf(void)
{
  return InpLenM;                      /* Return number of chars remaining in the Master
Rx buffer */
}

/*
** ===================================================================
**     Method      :  I2C1_SelectSlave (bean InternalI2C)
**
**     Description :
**         This method selects a new slave for communication by its
**         7-bit slave address value. Any send or receive method
**         directs to or from selected device, until a new slave
**         device is selected by this method. This method is not
**         available for the SLAVE mode.
**     Parameters  :
**         NAME            - DESCRIPTION
**         Slv             - 7-bit slave address value.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_BUSY - The device is busy, wait
**                           until the current operation is finished.
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
**                           ERR_DISABLED -  The device is disabled
** ===================================================================
*/
byte I2C1_SelectSlave(byte Slv)
{
  I2C1_SlaveAddr = (byte)(Slv << 1);   /* Set slave address */
  return ERR_OK;                       /* OK */
}

/*
** ===================================================================
**     Method      :  I2C1_GetSelected (bean InternalI2C)
**
**     Description :
**         This method returns 7-bit slave address value of the
**         slave, which is currently selected for communication with
**         the master. This method is not available for the SLAVE
**         mode.
**     Parameters  :
**         NAME            - DESCRIPTION
**       * Slv             - Current selected slave address value.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
** ===================================================================
*/
```

```
byte I2C1_GetSelected(byte *Slv)
{
  *Slv = (byte)(I2C1_SlaveAddr >> 1);  /* Get slave address */
  return ERR_OK;                       /* OK */
}

/*
** ===================================================================
**     Method      :  I2C1_GetMode (bean InternalI2C)
**
**     Description :
**         This method returns the actual operating mode of this
**         bean.
**     Parameters  : None
**     Returns     :
**         ---             - Actual operating mode value
**                           TRUE - Master
**                           FALSE - Slave
** ===================================================================
*/
/*
bool I2C1_GetMode(void)

**  This method is implemented as a macro. See I2C1.h file.  **
*/

/*
** ===================================================================
**     Method      :  I2C1_ConnectPin (bean InternalI2C)
**
**     Description :
**         This method reconnects requested pin associated with the
**         selected peripheral in this bean. This method is available
**         only for CPU derivatives and peripherals that support runtime
**         pin sharing with other internal on-chip peripherals.
**     Parameters  :
**         NAME            - DESCRIPTION
**         PinMask         - Mask for the requested pins.
**                           The peripheral pins are reconnected
**                           according to this mask.
**                           Possible parameters:
**                           I2C1_IN_PIN     - Input pin
**                           I2C1_OUT_PIN  - Output pin
**                           I2C1_CLK_PIN   - Clock pin
**                           I2C1_SS_PIN   - Slave select pin
**     Returns     : Nothing
** ===================================================================
*/
/*
void I2C1_ConnectPin(byte PinMask)

**  This method is implemented as a macro. See I2C1.h file.  **
*/

/*
** ===================================================================
**     Method      :  I2C1_Init (bean InternalI2C)
```

```
**
**     Description :
**         Initializes the associated peripheral(s) and the beans
**         internal variables. The method is called automatically as a
**         part of the application initialization code.
**         This method is internal. It is used by Processor Expert only.
** ===================================================================
*/
void I2C1_Init(void)
{
  /* IBCR: IBEN=0,IBIE=0,MS_SL=0,TX_RX=0,TXAK=0,RSTA=0,??=0,??=0 */
  setReg(IBCR,0);                      /* Clear control register */
  I2C1_SerFlag = 128;                  /* Reset all flags */
  I2C1_SlaveAddr = 160;                /* Set variable for slave address */
  setReg(IBNR,5);                      /* Set Noise filter register */
  /* IBFD: IBC=128 */
  setReg(IBFD,128);                    /* Set prescaler bits */
  setRegBit(IBCR,IBEN);                /* Enable device */
  /* IBCR: IBEN=1,IBIE=1,MS_SL=0,TX_RX=0,TXAK=0,RSTA=0,??=0,??=0 */
  setReg(IBCR,192);                    /* Control register settings */
}

/*
** ===================================================================
**     Method      :  I2C1_CheckBus (bean InternalI2C)
**
**     Description :
**         This method returns the status of the bus. If the START
**         condition has been detected, the method returns
**         'BeanName'_BUSY. If the STOP condition has been detected,
**         the method returns 'BeanName'_IDLE.
**     Parameters  : None
**     Returns     :
**         ---             - Status of the bus.
** ===================================================================
*/
/*
byte I2C1_CheckBus(void)

