LAUTERBACH
DEVELOPMENT TOOLS

General Commands Reference Guide D

Release 09.2023

MANUAL

General Commands Reference Guide D

General Commands Reference Guide D

Version 09-Oct-2023

History

21-Feb-2023 Removed /**TURBOPACK** option for Data.LOAD.ROF.

21-Feb-2023 Removed /**FASTPACK** option for Data.LOAD.AIF, Data.LOAD.COFF, Data.LOAD.DBX, Data.LOAD.Elf, Data.LOAD.Icoff, Data.LOAD.Ieee and Data.LOAD.SDS.

21-Nov-2022 New options for Data.LOAD.Elf:
/**PREFIX**, /**RELOCSTRIPPED**, /**NOLINKAGENAME**, /**FILTERBYCORE**.

21-Nov-2022 Removed /**PACK** option for Data.LOAD.COFF, Data.LOAD.DBX and Data.LOAD.Ieee.

01-Nov-2022 New options for Data.LOAD.Elf:
/**CODEPROG**, /**DWOFILES**, /**DWPFILE**, /**NOARGCOERCE**, /**NODEBUG**, /**NODEBUGFRAME**, /**NOEHFRAME**, /**NODOUBLE** and /**IgnoreModuleRange**.

22-Aug-2022 New option /**IgnoreFUNCLines** for Data.LOAD.Elf command.

05-Jul-2022 New option /**Append** for Data.SAVE.IntelHex command.

24-May-2021 Description of the DTM<trace> command group.

Data

Memory access

Overview Data

Memory Access by the TRACE32 Debugger

TRACE32 debuggers operate on the memory of the target system.

Access Procedures

The following examples show typical memory access commands:

```
Data.dump 0x1000                        ; display a hex dump starting at
                                        ; address 0x1000

Data.Set 0x1000 %Byte 0x55              ; write 0x55 as a byte to the
                                        ; address 0x1000

Data.LOAD.Elf demo.elf                  ; load program from file to
                                        ; the target memory
```

An **access class** can be used to specify memory access details.

Examples:

```
Data.dump A:0x1000                      ; display a hex dump starting at
                                        ; address 0x1000, physical access

Data.dump NC:0x1234                     ; display a hex dump starting at
                                        ; address 0x1234, non-cached access

Data.dump L2:0x1234                     ; display a hex dump starting at
                                        ; address 0x1234, L2 cache access
```

Keywords for <format>

You can display memory in TRACE32 using the following formats. Please note that not all format are supported by all **Data.*<sub_cmd>*** commands. Please refer to the documentation of the single commands for more information.

[*<format>*]: **Decimal** [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
DecimalU [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Hex [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
HexS [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
OCTal [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Ascii [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Binary [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Float. [**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeMFFP** | …]
sYmbol [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Var
DUMP [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Byte [.*<endianness>* [.*<bitorder>*]]
Word [.*<endianness>* [.*<bitorder>*]]
Long [.*<endianness>* [.*<bitorder>*]]
Quad [.*<endianness>* [.*<bitorder>*]]
TByte [.*<endianness>* [.*<bitorder>*]]
PByte [.*<endianness>* [.*<bitorder>*]]
HByte [.*<endianness>* [.*<bitorder>*]]
SByte [.*<endianness>* [.*<bitorder>*]]

Decimal	Display the data as decimal number.
DecimalU	Display the data as unsigned decimal number.
Hex	Display the data as hexadecimal number.
HexS	Display the data as signed hexadecimal number.
OCTal	Display the data as octal number.
Ascii	Display the data in an ASCII representation.
Binary	Display the data as binary number.
Float	Display the data in a floating point representation.
sYmbol	Display the data as hexadecimal number. The column "symbol" in the window will show the symbol corresponding to the address to which the data *points* (while without **sYmbol** the column symbol shows the symbol corresponding to the address *containing* the data).
Var	Display HLL variables at their memory location (similar to **Var.Watch**).

DUMP Hexadecimal dump.

Byte, Word, Long,... See **"Keywords for <width>"**, page 11

Keywords for <width>

In TRACE32, you can access and display memory and register contents by using the following keywords for *<width>*:

[*<width>*]: **Byte**
Word
TByte
Long
PByte
HByte
SByte
Quad

Byte 8-bit

Word 16-bit

TByte 24-bit (tribyte)

Long 32-bit (long word)

PByte 40-bit (pentabyte)

HByte 48-bit (hexabyte)

SByte 56-bit (septuabyte)

Quad 64-bit (quad word)

The following table lists frequently-used **Data.*()** functions. For a complete list, see **"Data Functions"** in General Function Reference, page 135 (general_func.pdf).

Data.Byte(*<address>***)**	Returns memory content of a byte.
Data.Word(*<address>***)**	Returns memory content of a word (16-bit).
Data.Long(*<address>***)**	Reads memory at specified address. Address must be used with access class.
Data.String(*<address>***)**	Reads zero-terminated string from memory, the result is a string.
Data.SUM()	Gets the checksum of the last executed **Data.SUM** command.

Examples:

```
Register.Set PC Data.Long(SD:4)     ; set PC to start value
```

```
PRINT Data.Byte(SD:flag+4)          ; display single byte
```

```
&memstring=Data.String(string1)     ; copy and print string
PRINT "&memstring"
Data.Set string2 "&memstring"
```

```
Data.SUM 0x0--0x0fffe /Byte         ; fill last byte to build zero
Data.Set 0x0ffff Data.SUM()         ; checksum in 64K block
```

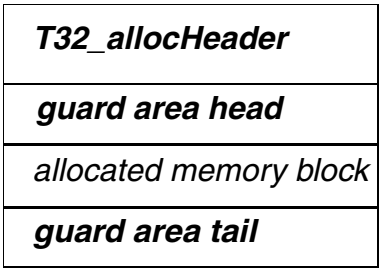
Format: **Data.AllocList** [*<address>*] [**/***<option>* …]

<option>:
TIme
Address
Caller
Size
SumCaller
SumSize
Track
Guard *<size>*

The basic idea of the static memory allocation analysis is the following:

- The user program manages a double linked list that contains all information about the allocated memory blocks.
- The TRACE32 software offers the command **Data.AllocList** to analyze this information.

Each element of the double linked list has the following structure:



Each allocated memory block is surrounded by 2 so-called guard areas. The default size of each guard area is 16 bytes. The option **/Guard** *<size>* allows to use a different size for the guard areas.

Each guard area has to be filled with a fixed pattern when the memory block is allocated.

```
static void SetGuard(unsigned char * guard)
{
     int i;
     for (i = 0; i < T32_GUARD_SIZE; i++)
          guard[i] = (unsigned char) (i + 1);
}
```

- The user program can check if there were write accesses beyond the upper or lower bound of the allocated memory block when the memory block is freed and stop the program execution in such a case.
- The TRACE32 software can check all blocks for writes beyond the upper or lower bound when the **Data.AllocList** window is displayed.

The *T32_allocHeader* contains information to maintain the double linked list, information about the caller who requested the memory block and information about the originally requested memory size.

```
typedef struct T32_allocHeader
{
     struct T32_allocHeader * prev;
     struct T32_allocHeader * next;
     void * caller;
     size_t size;
#if T32_GUARD_SIZE
     unsigned char guard[T32_GUARD_SIZE];
#endif
}
T32_allocHeader;
```

In order to maintain the double linked list that is required by the TRACE32 software to analyze the static memory allocation all ***malloc(size), realloc(ptr,size), free(ptr)*** calls in the user program have to be replaced by an extended version.

This can be done in two ways:

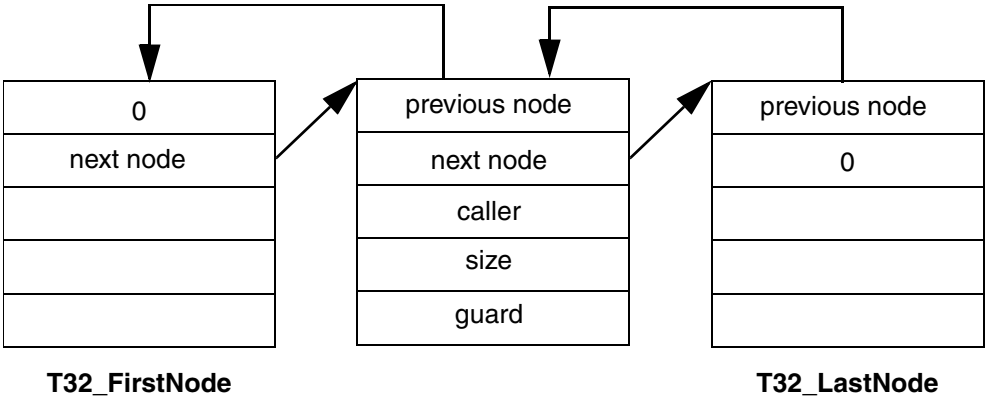
1. Within the source files.

```
#ifdef PATCHING_REQUIRED
#define malloc(size)T32_malloc(size)
#define realloc(ptr,size)T32_realloc(ptr,size)
#define free(ptr)T32_free(ptr)
extern void * T32_malloc();
extern void * T32_realloc();
extern void T32_free();
#endif
```

2. By using the **Data.ReRoute** command for a binary patch.

```
Data.ReRoute sYmbol.SECRANGE(.text) malloc T32_malloc \t32mem
Data.ReRoute sYmbol.SECRANGE(.text) realloc T32_realloc \t32mem
Data.ReRoute sYmbol.SECRANGE(.text) free T32_free \t32mem
```

1. A memory block is allocated. This memory block has the following size:
   size of the requested memory block + sizeof(T32_allocHeader) + T32_GUARD_SIZE
2. The caller of the T32_malloc function is stored in the structure of the type T32_allocHeader.
3. The size of the requested memory is stored in the structure of the type T32_allocHeader.
4. Both guard areas are initialized with fixed values, so that the TRACE32 software can later check if there are any write accesses beyond the block bounds (ERROR HEAD, ERROR TAIL).
5. The information about the allocated memory block is entered into the double linked list.



The TRACE32 software assumes that **T32_FirstNode** is the default symbol for the first element of the list. If another symbol is used this information has to be provided when the command **Data.AllocList** is used.

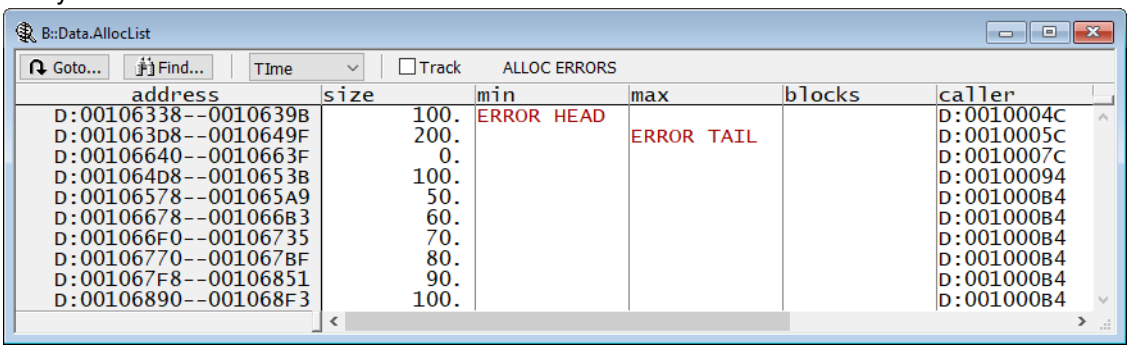
```
Data.AllocList List_M2          ; List_M2 is the start of the linked
                                ; list for the command Data.AllocList
```

What does T32_free(ptr) do?

1. Both guard areas are checked to detect any write access beyond the block bounds. If such a write access happened a error handling function is called.

2. The information about the allocated memory block is removed from the double linked list.

A complete example for the implementation of the linked list and for the use of the command **Data.AllocList** can be found in ~~/demo/powerpc/etc/malloc. The example can be used with the simulator for the PowerPC family.



Description of the Data.AllocList Options

Track	The allocated memory blocks are displayed in the order of their entry.
Address	The allocated memory blocks are sorted by address.
Caller	The allocated memory blocks are sorted by the caller.
Size	The allocated memory blocks are sorted by their size.
SumCaller	The allocated memory blocks are sorted by the caller and for each caller the sum of all allocated blocks is displayed.
SumSize	The allocated memory blocks are sorted by the caller. The caller who allocated most memory blocks is on top of the list.
TIme (default)	Tracks the window to the reference position of other windows.
Guard	Defines the size of the guard areas. Default is 16.

Examples

```
Data.AllocList /Size              ; display a static memory allocation
                                  ; sorted by size

Data.AllocList /Guard 20.         ; the user program uses a guard area
                                  ; of 20 bytes

Data.AllocList List_M2            ; List_M2 is the start of the linked
                                  ; list for the command Data.AllocList
```

See also

❑ Data.AL.ERRORS()

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.Assemble** [*<address>*] *<mnemonic>* [{*<mnemonic>*}]

Writes an opcode specified by an assembler instruction *<mnemonic>* to the memory at *<address>*.

Entering a specific opcode is facilitated by softkeys indicating the available options (e.g. offsets, registers, …) according to the current CPU architecture.

Multiple mnemonics can be specified with a single **Data.Assemble** command. For improving readability in scripts you may use a line continuation character

```
Data.Assemble T:0x20 push lr  pop pc       ; two commands in one line

Data.Assemble R:0x0 blx 0x21 \
                    add r6 , r6 , #1\
                    b 0x0
```

To quickly modify a code line, use the commands **Modify here** or **Assemble here** from the popup menu in the **List** window. The commands **Data.PROGRAM** and **Data.ReProgram** can be used to enter multiple instructions or small programs.

```
; fill memory-range from 0x0 up to 0xffff with NOP:
Data.Assemble 0x0--0x0ffff nop

; insert and assemble am move- command at address 0 next command to next
; address:

Data.Assemble 0 move.b d0,d1
Data.Assemble , move.b d3,d4
```

NOTE: Note the syntax for expressing

- *absolute addresses* using the plain constant and
- *PC-relative offsets* using the format `${+|-}offset`).
  (Always one of the +/- sign specifiers must be present!)

Examples:

```
J 0xFC000000    ; Jump to absolute address 0xFC000000
JLA $-0x10      ; Jump And Link to PC-0x10
JLA $+100       ; Jump And Link to PC+0x100
```

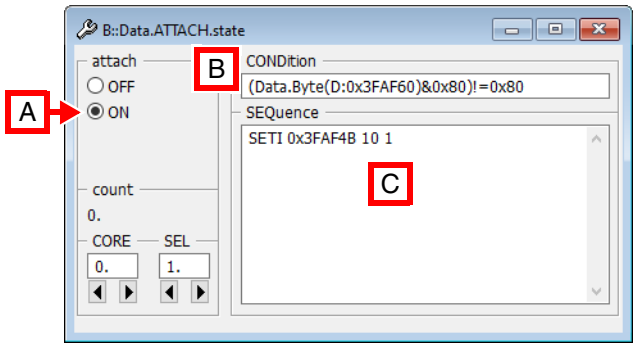
NOTE:	If there are multiple ISAs for a CPU family (e.g. ArmV4, Arm4T, ARMv7, for the family of ARM cores), the **Data.Assemble** command might not check whether the *<mnemonic>* is supported by the CPU currently chosen by **SYStem.CPU** and the opcode is written regardless.

See also

- Data.dump
- Data.PROGRAM

The **Data.ATTACH** command group allows to define a sequence of **Data.Set** commands that are executed when the debugger is activated with **SYStem.Mode Attach**.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **Data.ATTACH.state** window:



A For descriptions of the commands in the **Data.ATTACH.state** window, please refer to the **Data.ATTACH.*** commands in this chapter.
Example: For information about **ON**, see **Data.ATTACH.ON**.

B Conditions can be set up in the **CONDition** field using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** field using the *<data_set_commands>* **SET**, **SETI**, **GETS**, and **SETS**.

See also

■ Data.EPILOG ■ Data.PROLOG ■ Data.STARTUP ■ Data.TIMER

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.ATTACH.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **Data.ATTACH.SEQuence** will be executed periodically.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

Examples:

```
; reads the long at address D:0x3faf30, proceeds a binary AND with
; a constant (here 0xffffffff). If the result is equal to 0x80000000 the
; condition is true and the defined sequence is executed.
Data.ATTACH.CONDition (Data.Long(D:0x3faf30)&0xffffffff)==0x80000000
```

```
; read the word at address D:0x3xfaf30
Data.ATTACH.CONDition (Data.Word(D:0x3faf30)&0xff00)!=0x8000
```

```
; reads the byte at address D:0x3xfaf30
Data.ATTACH.CONDition (Data.Byte(D:0x3faf30)&0xf0)!=0x80
```

Data.ATTACH.CORE Select core for attach sequence

Format:	**Data.ATTACH.CORE** *<core_number>*

Selects the core for which you want to define one or more data attach sequences.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: This script shows how to define a data attach sequence that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a data attach sequence
Data.ATTACH.CORE 3.

;Define the data attach sequence for core 3
Data.ATTACH.CONDition <your_code>
Data.ATTACH.SEQuence  <your_code>
```

For information on how to configure two different attach sequences, see **Data.ATTACH.SELect**.

Data.ATTACH.OFF — Switch attach sequence off

Format: **Data.ATTACH.OFF**

Switches the **Data.ATTACH** feature off.

Data.ATTACH.ON — Switch attach sequence on

Format: **Data.ATTACH.ON**

Switches the **Data.ATTACH** feature on.

Data.ATTACH.RESet — Reset attach data sequence

Format: **Data.ATTACH.RESet**

Switches the **Data.ATTACH** feature off and clears all settings.

Format: **Data.ATTACH.SELect** *<serial_number>*

Increments the index number for each new data attach sequence. This is useful, for example, if you need two separate attach sequences with each sequence having its own **Data.ATTACH.CONDition**.

TRACE32 automatically assigns the index number 1. to the first **Data.ATTACH.SEQuence**. If you require a second, separate attach sequence, then increment the *<index_number>* to 2. Otherwise the second attach sequence will overwrite the first one. You can define a maximum of 10 attach sequences.

Example 1: Two attach sequences with the *same* **Data.ATTACH.CONDition** may have the *same* index number. The backslash **** is used as a line continuation character. No white space permitted after the backslash.

```
;Set the index number to 1.
Data.ATTACH.SELect 1.

;Attach sequences shall be executed only if this condition is true:
Data.ATTACH.CONDition (Data.Word(D:0x4faf34)&0xff00)==0x4000

;Define the two attach sequences:
Data.ATTACH.SEQuence SET 0x4faf54 %Word 0xC0C0 \
                     SET 0x4faf64 %Word 0xD0D0
```

Example 2: Two attach sequences with *different* **Data.ATTACH.CONDition** settings require two *different* index numbers.

```
;1st attach sequence - TRACE32 automatically sets the index number to 1.
Data.ATTACH.SELect 1.

;If this data attach condition is true, ...
Data.ATTACH.CONDition (Data.Word(D:0x4faf38)&0xff00)==0x2000

;... then the 1st attach sequence will be executed
Data.ATTACH.SEQuence SET 0x4faf58 %Word 0xE0E0

;Increment the index number to define the 2nd attach sequence
Data.ATTACH.SELect 2.