**  This method is implemented as a macro. See I2C1.h file.  **
*/


/* END I2C1. */

/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 2.98.02 [03.79]
**     for the Freescale 56800 series of microcontrollers.
**
** ###################################################################
*/
```

3 MC56F8037 Interfacing to M24C64 Over I2C Bus

3.1 System Description

The system consists of:

- MC56F8037 EVM DSC evaluation board. A full description is available at www.freescale.com. Access to the IIC SDA and SCL is available at the daughter card connector. Please refer to MC56F8037EVMUM, *56F8037 Evaluation Module User Manual*, also available at www.freescale.com. (Hint: Be sure to connect to PTB0/SCL and PTB1/SDA.)
- Daughter card with M24C64 EEPROM device — this supplies the system with a remote serial memory interface connected via the I2C bus, a two-wire interface
- Software generated by the Processor Expert code generator included with CodeWarrior — the software runs a test of the memory device over the I2C bus

3.2 Hardware

The hardware consists of the MC56F8037 EVM board and a breadboard connected to the EVM board with twisted pair wiring. The breadboard contains the M24C64 EEPROM device. The daughter card wiring is as shown in Figure 3 and Figure 4 (viewed from underneath board).

To wire the daughter card to the EVM board, three twisted-pair wires are used. The wiring and daughter board are similar to the MC56F8013 case.

3.3 Software

The directions given in Section 2.3, "Software," can be followed to create a new project for the MCF8037.

The software consists of:

- First, at the bean level or highest level, the parameters that feed into the C code generator (Processor Expert) to generate the code
- Second, the generated code will be presented at the C level, starting with the main program, and descending down through the call hierarchy to the driver code:
  - main top-level C routine supplied by application designer
  - Events.c — high-level user-suppled events routine supplied by application designer
  - I2C1.c low-level code generated by Processor Expert in C language, based on GUI inputs

3.3.1 Processor Expert Bean Parameters

For the MC56F8013, the I2C peripheral is served by an I2C hardware bean, configured as described in the next subsections.

3.3.1.1 I2C1 Internal I²C Hardware Bean Properties

The properties of the I²C bean have been selected via the GUI interface for this application as shown on the left side of Figure 8. The bean's properties determine how the DSC peripheral registers for the I²C peripheral will be initialized, as shown on the right side of Figure 8.

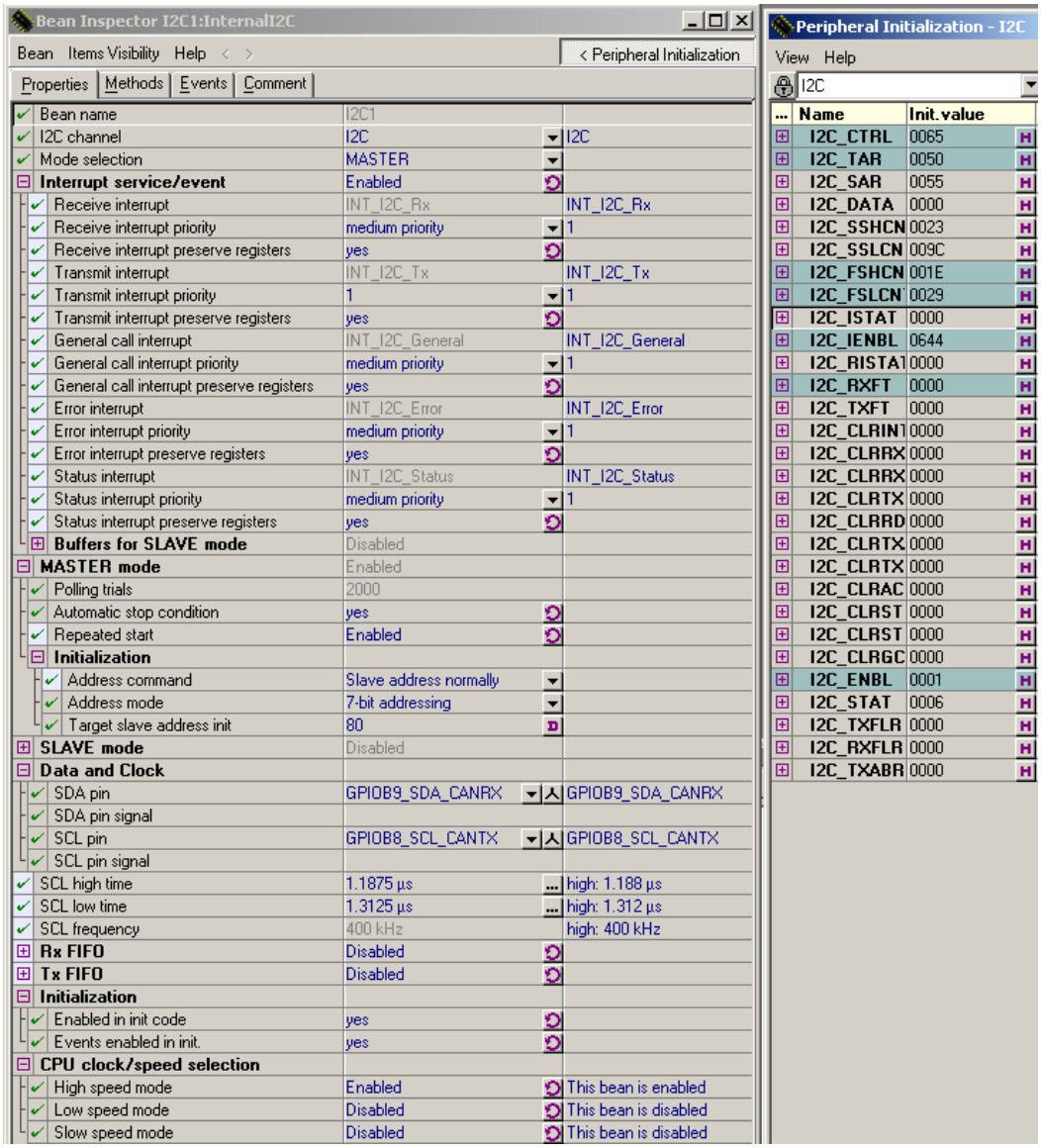

Figure 8.

3.3.1.2 I2C1 Internal I²C Hardware Bean Methods

This application needs only some of the available methods, or driver calls, for this peripheral. These are checked as shown in the right portion of Figure 9. For comparison, the methods of the MC56F8013 I²C

bean are shown to the left. Notice that the same methods are used for the MC56F8037 and the MC56F8013.

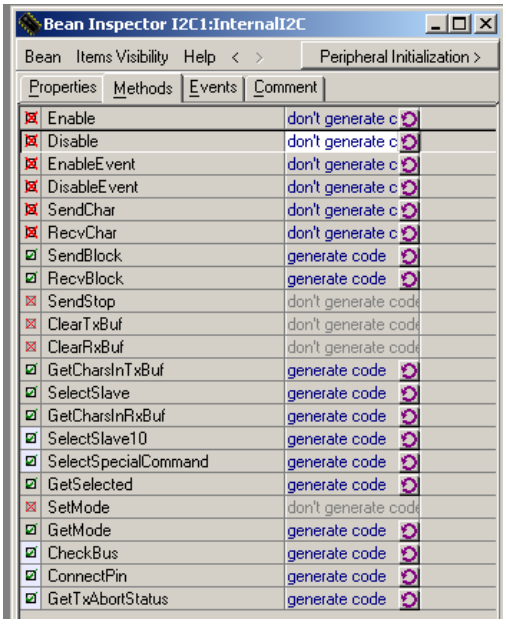

Figure 9.

3.3.1.3 I2C1 Internal I2C Hardware Bean Events

The events for this family differ slightly in division of functionality. There are options for events. In the right portion of Figure 10 are shown the events available for the MC56F8037. On the left, shown for comparison, are the events available on the MC56F8013. Note that only the events checked in green are used in this application. Even though the events differ between the two families the same functionality can be achieved using Processor Expert and slight code changes.

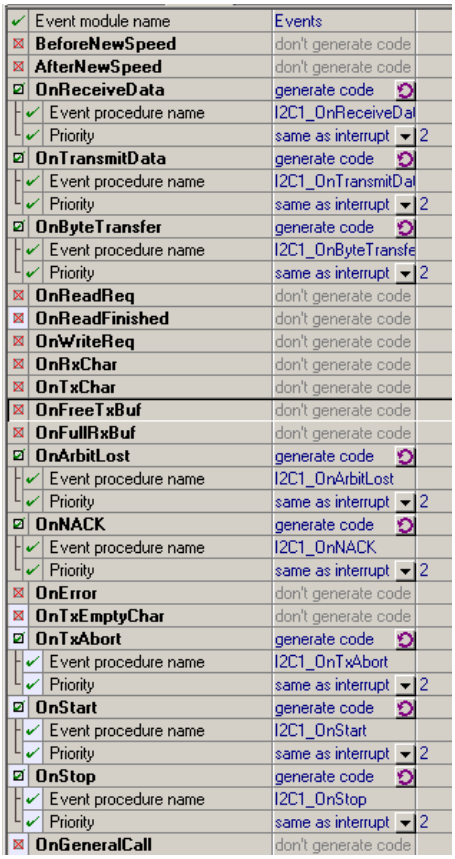

Figure 10.

3.3.2 MC56F8037 C Code with Annotation

So, what code is needed to perform this same function on the other family? Migration is trivial, because the main programs are nearly identical. The differences lie only in the lower level.

This lower level code is generated by Processor Expert. A comparison with the code above will reveal that the code shown here is nearly the same. Changes are extremely concentrated in the driver code generated automatically by Processor Expert. Note that although the available methods and events for the MC56F8037 are more diverse, porting an application from the MC56F8013 does not imply that these new methods must all be used. This should be kept in mind if migrating from the MC56F8013 to the MC56F8037.

3.3.2.1 MC56F8037 Main Program

The changes highlighted in red in Section 2.3.2.1, "MC56F8013 Main Program," to perform 16-bit address must be added to the 56MCF8037 main program. Follow the directions from section 2.3.2.1 to perform the changes. With these changes implemented there is little additional change to the main program. Only the changes are shown below.

Code not shown is indicated by an ellipsis. Code that has been added automatically by Processor Expert is made bold. Code that has changed from the MC56F8013 version is noted with comments. Code that the user must change is highlighted in red.

Code is the same starting at the beginning of the code.

...