;If this data attach condition is true, ...
Data.ATTACH.CONDition (Data.Word(D:0x4faf38)&0xff00)==0x3000

;... then the 2nd attach sequence will be executed
Data.ATTACH.SEQuence SET 0x4faf58 %Word 0xF0F0
```

Format: **Data.ATTACH.SEQuence** *<command>* …

<command>: **SET** *<address>* **%***<format>* *<data>*
SETI *<address>* **%***<format>* *<data>* *<increment>*
SETS *<address>*
GETS *<address>*

Defines a sequence of **Data.Set** commands that are executed when the emulation system in activated.

SET Write *<data>* to *<address>*.

SETI Write *<data>* to *<address>*.
Then *<data>* is incremented by *<increment>*.

GETS Save the data at *<address>*.

SETS Write the data that was saved with a previous GETS back to *<address>*.

Examples:

```
Data.ATTACH.SEQuence SET 0x3faf50 %Word 0xa0a0

Data.ATTACH.SEQuence SETI 0x3faf50 %Word 0xa0a0 2

Data.ATTACH.SEQuence SETS 0x3faf60

Data.ATTACH.SEQuence GETS 0x3faf60
```

Data.ATTACH.state Attach data state display

Format: **Data.ATTACH.state**

Displays the **Data.ATTACH** state window.

Format: **Data.BDTAB** *<address> <size>*

The command **Data.BDTAB** is implemented for most PowerPC processors.

<address>	Defines the start address of the buffer descriptor.
<size>	Defines the size of each entry in the buffer descriptor table. Possible is a size of 8 or 16 byte.

Example:

```
Data.BDTAB iobase()+Data.Word(D:iobase()+0x8400) 8
```

It is recommended to use a mouse click in the peripheral window to display the buffer descriptor table.

See also

- Data.CHAIN

[Examples]

Format:	**Data.BENCHMARK** *<range>* [*<range>* [*<range>*]] [*<size>* …]
<range>:	*<program_range>* *<data_stack_range>* *<data_test_range>*
<size>:	*<ic_size>* *<dc_size>* *<l2_size>* *<l3_size>*

Basic Concept

The basic idea of the **Data.BENCHMARK** command is the following:

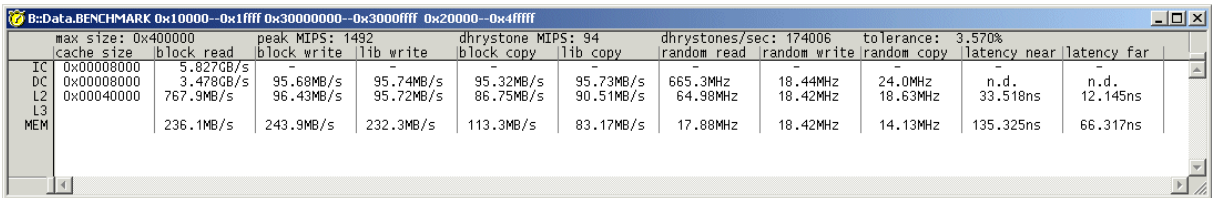
- Load a benchmark program that performs various memory read, memory write and memory copy operations to the target.
- Enable all caches.
- The command **Data.BENCHMARK** starts the benchmark program and measures the bandwidth of all caches and memories with the help of the **RunTime** counters.

The Benchmark Program

Precompiled benchmark program can be found in ~~/demo/*<cpu>*/etc/benchmark, e.g. ~~/demo/arm/etc/benchmark

In the same directory you can also find the C source for the benchmark program. It is recommended to compile the benchmark program with your compiler if you want to test the functions (block write, copy etc.) provided by your compiler. Before you compile the benchmark program with your compiler please read the comments in the C source.

The following window displays the result of the **Data.BENCHMARK** command:



B::Data.BENCHMARK 0x10000--0x1ffff 0x30000000--0x3000ffff 0x20000--0x4fffff

max size: 0x400000 peak MIPS: 1492 dhrystone MIPS: 94 dhrystones/sec: 174006 tolerance: 3.570%

	cache size	block read	block write	lib write	block copy	lib copy	random read	random write	random copy	latency near	latency far
IC	0x00008000	5.827GB/s	-	-	-	-	-	-	-	-	-
DC	0x00008000	3.478GB/s	95.68MB/s	95.74MB/s	95.32MB/s	95.73MB/s	665.3MHz	18.44MHz	24.0MHz	n.d.	n.d.
L2	0x00040000	767.9MB/s	96.43MB/s	95.72MB/s	86.75MB/s	90.51MB/s	64.98MHz	18.42MHz	18.63MHz	33.518ns	12.145ns
L3											
MEM		236.1MB/s	243.9MB/s	232.3MB/s	113.3MB/s	83.17MB/s	17.88MHz	18.42MHz	14.13MHz	135.325ns	66.317ns

Column / Header	Description
block copy	Bandwidth for block copy algorithm of the benchmark program
block read	Bandwidth for block read
block write	Bandwidth for block write
cache size	Cache size of the different caches If off-chip caches are used, their sizes has to be defined as parameter when using the **Data.BENCHMARK** command.
latency far	laTency until data is available for far addresses
latency near	lateNcy until data is available for near addresses
lib copy	Bandwidth for block copy function provided by the compiler library
lib write	Bandwidth for block write function provided by the compiler library
max size	Maximum block size used during the test
random copy	Frequency of random copy operations
random read	Frequency of random read operations
random write	Frequency of random write operations
tolerance	Tolerance of the **RunTime** counters. Depending of the implementation the **RunTime** counters the result of the time measurement deviates slightly.

Examples

Example for the PowerQuicc III:

```
; select the CPU
SYStem.CPU MPC85xx

SYStem.Option.FREEZE OFF

; initialize your target hardware
…

; load the benchmark program to address 0x1000
Data.LOAD.Elf benchmark.x 0x1000

; enable L1 cache
Data.Set SPR:0x3F2 %Long 3
Data.Set SPR:0x3F3 %Long 3

; enable L2 cache
Data.Set A:0xFDF20000 %Long 0XD4000000

; execute the Data.BENCHMARK program

; the <program_range> is 0x10000--0x1ffff

; the <data_stack_range> is 0x30000000--0x3000ffff
; the test data and the stack for the benchmark program are located from
; 0x30000000--0x3000ffff

; the <data_test_range> is 0x20000--0x4fffff
; the memory range that is tested by the benchmark program is
; 0x20000--0x4fffff

; Data.BENCHMARK <program_range> <data_stack_range> <data_test_range>
; <ic_size> <dc_size> <l2_size> <l3_size>
Data.BENCHMARK 0x10000--0x1ffff 0x30000000--0x3000ffff 0x20000--0x4fffff
```

Further examples:

```
; load the benchmark program to address 0x20000
Data.LOAD.Elf benchmark.x 0x20000

; the address range 0x20000--0x4fffff is used for the program,
; data/stack and is also the test address range

; Data.BENCHMARK <program_range> <data_stack_range> <data_test_range>
; <ic_size> <dc_size> <l2_size> <l3_size>

Data.BENCHMARK 0x20000--0x4fffff
```

```
; load the benchmark program to address 0x20000
Data.LOAD.Elf benchmark.x 0x20000

; the address range 0x20000--0x4fffff is used for the program
; data/stack and is also the test address range

; the size of the L2 cache is 128K

; parameters that are skipped are represented by a comma

; Data.BENCHMARK <program_range> <data_stack_range> <data_test_range>
; <ic_size> <dc_size> <l2_size> <l3_size>

Data.BENCHMARK 0x20000--0x4fffff ,,,, 0x1000
```

Format: **Data.CHAIN** *<base> <link_offset> <elements>* [*/<option>* …]

<elements>: [[**%***<format>*] [*<address>* | *<range>*] …]

<format>:
Decimal [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
DecimalU [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Hex [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
HexS [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
OCTal [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Ascii [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Binary [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Float[**.***<float_rep>*[**.***<endianness>*]]
sYmbol [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Var
DUMP [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Byte [**.***<endianness>* [**.***<bitorder>*]]
Word [**.***<endianness>* [**.***<bitorder>*]]
Long [**.***<endianness>* [**.***<bitorder>*]]
Quad [**.***<endianness>* [**.***<bitorder>*]]
TByte [**.***<endianness>* [**.***<bitorder>*]]
PByte [**.***<endianness>* [**.***<bitorder>*]]
HByte [**.***<endianness>* [**.***<bitorder>*]]
SByte [**.***<endianness>* [**.***<bitorder>*]]

<width>: **DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

<float_rep>: **Ieee** | **IeeeRev** | **IeeeS** | **IeeeDbl** | **...**

<endianness>: **DEFault** | **LE** | **BE**

<bitorder>: **DEFault** | **BitSwap**

<option>:
CORE *<core_number>*
COVerage
CTS
Track
FLAG *<flag>*
CFlag *<cflag>*
Mark *<break>*

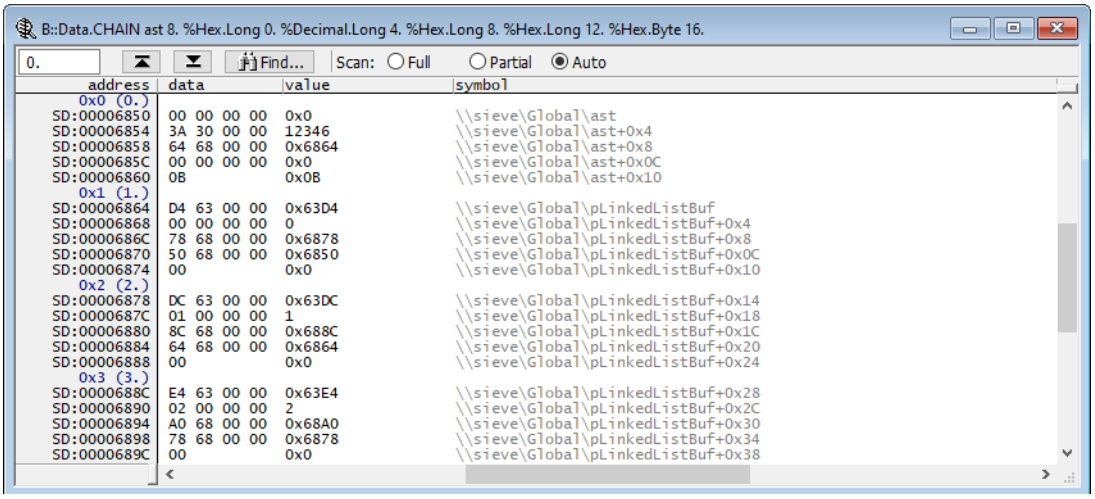
<flag>:	**Read** | **Write** | **NoRead** | **NoWrite**
<cflag>:	**OK** | **NoOK** | **NOTEXEC** | **EXEC**
<break>:	**Program** | **Hll** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

Displays a linked list without high-level information. The link to the next element is taken from the current element address plus *link offset*. The size of the pointer is one, two or four bytes, depending on the CPU type and address space.

Decimal, DecimalU,...	Refer to **"Keywords for <format>"**, page 10
Byte, Word, ...	Refer to **"Keywords for <width>"**, page 11
DEFault, BE, LE	Define byte-order display direction: default target endianness, Big Endian or Litte Endian
DEFault, BitSwap	**BitSwap** allows to display data in reverse bit-order in each byte. If **BitSwap** is used together with BE or LE, the byte order will not change, otherwise **BitSwap** will also reverse the byte-order.
CORE *<number>*	Display memory from the perspective of the specified core /SMP debugging only).
COVerage	Highlight data memory locations that have never been read/written.
Track	Track the window to the reference address of other windows.
Mark *<break>*	Highlight memory locations for which the specified breakpoint is set.
CTS	Display **CTS** access information when CTS mode is active.

Example:

```
; Display a linked list, the first element is the symbol ast. The
; pointer to the next element is found at offset 8. from the base
; address. The element consists of a pointer, a counter, 2 pointers
; and a byte

Data.CHAIN ast 8. %Hex.Long 0. %Decimal.Long 4. %Hex.Long 8. %Hex.Long
12. %Hex.Byte 16.
```



By double-clicking a data word, a **Data.Set** command can be executed on the current address.

The **Data.CHAIN** window supports three different scan modes:

Full	The linked list is always completely updated.
Partial	The linked list is only partially updated. The update starts at the element that was on top of the window when the Partial button was selected the last time.
Auto (default)	The linked list is always completely updated. To balance the effect on the user interface, the list is updated for a specific time interval, then the update is stopped for a specific time interval to allow other activities on the user interface etc. The number of the last updated element is displayed beside the Auto button.

See also

- Data.CHAINFind
- Data.CSA
- Data.BDTAB
- Data.dump
- Data.TABle
- Data.View

Format:	**Data.CHAINFind** *<address> <value>* [*/<option>*]
<option>:	**Address** *<range>*

Searches for data in linked lists. Currently only searching for invalid address pointers is implemented. The search stops when the address of the element is inside the given address range.

The **Data.CHAINFind** command affects the following functions:

FOUND()	Returns TRUE if data was found.
TRACK.ADDRESS()	Returns the address of the last found data.

Example:

```
; A linked list is searched starting with the first element
; at symbol 'xlist'.
; The pointer to the next element is found at offset 12. from the
; base address of a element.
; Look for an address outside the allowed range 0x10000--0x1ffff.

Data.CHAINFind xlist 0x0c /Address !0x10000--0x1ffff
IF FOUND()
  Data.dump TRACK.ADDRESS()
```

See also

- Data.CHAIN

Format: **Data.CLEARVM** [*<address>* | *<addressrange>*]

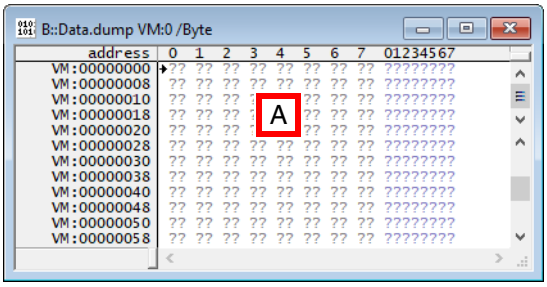
Clears the entire TRACE32 virtual memory (VM:) if *<address>* or *<range>* are not specified.

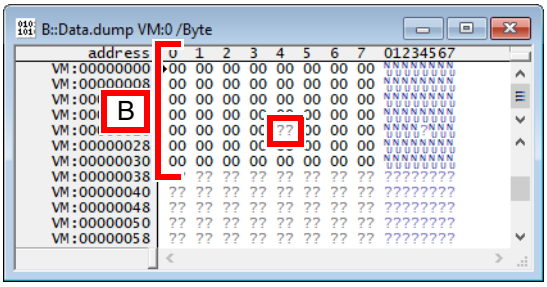
<address> Clears one byte at the specified address.

<addressrange> Clears the specified range.

Example:

```
;Open the Data.dump window displaying the virtual memory at address 0x0
Data.dump VM:0x0 /Byte

;clear the entire virtual memory (VM:), see [A]
Data.CLEARVM

;write 0x0 to the specified range in the virtual memory (VM:), see [B]
Data.Set VM:0x0++0x3F %Byte 0x0

;clear one byte at the specified address, see [B]
Data.CLEARVM VM:0x24
```





A Question marks (?????…) indicate uninitialized address locations.

B A 64-byte *<range>* was initialized with 0x0. The address VM:0x24 was uninitialized again.

Format:	**Data.ComPare** *<addressrange>* [*<address>*] [**/***<option>*]
<option>:	**Back** | **NoFind** | **ALL**

The contents of a memory area is compared byte-wise against the range starting with the second argument.

```
Data.ComPare 0x0--0x3fff 0x4000    ; compare two memory regions
```

```
; copy contents of specified address range to TRACE32 virtual memory
Data.Copy 0x3fa000++0xfff VM:

; display contents of TRACE32 virtual memory at specified address
Data.dump VM:0x3fa000

Go

Break

; compare contents of target memory with contents of TRACE32 virtual
; memory for specified address range
Data.ComPare 0x3fa000++0xfff VM:0x3fa000

; search for next difference
Data.ComPare

…
```

The **Data.ComPare** command affects the following functions:

FOUND()	Returns TRUE if a difference was found.
TRACK.ADDRESS()	Returns the address of the last found difference.

```
Data.ComPare 0x100--0xfff 0x3f

IF FOUND()
     Data.dump TRACK.ADDRESS()
```

```
Data.ComPare 0x100--0xfff 0x3f

IF FOUND()
     PRINT "Difference found at address " TRACK.ADDRESS()
```

See also

■ Data.COPY

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.COPY** *<addressrange>* [*<address>*] [**/***<option>*]
<option>:	**Verify** | **ComPare** | **DIFF** **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** **wordSWAP** | **LongSWAP** | **QuadSWAP** | **BitSwap** **SkipErrors** **PlusVM**

Data areas are copied. The address ranges may overlap.

Verify Verify data by following a read operation.

ComPare Data at address ranges is compared. Memory is not changed. The comparison stops with an error message after the first difference.

DIFF Data at address ranges is compared. Memory is not changed. The address of the compare can be checked using the **FOUND()** function.

Byte, **Word**, … See **"Keywords for <width>"**, page 11.

wordSWAP Swaps high and low bytes of a 16-bit word during copy.

LongSWAP Swaps high and low bytes of a 32-bit word during copy.

QuadSWAP Swaps high and low bytes of a 64-bit word during copy.

BitSwap Swaps the bits of each byte during copy.

SkipErrors Skips memory that cannot be read. Otherwise TRACE32 would abort the command if a bus error occurs while copying the specified *<range>*.

PlusVM The code is loaded into the target memory plus into the virtual memory.

Example 1

```
; copy within memory

Data.COPY 0x1000--0x1fff 0x3000       ; move 4 K block
```

Example 2

The **Data.COPY** *<addressrange>* **/DIFF** command is used together with the following functions:

FOUND()	Returns TRUE if a difference was found in the comparison.
TRACK.ADDRESS()	Returns the address of the first difference.

```
;Copy from VM: to SD:
Data.COPY VM:0x0--0x1F SD:0xB0

;Check if there are any differences between VM: and SD:
Data.COPY VM:0x0--0x1F SD:0xB0 /DIFF

IF FOUND()
     PRINT "Difference found at address " TRACK.ADDRESS()
```

Example 3

The **Data.COPY** *<addressrange>* **/LongSWAP** command is used to copy and swap a memory range and to convert it e.g. from Little- to Big-Endian or vice versa.

```
; set VM:0x0
Data.Set  VM:0x0 %Long %LE 0x11223344 0x55667788

; now copy the buffer to VM:0x20
Data.COPY VM:0x0++0x7 VM:0x20 /LongSWAP

; at VM:0x20 the memory content is now 0x44332211 0x88776655
```

See also

- Data.ComPare

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.CSA** *<csa_link>*

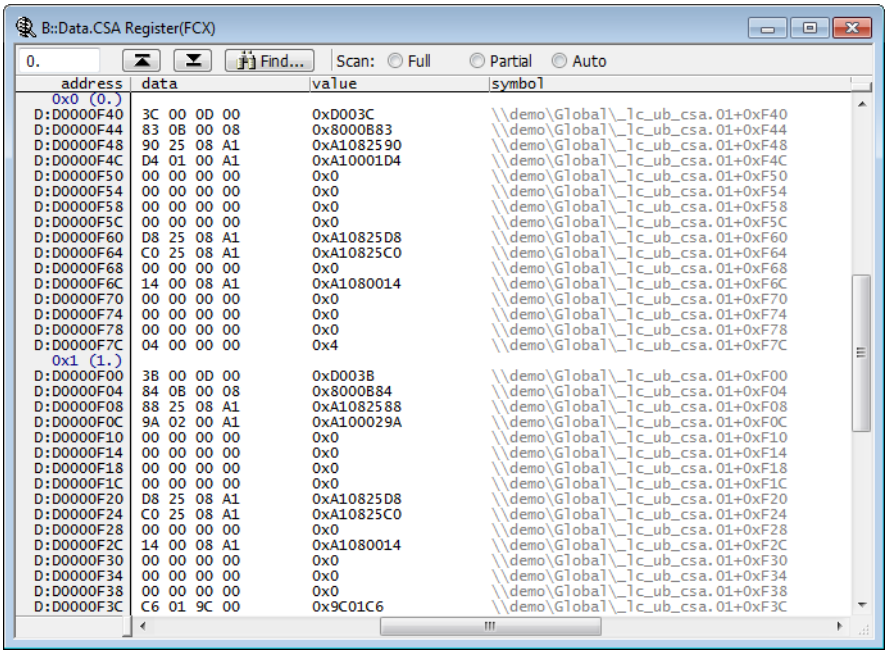
Displays a linked list of CSA entries. **Data.CSA** is a specialized variant of the **Data.CHAIN** command.

TriCore does not store the context information on the stack. Instead it saves them as a linked list in user-definable memory areas. The **Data.CSA** command displays the content of these lists in a user-friendly format. TRACE32 knows about the structure of the CSA lists and detects the end of the list. The user only has to specify the base link.

<csa_link> Link to the first CSA entry to display. The link needs to be encoded in the format TriCore uses internally. This allows a to pass the content of the corresponding register directly. See example below.

Example:

```
Data.CSA Register(FCX)
```



See also

- Data.CHAIN

Format: **Data.DRAW** [**%***<format>*] *<range1>* [[**%***<format>*] *<range2>* …] [*<scale>* [*<offset>*]] [**/***<options>*…]

<format>:
Decimal.[*<width>*[**.***<endianness>*]]
DecimalU.[*<width>*[**.***<endianness>*]]
Hex.[*<width>*[**.***<endianness>*]]
HexS.[*<width>*[**.***<endianness>*]]
OCTal.[*<width>*[**.***<endianness>*]]
Float[**.***<float_rep>*[**.***<endianness>*]]
Byte [**.***<endianness>*]
Word [**.***<endianness>*]
Long [**.***<endianness>*]
Quad [**.***<endianness>*]
TByte [**.***<endianness>*]
PByte [**.***<endianness>*]
HByte [**.***<endianness>*]
SByte [**.***<endianness>*]

<width>: **DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

<float_rep>: **Ieee** | **IeeeRev** | **IeeeS** | **IeeeDbl** | **...**

<endianness>: **DEFault** | **LE** | **BE**

<options>:
LOG | **Track**
Vector | **Points** | **Steps** | **Impulses**
Alternate *<number>* | **Element** *<number>*

The **Data.DRAW** command is used to visualize the contents of arrays. The array index is the x-axis, and the array content is the y-axis.

The command is useful, for example, for sampling signal quality in the mobile communications area or for sampling fuel injection in the automotive area. The **Var.DRAW** command has the same functionality, but takes the *<format>* information from the debug symbols.

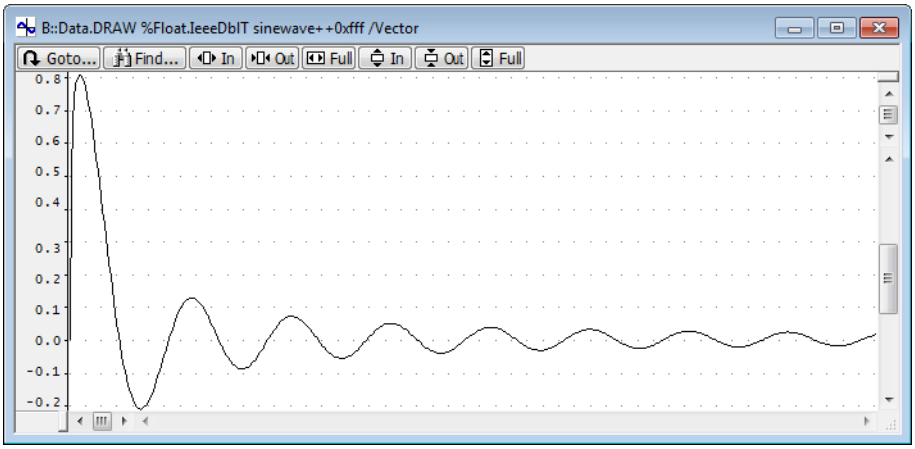
Decimal, DecimalU,...	Refer to **"Keywords for <format>"**, page 10
Byte, Word, ...	Refer to **"Keywords for <width>"**, page 11
DEFault, BE, LE	Define byte-order display direction: default target endianness, Big Endian or Litte Endian

<range1> ... *<range6>*	Array address range. Depending on the *<array_options>*, one or more lines are drawn per specified array.
<scale>	Units per pixel of y-axis (floating point). Default: **1.0**
<offset>	Offset of y-axis (floating point). Default: **0.0**

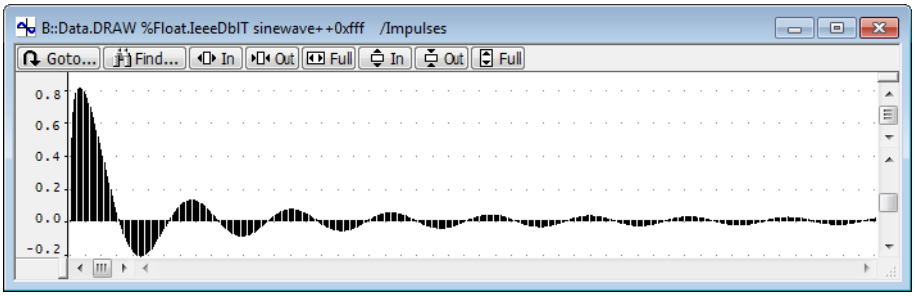
LOG	display y-axis in logarithmic scale
Track	Track the window to the reference address of other windows.
Vector (default)	draw each data value as a single pulse.
Points	display each data value as a dot.
Steps	connect the dots for the data values by steps.
Impulses	draw each data vales as a single pulse.
Alternate *<number>*	Split the array in *<number>* graphs. *<number>*=2 first graph display even elements second graph displays odd element. *<number>*=3 first graph displays element 0, n, 2n, … second graph displays 1, n+1, 2n+1, … third graph display 2, n+2, 2n+2, …
Element *<number>*	Specify the structure component to be displayed graphically.

```
SYStem.Up

; Load application
Data.LOAD.Elf ~~/demo/arm/compiler/arm/armle.axf

Register.Set PC main
Go
WAIT 2.s
Break

; See result 1.
;          <format>         <range1>            <scale> <offset> <option>
Data.DRAW %Float.IeeeDblT Var.RANGE(sinewave) 0.003    -0.2     /Vector

; See result 2.
Data.DRAW %Float.IeeeDblT sinewave++0xfff 0.003 -0.2 /Impulses

; See result 3.
;          <format>         <range1>         <range2>                 <disp>
Data.DRAW %Float.IeeeDblT sinewave++0xfff (sinewave+0x60)++0xfff
/Impulses

; See result 4.
;          <format>         <range1>         <arrayoptions>
Data.DRAW %Float.IeeeDblT sinewave++0xfff /Alternate 12 /Element 1 6 9
```

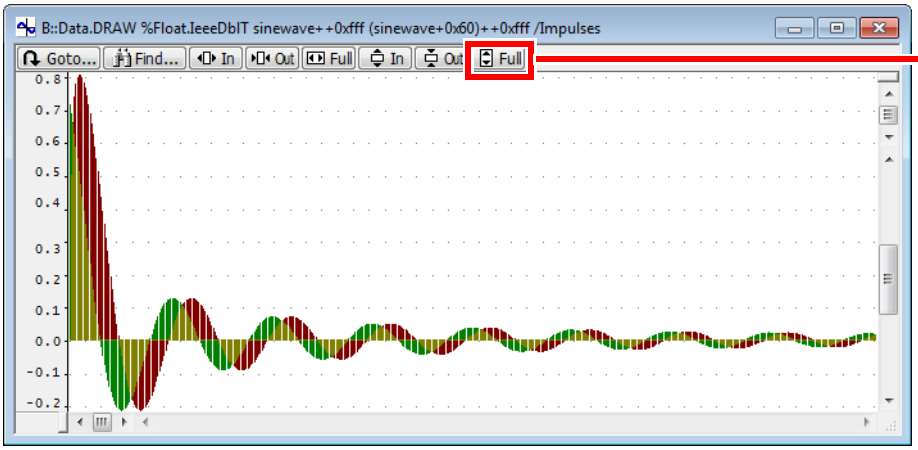
Result 1:



Result 2:



Result 3:



Click the **Full** button.

Result 4:

Click the **Full** button.

See also

- Data.DRAWFFT
- Data.DRAWXY
- Data.IMAGE
- Data.PROfile
- <trace>.DRAW
- Var.DRAW

▲ 'Release Information' in 'Legacy Release History'

[Example]

Format:	**Data.DRAWFFT** [**%***<format>*] *<range> <scale> <fftsize>* [**/***<option>*]
<format>:	**Decimal.**[*<width>*[**.***<endianness>*]] **DecimalU.**[*<width>*[**.***<endianness>*]] **Hex.**[*<width>*[**.***<endianness>*]] **HexS.**[*<width>*[**.***<endianness>*]] **OCTal.**[*<width>*[**.***<endianness>*]] **Float**[**.***<float_rep>*[**.***<endianness>*]] **Byte** [**.***<endianness>*] **Word** [**.***<endianness>*] **Long** [**.***<endianness>*] **Quad** [**.***<endianness>*] **TByte** [**.***<endianness>*] **PByte** [**.***<endianness>*] **HByte** [**.***<endianness>*] **SByte** [**.***<endianness>*]
<width>:	**DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<float_rep>:	**Ieee** | **IeeeRev** | **IeeeS** | **IeeeDbl** | **...**
<endianness>:	**DEFault** | **LE** | **BE**
<option>:	**BLACKMAN** | **HAMMING** | **HANN** **REAL** | **COMPLEX** **Vector** | **Points** | **Steps** | **Impulses**

Computes a fast Fourier transform (FFT) of the input data located in the specified memory range and graphically displays the spectrum.

This command can be used to visualize the frequencies in a signal; for example, the frequencies of audio and video input data. However, to illustrate and explain the command in this manual, a very simple example data set is used.

Decimal, DecimalU,...	Refer to **"Keywords for <format>"**, page 10
Byte, Word, ...	Refer to **"Keywords for <width>"**, page 11
DEFault, BE, LE	Define byte-order display direction: default target endianness, Big Endian or Litte Endian

<scale>	Scale factor (normalization) for the x-axis as floating-point value. The spectrum will range from 0 to <scale>/2. Example: **44100.0**
<fftsize>	Number of points, must be power of 2, e.g. **128.** | **256.** | **512.**

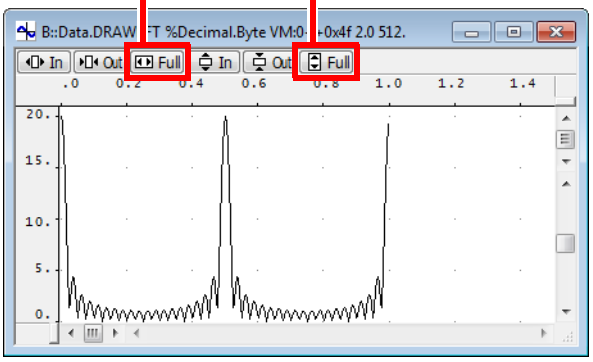
Real	array consists of real numbers only.
COMPLEX	array consists of complex number pairs.
Vector (default)	draw each data value as a single pulse.
Points	display each data value as a dot.
Steps	connect the dots for the data values by steps.
Impulses	draw each data vales as a single pulse.
BLACKMAN, HAMMING, HAHN	window options.

Example for Data.DRAWFFT

```
;set a test pattern to the virtual memory of the TRACE32 application
Data.Set VM:0--0x4f %Byte 1 0 0 0

Data.dump VM:0x0 ;open the Data.dump window to view the test pattern

;visualize the contents of the TRACE32 virtual memory as a graph
Data.DRAWFFT %Decimal.Byte VM:0++0x4f 2.0 512.

Data.DRAWFFT %Decimal.Word VM:0++0x4f 2.0 512.

Data.DRAWFFT %Decimal.Long VM:0++0x4f 2.0 512.

Data.DRAWFFT %Decimal.Quad VM:0++0x4f 2.0 512.
```

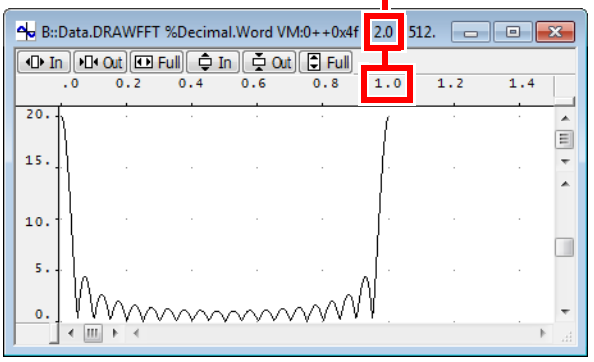
Result:

Resize the window, and then click the two **Full** buttons.

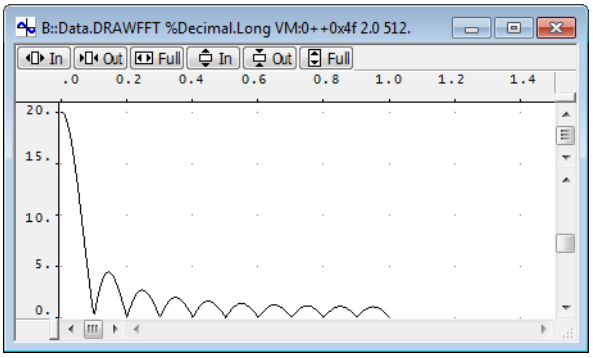


<format>: **Decimal.Byte**

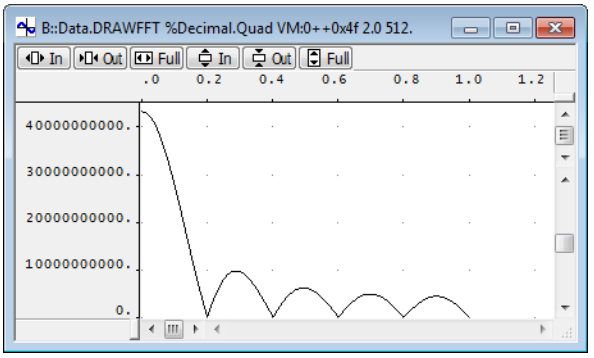
<scale> = 2.0
2.0 divided by 2 = 1.0



<format>: **Decimal.Word**



<format>: **Decimal.Long**



<format>: **Decimal.Quad**

See also

- Data.DRAW
- Data.DRAWXY
- Data.IMAGE
- <trace>.DRAW
- Var.DRAW

▲ 'Release Information' in 'Legacy Release History'

[Example]

Format:	**Data.DRAWXY** [**%***<format>*] *<range_y>* *<range_x>* [*<scale>* [*<offset>*]] [**/***<options>*]
<format>:	**Decimal.**[*<width>*[**.***<endianness>*]] **DecimalU.**[*<width>*[**.***<endianness>*]] **Hex.**[*<width>*[**.***<endianness>*]] **HexS.**[*<width>*[**.***<endianness>*]] **OCTal.**[*<width>*[**.***<endianness>*]] **Float**[**.***<float_rep>*[**.***<endianness>*]] **Byte** [**.***<endianness>*] **Word** [**.***<endianness>*] **Long** [**.***<endianness>*] **Quad** [**.***<endianness>*] **TByte** [**.***<endianness>*] **PByte** [**.***<endianness>*] **HByte** [**.***<endianness>*] **SByte** [**.***<endianness>*]
<width>:	**DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<float_rep>:	**Ieee** | **IeeeRev** | **IeeeS** | **IeeeDbl** | **...**
<options>:	**LOG**| **Track** | **YX** **Vector** | **Points** | **Steps** | **Impulses** **Alternate** *<number>* | **Element** *<number>*

Draws a graph based on array with x and y coordinates.

Decimal, DecimalU,...	Refer to **"Keywords for <format>"**, page 10
Byte, Word, ...	Refer to **"Keywords for <width>"**, page 11
DEFault, BE, LE	Define byte-order display direction: default target endianness, Big Endian or Litte Endian
<scale>	Units per pixel of y-axis (floating point). Default: **1.0**
<offset>	Offset of y-axis (floating point). Default: **0.0**

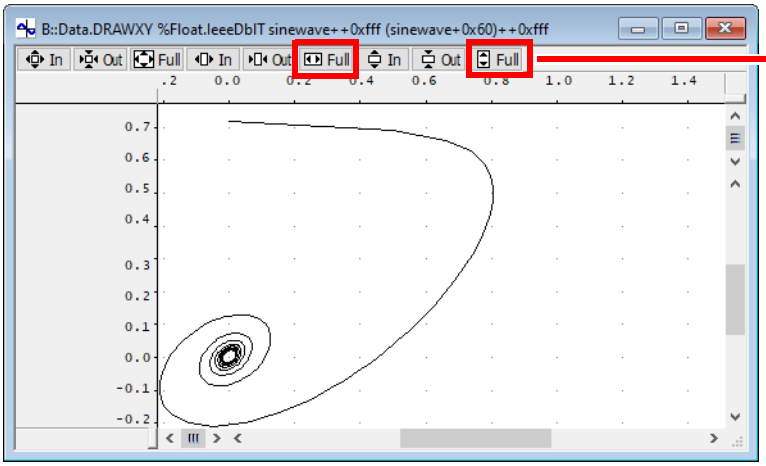
LOG	display y-axis in logarithmic scale
Track	Track the window to the reference address of other windows.
YX	swap *<range_y> and <range_x>.*
Vector (default)	draw each data value as a single pulse.
Points	display each data value as a dot.
Steps	connect the dots for the data values by steps.
Impulses	draw each data vales as a single pulse.
Alternate *<number>*	Split the array in *<number>* graphs. *<number>*=2 first graph display even elements second graph displays odd element. *<number>*=3 first graph displays element 0, n, 2n, … second graph displays 1, n+1, 2n+1, … third graph display 2, n+2, 2n+2, …
Element *<number>*	Specify the structure component to be displayed graphically.

Example

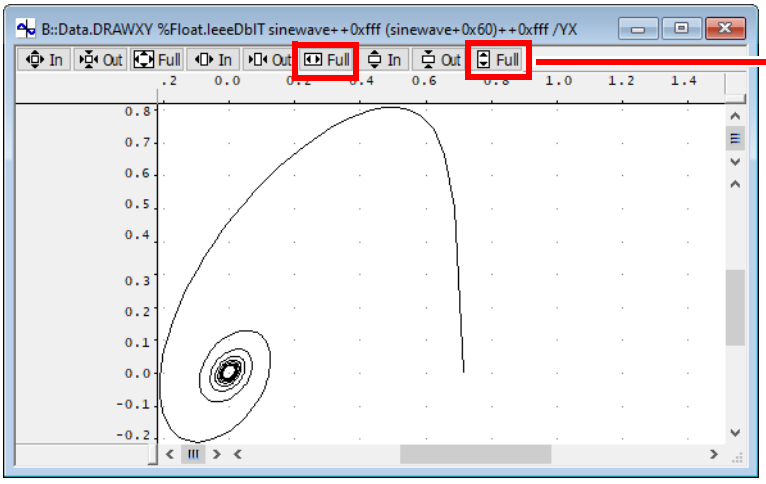
In this example, **Data.DRAWXY** is executed in the TRACE32 Instruction Set Simulator for ARM.