```
volatile word OnTxAbortNum = 0;
volatile word OnStartNum = 0;
volatile word OnStopNum = 0;
```

...

The test comment has new instructions for connecting the pins due to the different pins on the different device:

```
TEST1_testComment(&testRec, "Please interconnect the pins GPIOB8_SCL_CANTX and
GPIOB9_SDA_CANRX with 24xx08 serial EEPROM.");
```

...

A new construct is used to check if the bus is busy (rather than using I2C1_BUSOFF, I2C1_BUSY is used.) This change is in several places, but is always the same and is only shown once here:

```
while (I2C1_CheckBus() == I2C1_BUSY) {}
```

...

And because the event code differs, different statistics appear at the end of the successful test, checked here:

```
/* Event test */
  Error = FALSE;
  if (OnReceiveDataNum != 2) {
    Error = TRUE;
  }
  if (OnTransmitDataNum != 5) {
    Error = TRUE;
  }
  if (OnByteTransferNum != 49) {
    Error = TRUE;
  }
  if (OnArbitLostNum != 0) {
    Error = TRUE;
  }
  if (OnNACKNum != 1) {
    Error = TRUE;
  }
```

Note the use of the increased delineation of reported events:

```
if (OnTxAbortNum != 1) {
  Error = TRUE;
}
if (OnStartNum != 7) {
  Error = TRUE;
}
if (OnStopNum != 7) {
  Error = TRUE;
```

```
  }
  if (!Error) {
    TEST1_testComment(&testRec, "Event test OK.");
  }
  else {
    TEST1_testFailed(&testRec, "Event test.");
  }

  TEST1_testEnd(&testRec);
}
```

3.3.2.2 MC56F8037 Events.c

Here we see the events, and how they are used for this ported application. Most of them are used to keep track of statistics during the test. Functions for events I2C1_OnReceiveData, I2C1_OnTransmitData, I2C1_OnByteTransfer are the same for both the MC56F8013 and the MC56F8037. All the other events are slightly modified for the MC56F8037. Three new events for the MC56F8037 highlight its ability to delineate: I2C1_OnStart, I2C1_OnStop, I2C1_OnTxAbort. The event code is shown below, minus the three functions that were the same for both devices. Bold text indicates code not present or a difference in the case of the MC56F8013 device:

```
#include "Cpu.h"
#include "Events.h"

extern volatile word OnReceiveDataNum;
extern volatile word OnTransmitDataNum;
extern volatile word OnByteTransferNum;
extern volatile word OnArbitLostNum;
extern volatile word OnNACKNum;
extern volatile word OnTxAbortNum;
extern volatile word OnStartNum;
extern volatile word OnStopNum;

extern volatile word OnSendRecvDataNum;
extern volatile word OnErrorNum;

I2C1_TTxAbortStatus Status;
```

(Three modules with same code as MC56F8013 not shown here.)

...

```
/*
** ===================================================================
**     Event       :  I2C1_OnArbitLost (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         This event is called when the master lost the bus
**         arbitration or the device detects an error on the bus.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
```

```
                        /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR)      */
void I2C1_OnArbitLost(void)
{
  OnArbitLostNum++;
```

OnErrorNum is not incremented here, but was in the case of the MC56F8013.

```
}

/*
** ===================================================================
**     Event       :  I2C1_OnNACK (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         Called when a no slave acknowledge (NAK) occurs during
**         communication. This event is not available for the SLAVE
**         mode.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
                        /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR)      */
void I2C1_OnNACK(void)
{
  OnNACKNum++;
```

OnErrorNum is not incremented here, but was in the case of the MC56F8013.