```
SYStem.Up

; Load application
Data.LOAD.Elf ~~/demo/arm/compiler/arm/armle.axf

Register.Set PC main
Go
WAIT 2.s
Break

; See result 1.
;           <format>        <range>          <range>
Data.DRAWXY %Float.IeeeDblT sinewave++0xfff (sinewave+0x60)++0xfff

; See result 2.
Data.DRAWXY %Float.IeeeDblT sinewave++0xfff (sinewave+0x60)++0xfff /YX
```

Result 1:



Result 2:



See also

- Data.DRAW
- Data.DRAWFFT
- Data.IMAGE
- <trace>.DRAW
- Var.DRAW

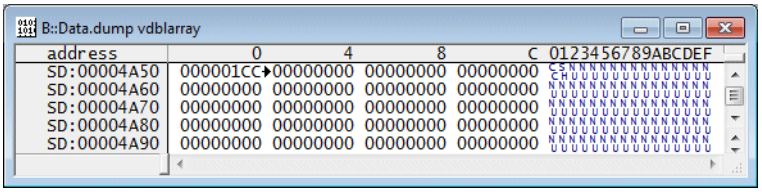
Format:	**Data.dump** [*<address>* | *<range>*] [**/***<option>* …]
<option>: (**format**)	**Hex** | **NoHex** **Decimal** | **DecimalU** **Ascii** | **NoAscii** **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** **BE** | **LE** | **Reverse**
<option>: (**standard**)	**DIALOG** **Track** **CORE** *<core_number>* **Orient** | **NoOrient** **SpotLight** | **NoSpotLight** **STRING** **COLumns** [*<columns>*] **ICache** | **DCache** | **L2Cache** **Mark** *<break>*
<option>: (**advanced**)	**ICacheHits** | **DCacheHits** | **L2CacheHits** **XICacheHits** | **XDCacheHits** | **XL2CacheHits** **COVerage** **CFlag** *<cflag>* **FLAG** *<flag>* **CTS**
<flag>:	**Read** | **Write** | **NoRead** | **NoWrite**
<cflag>:	**OK** | **NoOK** | **NOTEXEC** | **EXEC**
<break>:	**Program** | **HII** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

NOTE: Please be aware that TRACE32 can perform a memory dump only under the following conditions:

- The program execution is stopped.
- Or alternatively, run-time memory access is enabled.

If the parameter is a single address, it specifies the initial position of the memory dump.

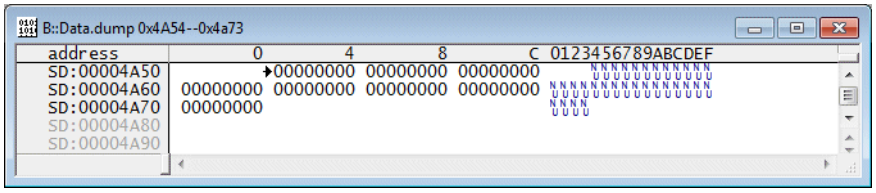
```
Data.dump 0x4A54

Data.dump vdblarray                    ; symbolic address
```



If the parameter is an address range, the memory dump is only displayed for the specified address range.

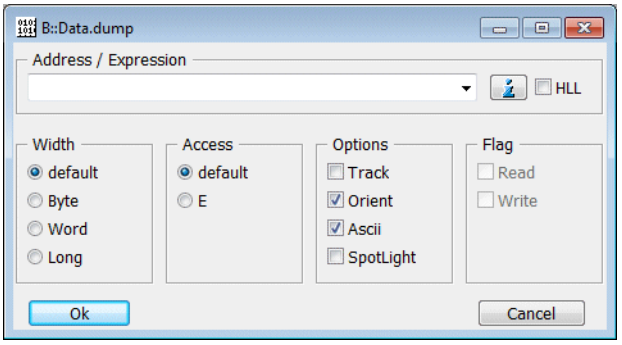
```
Data.dump 0x4A54--0x4A73

Data.dump Var.RANGE(vdblarray)
```

Var.RANGE(<hll_expression>)	Returns the address range occupied by the specified HLL expression



If the **Data.dump** command is entered without parameter the **Data.dump** dialog is displayed.

```
Data.dump
```



The use of access class information might be helpful if you want to specify the memory access more precisely:

```
Data.dump A:0xc3f90004                ; advise TRACE32 to perform an
                                      ; access to physical memory by
                                      ; bypassing the MMU

Data.dump NC:0x5467                   ; advise TRACE32 to perform a
                                      ; non-cached access

Data.dump Z:0x5467                    ; advise TRACE32 to perform a
                                      ; secured access
                                      ;(TrustZone ARM cores)
```

Format Options

[Back to Top]

Byte, **Word**, …	If a memory dump is displayed, TRACE32 PowerView uses the default processing width of the core/processor. Another display format can be specified by format options. See **"Keywords for <width>"**, page 11.

```
Data.dump flags /Byte
```

BE (big endian) **LE** (little endian)	If a memory dump is displayed TRACE32 PowerView uses the default endianness of the core/processor. Another endianness can be specified by the format options **BE** / **LE**.
Decimal **DecimalU** (unsigned dec.)	If a memory dump is displayed TRACE32 PowerView displays the memory contents in hex. The options **Decimal** and **DecimalU** allow a decimal display.

```
Data.dump flags /Decimal

Data.dump flags /DecimalU /Byte
```

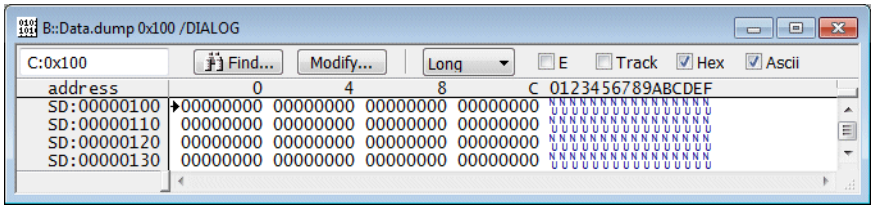
Hex (default) **NoHex**	If a memory dump is displayed TRACE32 PowerView displays an hex representation of the memory contents. This can be suppressed by the option **NoHex**.
Ascii (default) **NoAscii**	If a memory dump is displayed TRACE32 PowerView displays an ASCII representation of the memory contents. This can be suppressed by the option **NoAscii**.
Reverse	Reverses the order of columns in the **Data.dump** window.

Standard Options

[Back to Top]

DIALOG	Display memory dump window with dialog elements.

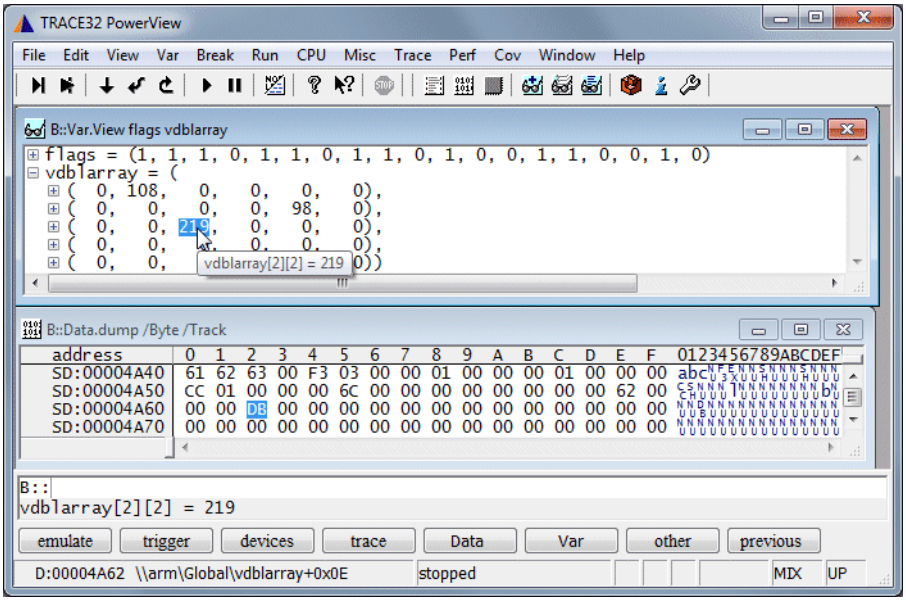
```
Data.dump 0x100 /DIALOG
```



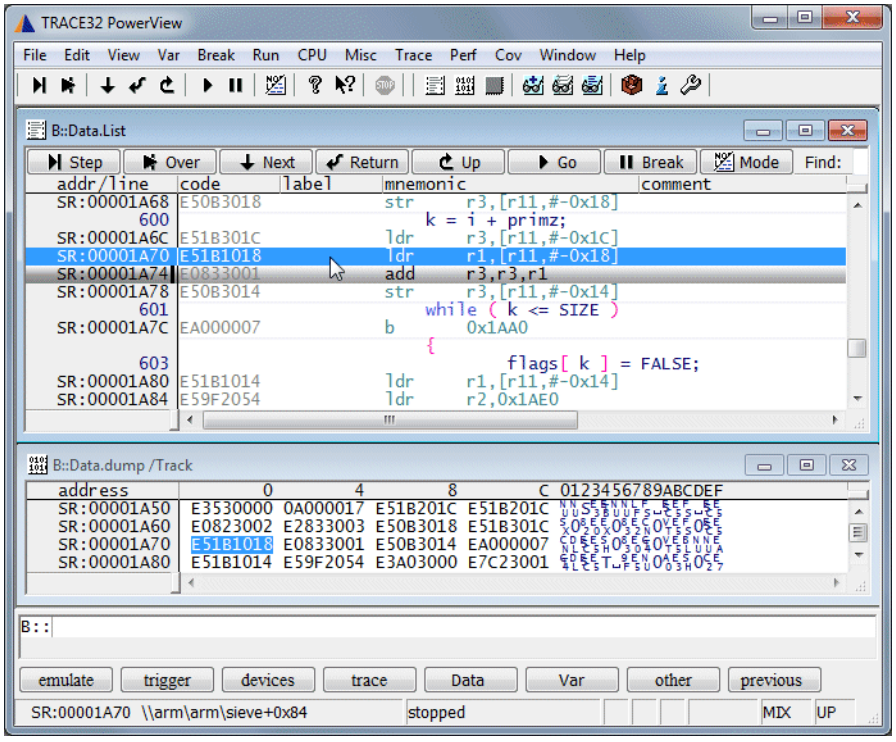
Track	Tracks the window to the reference address of other windows. If this option is combined with a variable argument, like a register value, an argument tracking is performed. This will hold the argument value in the middle of the window and follow the value of the argument.