```
}

/*
** ===================================================================
**     Event       :  I2C1_OnTxAbort (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         This event is called when the transmit aborted.
**         Conditions can be read using <GetTxAbortStatus> method.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
void I2C1_OnTxAbort(void)
{
  I2C1_GetTxAbortStatus(&Status) != ERR_OK;
  OnTxAbortNum++;
  OnErrorNum++;
}

/*
** ===================================================================
**     Event       :  I2C1_OnStart (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
```

```
**     Description :
**         This event is called when the Start detected on the bus.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
void I2C1_OnStart(void)
{
  OnStartNum++;
}

/*
** ===================================================================
**     Event       :  I2C1_OnStop (module Events)
**
**     From bean   :  I2C1 [InternalI2C]
**     Description :
**         This event is called when the Stop detected on the bus.
**     Parameters  : None
**     Returns     : Nothing
** ===================================================================
*/
void I2C1_OnStop(void)
{
  OnStopNum++;
}
```

3.3.2.3 MC56F8037 I2C1.c

The extensive driver code generated by Processor Expert for this application is printed here for reference. The methods (API for functions to perform) are provided here, as well as the interrupt service routines that call the events.

```
/** ###################################################################
**     THIS BEAN MODULE IS GENERATED BY THE TOOL. DO NOT MODIFY IT.
**     Filename  : I2C1.C
**     Project   : I2Cbean8037Fast
**     Processor : 56F8037
**     Beantype  : InternalI2C
**     Version   : Bean 01.171, Driver 01.07, CPU db: 2.87.240
**     Compiler  : Metrowerks DSP C Compiler
**     Date/Time : 8/24/2007, 6:16 PM
**     Abstract  :
**          This bean encapsulates the internal I2C communication
**          interface. The implementation of the interface is based
**          on the Philips I2C-bus specification version 2.0.
**          Interface features:
**          MASTER mode
**            - Multi master communication
**            - The combined format of communication possible
**              (see "Automatic stop condition" property)
**            - 7-bit slave addressing (10-bit addressing can be made as well)
**            - Acknowledge polling provided
**            - No wait state initiated when a slave device holds the SCL line low
**          - Holding of the SCL line low by slave device recognized as 'not available bus'
**            - Invalid start/stop condition detection provided
**          SLAVE mode
```

```
**            - 7-bit slave addressing
**            - General call address detection provided
**     Settings  :
**         Serial channel              : I2C
**
**         Protocol
**             Mode                    : MASTER
**             Auto stop condition     : yes
**             SCL frequency           : 400 kHz
**
**         Initialization
**
**             Slave address           : 80
**             Bean function           : Enabled
**             Events                  : Enabled
**
**         Registers
**             Input buffer            : I2C_DATA  [F288]
**             Output buffer           : I2C_DATA  [F288]
**             Control register        : I2C_CTRL  [F280]
**             Status register         : I2C_RISTAT [F29A]
**             Address register        : I2C_SAR   [F284]
**
**
**         Used pins                   :
**       ----------------------------------------------------------
**             Function    | On package |    Name
**       ----------------------------------------------------------
**              SDA        |     46     |  GPIOB9_SDA_CANRX
**              SCL        |     54     |  GPIOB8_SCL_CANTX
**       ----------------------------------------------------------
**     Contents  :
**         SendBlock            - byte I2C1_SendBlock(void* Ptr,word Siz,word *Snt);
**         RecvBlock            - byte I2C1_RecvBlock(void* Ptr,word Siz,word *Rcv);
**         GetCharsInTxBuf      - word I2C1_GetCharsInTxBuf(void);
**         SelectSlave          - byte I2C1_SelectSlave(byte Slv);
**         GetCharsInRxBuf      - word I2C1_GetCharsInRxBuf(void);
**         SelectSlave10        - byte I2C1_SelectSlave10(word Slv);
**         SelectSpecialCommand - byte I2C1_SelectSpecialCommand(byte Cmd);
**         GetSelected          - byte I2C1_GetSelected(byte *Slv);
**         GetMode              - bool I2C1_GetMode(void);
**         CheckBus             - byte I2C1_CheckBus(void);
**         ConnectPin           - void I2C1_ConnectPin(byte PinMask);
**        GetTxAbortStatus    - byte I2C1_GetTxAbortStatus(I2C1_TTxAbortStatus *Status);
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2006
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
** ###################################################################*/


/* MODULE I2C1. */
```

```
#include "Events.h"
#include "I2C1.h"

#define OVERRUN_ERR      0x01          /* Overrun error flag bit   */
#define WAIT_RX_CHAR     0x02          /* Wait for received char. flag bit (Master)  */
#define CHAR_IN_TX       0x04          /* Char is in TX buffer (Master)    */
#define CHAR_IN_RX       0x08          /* Char is in RX buffer     */
#define FULL_TX          0x10          /* Full transmit buffer     */
#define SLAVE_IN_TX      0x20          /* Slave in transmit mode */
#define FULL_RX          0x40          /* Full receive buffer      */
#define MSxSL            0x80          /* Master x Slave flag bit  */
#define SLAVE_IN_RX      0x0100        /* Slave in receive mode */
#define MASTER_IN_TX     0x0200        /* Master in transmit mode */

static word InpLenM;                   /* Length of input bufer's content */
static word InpLenMTx;                /* Length of input bufer's content (for transmit
interrupt) */
static byte *InpPtrM;                  /* Pointer to input buffer for Master mode */
static word OutLenM;                   /* Length of output bufer's content */
static byte *OutPtrM;                  /* Pointer to output buffer for Master mode */
volatile word I2C1_SerFlag;            /* Flags for serial communication */
                                       /* Bits: 0 - OverRun error */
                                /*       1 - Wait for received char. flag bit (Master) */
                                       /*       2 - Char is in TX buffer (Master) */
                                       /*       3 - Char in RX buffer */
                                       /*       4 - Full TX buffer */
                                       /*       5 - Slave in transmit mode */
                                       /*       6 - Full RX buffer */
                                       /*       7 - Master x Slave */
                                       /*       8 - Slave in receive mode */
                                       /*       9 - Master in transmit mode */
static word TxAbortStatus;             /* Tx abort flags */



#define ON_ARBIT_LOST    0x01
#define ON_FULL_RX       0x02
#define ON_RX_CHAR       0x04
#define ON_FREE_TX       0x08
#define ON_TX_CHAR       0x10
#define ON_OVERRUN       0x20
#define ON_TX_EMPTY      0x40
#define ON_TX_DATA       0x80
#define ON_RX_DATA       0x80
#define ON_BYTE_TRANSFER 0x01
#define ON_READ_FINISHED 0x02
#define ON_NACK          0x02
#define ON_TX_ABORT      0x04
#define ON_START         0x01
#define ON_STOP          0x02

/*
** ===================================================================
**     Method      :  I2C1_InterruptRx (bean InternalI2C)
**
**     Description :
```

```
**         This method is internal. It is used by Processor Expert only.
** ===================================================================
*/
#pragma interrupt alignsp saveall
void I2C1_InterruptRx(void)
{
  register byte Flags = 0;             /* Temporary variable for flags */

  InpLenM--;                           /* Decrease number of chars for the receive */
  *(InpPtrM)++ = (byte)getReg(I2C_DATA); /* Receive character */
  Flags |= ON_BYTE_TRANSFER;           /* Set OnByteTransfer flag */
  if (!InpLenM) {                      /* Is any char. for reception? */
    Flags |= ON_RX_DATA;               /* Set OnReceiveData flag */
  }
  if (Flags & ON_BYTE_TRANSFER) {      /* Is OnByteTransfer flag set? */
    I2C1_OnByteTransfer();             /* If yes then invoke user event */
  }
  if (Flags & ON_RX_DATA) {            /* Is OnReceiveData flag set? */
    I2C1_OnReceiveData();              /* Invoke OnReceiveData event */
  }
}

/*
** ===================================================================
**     Method      :  I2C1_InterruptTx (bean InternalI2C)
**
**     Description :
**         This method is internal. It is used by Processor Expert only.
** ===================================================================
*/
#pragma interrupt alignsp saveall
void I2C1_InterruptTx(void)
{
  register byte Flags = 0;             /* Temporary variable for flags */

  if (getRegBit(I2C_ISTAT,TXEMPTY)) {  /* Is transmitter empty? */
    if (OutLenM) {                     /* Is any char. for transmitting? */
      OutLenM--;                       /* Decrease number of chars for the transmit */
      setReg(I2C_DATA,*(OutPtrM)++);   /* Send character */
      if (I2C1_SerFlag & MASTER_IN_TX) { /* Is Master in Tx flag set? */
        Flags |= ON_BYTE_TRANSFER;     /* Set OnByteTransfer flag */
      }
      else {
        I2C1_SerFlag |= MASTER_IN_TX;  /* Set MASTER_IN_TX flag */
      }
    }
    else {
      if (InpLenMTx) {                 /* Is any char. for reception? */
        InpLenMTx--;                   /* Decrease number of chars for the receive */
        setReg(I2C_DATA,0x0100);       /* Receive character */
        if (!InpLenMTx) {              /* Is not any char. for transmitting? */
          clrRegBit(I2C_IENBL,TXEMPTY); /* Disable TXEMPTY interrupt */
        }
      }
      else {
        clrRegBit(I2C_IENBL,TXEMPTY);  /* Disable TXEMPTY interrupt */
        Flags |= ON_TX_DATA;           /* Set OnTransmitData flag */
```

```
        Flags |= ON_BYTE_TRANSFER;     /* Set OnByteTransfer flag */
        I2C1_SerFlag &= ~MASTER_IN_TX; /* Clear MASTER_IN_TX flag */
      }
    }
  }
  if (Flags & ON_BYTE_TRANSFER) {      /* Is OnByteTransfer flag set? */
    I2C1_OnByteTransfer();             /* If yes then invoke user event */
  }
  if (Flags & ON_TX_DATA) {            /* Is OnTransmitData flag set? */
    I2C1_OnTransmitData();             /* Invoke OnTransmitData event */
  }
}

/*
** ===================================================================
**     Method      :  I2C1_InterruptError (bean InternalI2C)
**
**     Description :
**         This method is internal. It is used by Processor Expert only.
** ===================================================================
*/
#pragma interrupt alignsp saveall
void I2C1_InterruptError(void)
{
  register byte Flags = 0;             /* Temporary variable for flags */
  register word Status;                /* Safe status register */

  Status = getReg(I2C_TXABRTSRC);
  if (getReg(I2C_CLRTXABRT)) {         /* Is transmit aborted? */
    clrRegBit(I2C_IENBL,TXEMPTY);      /* Disable TXEMPTY interrupt */
    I2C1_SerFlag &= ~(WAIT_RX_CHAR | CHAR_IN_TX | MASTER_IN_TX); /* Clear WAIT_RX_CHAR,
CHAR_IN_TX and MASTER_IN_TX flag */
    OutLenM = 0;                       /* Any character is not for sent */
    InpLenM = 0;                       /* Any character is not for reception */
    if (Status & I2C_TXABRTSRC_AL_MASK) {
      Flags |= ON_ARBIT_LOST;          /* Set OnArbitLost flag */
    }
    if (Status & (I2C_TXABRTSRC_GCNACK_MASK | I2C_TXABRTSRC_TDNACK_MASK |
I2C_TXABRTSRC_AD2NACK_MASK | I2C_TXABRTSRC_AD1NACK_MASK | I2C_TXABRTSRC_AD7NACK_MASK)) {
      Flags |= ON_NACK;                /* Set OnNACK flag */
    }
    TxAbortStatus |= Status;
    Flags |= ON_TX_ABORT;              /* Set OnTxAbort flag */
  }
  if (Flags & ON_ARBIT_LOST) {         /* Is OnArbitLost flag set? */
    I2C1_OnArbitLost();                /* If yes then invoke user event */
  }
  if (Flags & ON_NACK) {               /* Is OnNACK flag set? */
    I2C1_OnNACK();                     /* If yes then invoke user event */
  }
  if (Flags & ON_TX_ABORT) {           /* Is OnTxAbort flag set? */
    I2C1_OnTxAbort();                  /* If yes then invoke user event */
  }
}

/*
** ===================================================================
```

```
**     Method      :  I2C1_InterruptStatus (bean InternalI2C)
**
**     Description :
**         This method is internal. It is used by Processor Expert only.
** ===================================================================
*/
#pragma interrupt alignsp saveall
void I2C1_InterruptStatus(void)
{
  register byte Flags = 0;             /* Temporary variable for flags */

  if (getReg(I2C_CLRSTDET)) {          /* Is start detected? */
    Flags |= ON_START;                 /* Set OnStart flag */
  }
  if (getReg(I2C_CLRSTPDET)) {         /* Is stop detected? */
    Flags |= ON_STOP;                  /* Set OnStop flag */
  }
  if (Flags & ON_START) {              /* Is OnStart flag set? */
    I2C1_OnStart();                    /* If yes then invoke user event */
  }
  if (Flags & ON_STOP) {               /* Is OnStop flag set? */
    I2C1_OnStop();                     /* If yes then invoke user event */
  }
}

/*
** ===================================================================
**     Method      :  I2C1_SendBlock (bean InternalI2C)
**
**     Description :
**         When working as a MASTER, this method writes one (7-bit
**         addressing) or two (10-bit addressing) slave address
**         bytes inclusive of R/W bit = 0 to the I2C bus and then
**         writes the block of characters to the bus. The slave
**         address must be specified before, by the "SelectSlave" or
**         "SlaveSelect10" method or in bean initialization section,
**         "Target slave address init" property. If interrupt
**         service is enabled and the method returns ERR_OK, it
**         doesn't mean that transmission was successful. The state
**         of transmission is detectable by means of events
**         (OnTransmitData, OnError or OnArbitLost). Data to be send
**         is not copied to an internal buffer and remains in the
**         original location. Therefore the content of the buffer
**         should not be changed until the transmission is complete.
**         Event OnTransmitData can be used to detect the end of the
**         transmission.
**         When working as a SLAVE, this method writes a block of
**         characters to the internal output slave buffer and then,
**         after the master starts the communication, to the I2C bus.
**         If no character is ready for a transmission (internal
**         output slave buffer is empty), the "Empty character" will
**         be sent (see "Empty character" property). In SLAVE mode
**         the data are copied to an internal buffer, if specified
**         by "Output buffer size" property.
**     Parameters  :
**         NAME            - DESCRIPTION
**       * Ptr             - Pointer to the block of data to send.
```

```
**         Siz             - Size of the block.
**       * Snt             - Amount of data sent (moved to a buffer).
**                           In master mode, if interrupt support is
**                           enabled, the parameter always returns
**                           the same value as the parameter 'Siz' of
**                           this method.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
**                           ERR_DISABLED -  Device is disabled
**                           ERR_BUSY - The slave device is busy, it
**                           does not respond by the acknowledge
**                           (only in master mode and when interrupt
**                           service is disabled)
**                           ERR_BUSOFF - Clock timeout elapsed or
**                           device cannot transmit data
**                           ERR_TXFULL - Transmitter is full. Some
**                           data has not been sent. (slave mode only)
**                           ERR_ARBITR - Arbitration lost (only when
**                           interrupt service is disabled and in
**                           master mode)
** ===================================================================
*/
byte I2C1_SendBlock(void* Ptr,word Siz,word *Snt)
{
  if (!Siz) {                          /* Test variable Size on zero */
    *Snt = 0;
    return ERR_OK;                     /* If zero then OK */
  }
  if ((getRegBit(I2C_STAT,ACT)) || (InpLenM)) { /* Is the bus busy */
    return ERR_BUSOFF;                 /* If yes then error */
  }
  EnterCritical();                     /* Enter the critical section */
  OutLenM = Siz;                       /* Set lenght of data */
  OutPtrM = (byte *)Ptr;               /* Save pointer to data for transmitting */
  setRegBit(I2C_IENBL,TXEMPTY);        /* Enable TXEMPTY interrupt */
  ExitCritical();                      /* Exit the critical section */
  *Snt = Siz;                          /* Dummy number of really sent chars */
  return ERR_OK;                       /* OK */
}

/*
** ===================================================================
**     Method      :  I2C1_RecvBlock (bean InternalI2C)
**
**     Description :
**         When working as a MASTER, this method writes one (7-bit
**         addressing) or two (10-bit addressing) slave address
**         bytes inclusive of R/W bit = 1 to the I2C bus, then reads
**         the block of characters from the bus and then sends the
**         stop condition. The slave address must be specified
**         before, by the "SelectSlave" or "SelectSlave10" method or
**         in bean initialization section, "Target slave address
**         init" property. If interrupt service is enabled and the
**         method returns ERR_OK, it doesn't mean that transmission
```

```
**         was finished successfully. The state of transmission must
**         be tested by means of events (OnReceiveData, OnError or
**         OnArbitLost). In case of successful transmission,
**         received data is ready after OnReceiveData event is
**         called.
**         When working as a SLAVE, this method reads a block of
**         characters from the input slave buffer.
**     Parameters  :
**         NAME            - DESCRIPTION
**       * Ptr             - A pointer to the block space for
**                           received data.
**         Siz             - The size of the block.
**       * Rcv             - Amount of received data. In master mode,
**                           if interrupt support is enabled, the
**                           parameter always returns the same value
**                           as the parameter 'Siz' of this method.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
**                           ERR_DISABLED -  Device is disabled
**                           ERR_BUSY - The slave device is busy, it
**                           does not respond by an acknowledge (only
**                           in master mode and when interrupt
**                           service is disabled)
**                           ERR_BUSOFF - Clock timeout elapsed or
**                           device cannot receive data
**                           ERR_RXEMPTY - The receive buffer didn't
**                           contain the requested number of data.
**                           Only available data (or no data) has
**                           been returned  (slave mode only).
**                           ERR_OVERRUN - Overrun error was detected
**                           from last character or block receiving
**                           (slave mode only)
**                           ERR_ARBITR - Arbitration lost (only when
**                           interrupt service is disabled and in
**                           master mode)
** ===================================================================
*/
byte I2C1_RecvBlock(void* Ptr,word Siz,word *Rcv)
{
  if (!Siz) {                          /* Test variable Size on zero */
    *Rcv = 0;
    return ERR_OK;                     /* If zero then OK */
  }
  if ((getRegBit(I2C_STAT,ACT)) || (InpLenM)) { /* Is the bus busy */
    return ERR_BUSOFF;                 /* If yes then error */
  }
  EnterCritical();                     /* Enter the critical section */
  InpLenM = Siz;                       /* Set lenght of data */
  InpLenMTx = Siz;                     /* Set lenght of data (for transmit interrupt)*/
  InpPtrM = (byte *)Ptr;               /* Save pointer to data for reception */
  setRegBit(I2C_IENBL,TXEMPTY);        /* Enable TXEMPTY interrupt */
  ExitCritical();                      /* Exit the critical section */
  *Rcv = Siz;                          /* Dummy number of really received chars */
  return ERR_OK;                       /* OK */
```

```
}

/*
** ===================================================================
**     Method      :  I2C1_GetCharsInTxBuf (bean InternalI2C)
**
**     Description :
**         Returns number of characters in the output buffer. In
**         SLAVE mode returns the number of characters in the
**         internal slave output buffer. In MASTER mode returns
**         number of characters to be sent from the user buffer
**         (passed by SendBlock method).
**         This method is not supported in polling mode.
**     Parameters  : None
**     Returns     :
**         ---             - Number of characters in the output
**                           buffer.
** ===================================================================
*/
word I2C1_GetCharsInTxBuf(void)
{
  return OutLenM;                      /* Return number of chars remaining in the Master
Tx buffer */
}

/*
** ===================================================================
**     Method      :  I2C1_GetCharsInRxBuf (bean InternalI2C)
**
**     Description :
**         Returns number of characters in the input buffer. In
**         SLAVE mode returns the number of characters in the
**         internal slave input buffer. In MASTER mode returns
**         number of characters to be received into a user buffer
**         (passed by RecvChar or RecvBlock method).
**         This method is not supported in polling mode.
**     Parameters  : None
**     Returns     :
**         ---             - Number of characters in the input
**                           buffer.
** ===================================================================
*/
word I2C1_GetCharsInRxBuf(void)
{
  return InpLenM;                      /* Return number of chars remaining in the Master
Rx buffer */
}

/*
** ===================================================================
**     Method      :  I2C1_SelectSlave (bean InternalI2C)
**
**     Description :
**         This method selects a new slave for communication by its
**         7-bit slave address value. Any send or receive method
**         directs to or from selected device, until a new slave
**         device is selected by this method. This method is not
```

```
**         available for the SLAVE mode.
**     Parameters  :
**         NAME            - DESCRIPTION
**         Slv             - 7-bit slave address value.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_BUSY - The device is busy, wait
**                           until the current operation is finished.
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
**                           ERR_DISABLED -  The device is disabled
** ===================================================================
*/
byte I2C1_SelectSlave(byte Slv)
{
  if ((getRegBit(I2C_STAT,MSTACT)) || (!getRegBit(I2C_STAT,TFE))) { /* Is the device in
the activ state? */
    return ERR_BUSY;                   /* If yes then error */
  }
  setReg16(I2C_TAR, (word)Slv);        /* Set slave address */
  return ERR_OK;                       /* OK */
}

/*
** ===================================================================
**     Method      :  I2C1_SelectSlave10 (bean InternalI2C)
**
**     Description :
**         This method selects a new slave for communication by its
**         10-bit slave address value. Any send or receive method
**         directs to or from selected device, until a new slave
**         device is selected by this method. This method is not
**         available for the SLAVE mode.
**     Parameters  :
**         NAME            - DESCRIPTION
**         Slv             - 10-bit slave address value.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_BUSY - The device is busy, wait
**                           until the current operation is finished.
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
**                           ERR_DISABLED -  The device is disabled
** ===================================================================
*/
byte I2C1_SelectSlave10(word Slv)
{
  if (getRegBits(I2C_STAT,I2C_STAT_MSTACT_MASK | I2C_STAT_TFE_MASK)) { /* Is the device
in the activ state? */
    return ERR_BUSY;                   /* If yes then error */
  }
  setReg16(I2C_TAR, ((word)Slv) | 0x1000); /* Set slave address */
  return ERR_OK;                       /* OK */
}
```

```
/*
** ===================================================================
**     Method      :  I2C1_SelectSpecialCommand (bean InternalI2C)
**
**     Description :
**         This method selects a special command. Any send or
**         receive method directs to or from selected device, until
**         a new slave device is selected by this method. This
**         method is not available for the SLAVE mode.
**     Parameters  :
**         NAME            - DESCRIPTION
**         Cmd             - Special command. Possible values:
**                           0 - General call address
**                           1 - Start byte
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_BUSY - The device is busy, wait
**                           until the current operation is finished.
**                           ERR_SPEED - This device does not work in
**                           the active speed mode
**                           ERR_DISABLED -  The device is disabled
** ===================================================================
*/
byte I2C1_SelectSpecialCommand(byte Cmd)
{
  if (Cmd > 1) {                       /* Is the value out of range? */
    return ERR_RANGE;                  /* If yes then error */
  }
  if (getRegBits(I2C_STAT,I2C_STAT_MSTACT_MASK | I2C_STAT_TFE_MASK)) { /* Is the device
in the activ state? */
    return ERR_BUSY;                   /* If yes then error */
  }
  if (Cmd == 0) {
    setReg16(I2C_TAR, 0x0800);         /* General call address */
  }
  else {
    setReg16(I2C_TAR, 0x0C00);         /* Start byte */
  }
  return ERR_OK;                       /* OK */
}

/*
** ===================================================================
**     Method      :  I2C1_GetSelected (bean InternalI2C)
**
**     Description :
**         This method returns 7-bit slave address value of the
**         slave, which is currently selected for communication with
**         the master. This method is not available for the SLAVE
**         mode.
**     Parameters  :
**         NAME            - DESCRIPTION
**       * Slv             - Current selected slave address value.
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
```

```
**                          ERR_SPEED - This device does not work in
**                          the active speed mode
** ===================================================================
*/
byte I2C1_GetSelected(byte *Slv)
{
  *Slv = (byte)(getReg16(I2C_TAR) & 0x7F); /* Get slave address */
  return ERR_OK;                         /* OK */
}

/*
** ===================================================================
**     Method      :  I2C1_GetMode (bean InternalI2C)
**
**     Description :
**         This method returns the actual operating mode of this
**         bean.
**     Parameters  : None
**     Returns     :
**         ---             - Actual operating mode value
**                           TRUE - Master
**                           FALSE - Slave
** ===================================================================
*/
/*
bool I2C1_GetMode(void)

**  This method is implemented as a macro. See I2C1.h file.  **
*/

/*
** ===================================================================
**     Method      :  I2C1_ConnectPin (bean InternalI2C)
**
**     Description :
**         This method reconnects requested pin associated with the
**         selected peripheral in this bean. This method is available
**         only for CPU derivatives and peripherals that support runtime
**         pin sharing with other internal on-chip peripherals.
**     Parameters  :
**         NAME            - DESCRIPTION
**         PinMask         - Mask for the requested pins.
**                           The peripheral pins are reconnected
**                           according to this mask.
**                           Possible parameters:
**                           I2C1_IN_PIN     - Input pin
**                           I2C1_OUT_PIN  - Output pin
**                           I2C1_CLK_PIN   - Clock pin
**                           I2C1_SS_PIN   - Slave select pin
**     Returns     : Nothing
** ===================================================================
*/
/*
void I2C1_ConnectPin(byte PinMask)

**  This method is implemented as a macro. See I2C1.h file.  **
*/
```

```
/*
** ===================================================================
**     Method      :  I2C1_GetTxAbortStatus (bean InternalI2C)
**
**     Description :
**         Returns and clears an accumulated set of Tx abort status
**         flags.
**     Parameters  :
**         NAME            - DESCRIPTION
**       * Status          - Pointer to returned set of status
**                           flags
**     Returns     :
**         ---             - Error code, possible codes:
**                           ERR_OK - OK
**                           ERR_SPEED - The device does not work in
**                           the active speed mode
** ===================================================================
*/
byte I2C1_GetTxAbortStatus(I2C1_TTxAbortStatus *Status)
{
  EnterCritical();                     /* Disable global interrupts */
  Status->status = TxAbortStatus;      /* Store Tx abort flags */
  TxAbortStatus = 0;                   /* Clear Tx abort flags */
  ExitCritical();                      /* Enable global interrupts */
  return ERR_OK;                       /* OK */
}

/*
** ===================================================================
**     Method      :  I2C1_Init (bean InternalI2C)
**
**     Description :
**         Initializes the associated peripheral(s) and the beans
**         internal variables. The method is called automatically as a
**         part of the application initialization code.
**         This method is internal. It is used by Processor Expert only.
** ===================================================================
*/
void I2C1_Init(void)
{
  /* I2C_ENBL:
??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,EN=0 */
  setReg16(I2C_ENBL, 0x00);            /* Disable device */
  TxAbortStatus = 0;                   /* Reset transmit abort flags */
  I2C1_SerFlag = 0x80;                 /* Reset all flags */
  /* I2C_FSHCNT: FSHCNT=0x1E */
  setReg16(I2C_FSHCNT, 0x1E);          /* Fast speed high count */
  /* I2C_FSLCNT: FSLCNT=0x29 */
  setReg16(I2C_FSLCNT, 0x29);          /* Fast speed low count */
  /* I2C_CTRL:
??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,SLVDIS=1,RSTEN=1,ADDRMST=0,ADDRSLV=0,SPD=2
,MSTEN=1 */
  setReg16(I2C_CTRL, 0x65);            /* Control */
  /* I2C_RXFT:
??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,RXTL=0 */
  setReg16(I2C_RXFT, 0x00);            /* Receive FIFO threshold level */
```

```
  /* I2C_TXFT:
??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,TXTL=0 */
  setReg16(I2C_TXFT, 0x00);             /* Transmit FIFO threshold level */
  /* I2C_IENBL:
??=0,??=0,??=0,??=0,GC=0,STDET=1,STPDET=1,ACT=0,TXDONE=0,TXABRT=1,RDREQ=0,TXEMPTY=0,TXO
VR=0,RXFULL=1,RXOVR=0,RXUND=0 */
  setReg16(I2C_IENBL, 0x0644);          /* Interrupt mask */
  /* I2C_TAR: ??=0,??=0,??=0,ADDRMST=0,SPCL=0,GCSTRT=0,TA=0x50 */
  setReg16(I2C_TAR, 0x50);              /* Target address */
  getReg(I2C_CLRINT);                   /* Clear flags */
  /* I2C_ENBL:
??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,??=0,EN=1 */
  setReg16(I2C_ENBL, 0x01);             /* Enable device */
}

/*
** ===================================================================
**     Method      :  I2C1_CheckBus (bean InternalI2C)
**
**     Description :
**         This method returns the status of the bus. If the START
**         condition has been detected, the method returns
**         'BeanName'_BUSY. If the STOP condition has been detected,
**         the method returns 'BeanName'_IDLE.
**     Parameters  : None
**     Returns     :
**         ---             - Status of the bus.
** ===================================================================
*/
/*
byte I2C1_CheckBus(void)

**  This method is implemented as a macro. See I2C1.h file.  **
*/


/* END I2C1. */

/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 2.98.02 [03.79]
**     for the Freescale 56800 series of microcontrollers.
**
** ###################################################################
*/
```

4 Comparing Applications — MC56F801x vs. MC56F802x3x

4.1 Main Program

The main programs are functionally equivalent, with only minor differences in the statistics maintained by the events code. At this level function calls look the same. This makes moving to the I2C on the MC56F802x3x quite simple.

4.2 Events.c

There is more information available on the MC56F802x3x about how statistics are collected. However, it is not necessary to use these because the application running on the MC56F8013 does not use the new events:

- I2C1_OnStart
- I2C1_OnStop

It is worthwhile to examine the handling of NAK. Both applications use the event I2C1_OnNACK. However, in the MC56F8037 code, the event counts the NACK but does not count an error. There is another event unique to the MC56F802x3x implementation, I2C1_OnTxAbort, that counts the errors.

4.3 I2C1.c

Because this code is written by Processor Expert, it is not required to fully address it. However, it can be seen that the interrupt structure differs, as do the IO registers. For the 56F802x3x, for example, there is an interrupt handler for receive (RX) and one for transmit (TX), whereas for the MC56F801x there is only one interrupt handler. For a porting strategy, we can directly use the function calls we already used, leaving the driver details to the code generator.

4.4 Conclusion

Porting the application to the new I2C peripheral is simplified with the provision of driver software from the Processor Expert bean for the I2C peripheral. Because many applications are similar to this one, due to memory-mapped I/O, this note should serve to assist those faced with porting an I2C application from the MC56F801x series to the MC56F802x3x series.

The complete projects are available for both devices along with this application note.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3608
Rev. 0
04/2009

freescale™
semiconductor