```
Var.View flags vdblarray

Data.dump /Byte /Track
```

Example for a DATA reference: If the contents of a variable is selected, the corresponding memory location is shown and highlighted in the memory dump window.



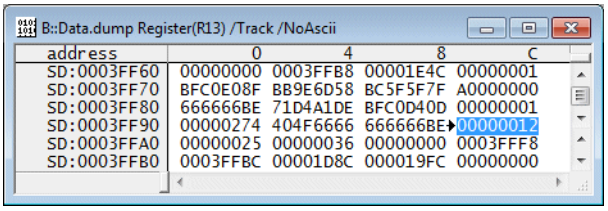
Example for a PROGRAM reference: If a source code line is selected the corresponding memory location is shown and highlighted in the memory dump window.



Example for register tracking:

```
Data.dump Register(R13) /Track /NoAscii
```

Register(<register_name>) Returns register contents



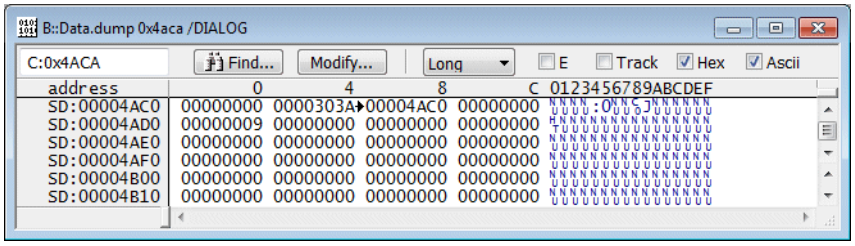
CORE *<number>*	Display memory dump from the perspective of the specified core (SMP debugging only)

TRACE32 assumes that an SMP system maintains memory coherency, so it should not matter from which perspective a memory dump is performed.

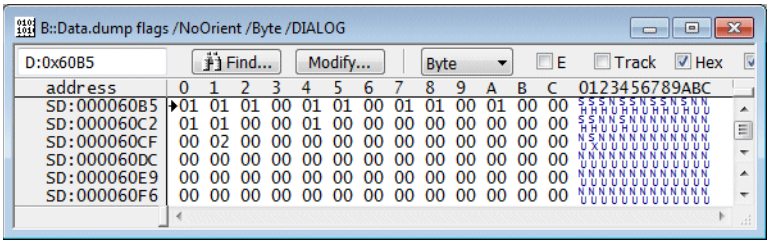
```
CORE.select 0.                          ; select core 0

Data.dump 0x1000                        ; display a memory dump from the
                                        ; perspective of the selected core

Data.dump 0x1000 /CORE 1.               ; display a memory dump from the
                                        ; perspective of the specified core
                                        ; (here core 1)
```

Orient (default)	The start address of the memory dump window matches to bounds of power of two. It is assumed that this is easier to read. The address specified in the **Data.dump** command is marked with a small arrow.
NoOrient	The dump starts exactly at the address specified in the **Data.dump** command.

```
Data.dump 0x4aca /DIALOG
```

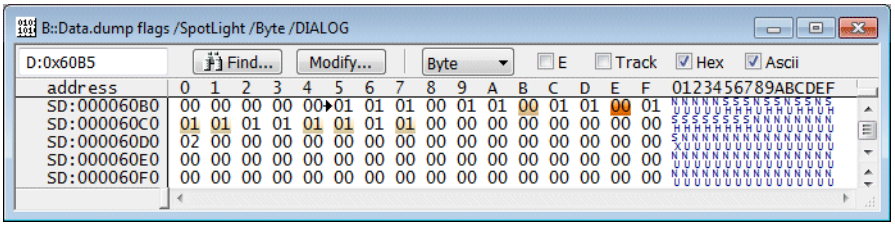



```
Data.dump flags /NoOrient /Byte /DIALOG
```



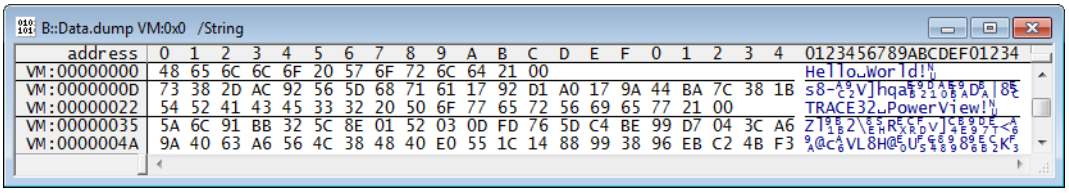
NoSpotLight (default)	No highlighting of changed memory locations.
SpotLight	Highlight changed memory locations. Memory locations changed by the last program run/single step are marked in dark red. Memory locations changed by the second to last program run/single step are marked a little bit lighter. This works up to a level of 4.

```
Data.dump flags /SpotLight /Byte /DIALOG
```



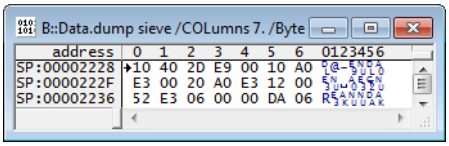
STRING	Display memory dump of zero-terminated string. If the **STRING** option is used, zero-terminated strings are underlined for improved visibility.

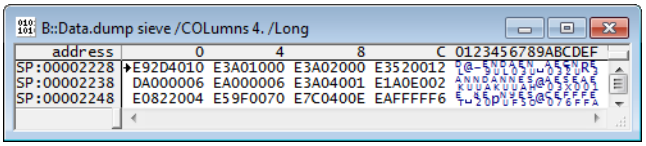
```
Data.dump 0x60b5 /STRING
```



COLumns [*<columns>*] **WIDTH** (deprecated)	Determines how many memory *<columns>* are displayed in the window. The column width can be formatted with the *<format>* options **Byte**, **Word**, **Long**, etc. • To use the TRACE32 default setting, omit option and parameter. When you now resize the window width, the number of columns adjusts to the window width. • **COLumns** *without* the *<columns>* parameter: The number of columns remains fixed when you resize the window width.

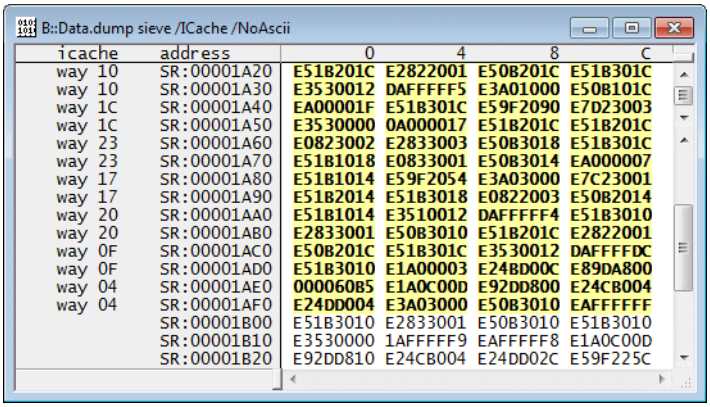
```
;display memory dump with 7 columns and format each col. as 8-bit values
;                 <number_of_columns>   <format>
Data.dump sieve      /COLumns 7.        /Byte

;display memory dump with 4 columns and format each col. as 32-bit values
Data.dump sieve      /COLumns 4.        /Long
```





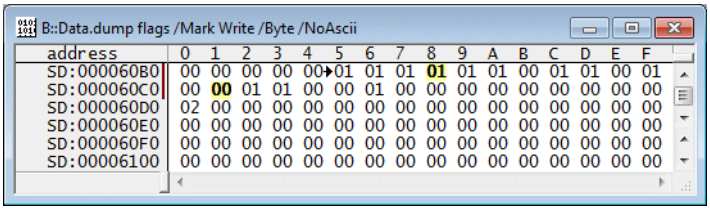
ICache	Highlight memory locations cached to the instruction cache and display way information in the memory dump
DCache	Highlight memory locations cached to the data cache and display way information in the memory dump
L2Cache	Highlight memory locations cached to the level 2 cache and display way information in the memory dump

```
Data.dump sieve /ICache /NoAscii
```



Mark *<break>*	Highlight memory locations for which the specified breakpoint is set.

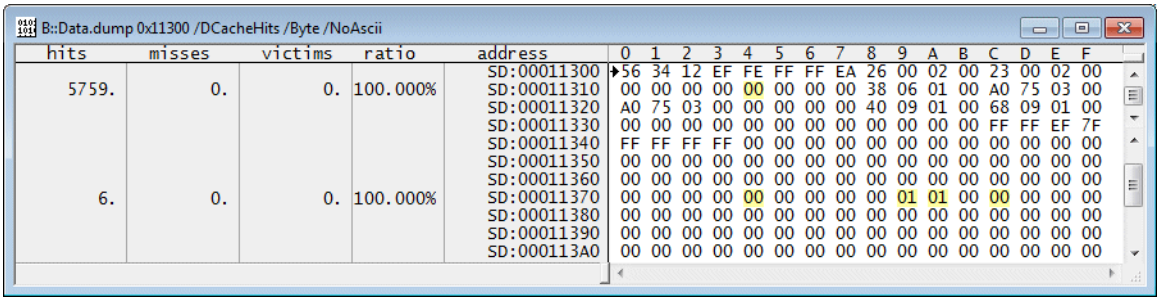
```
; highlight memory locations for which a Write breakpoint is set
Data.dump flags /Mark Write /Byte /NoAscii
```



TRACE32 PowerView uses its default formatting for the **Data.dump** command. These defaults can be changed by the command **SETUP.DUMP**.

The following options are used to map the results of the trace-based cache analysis (**CTS.CACHE**) to memory dumps.

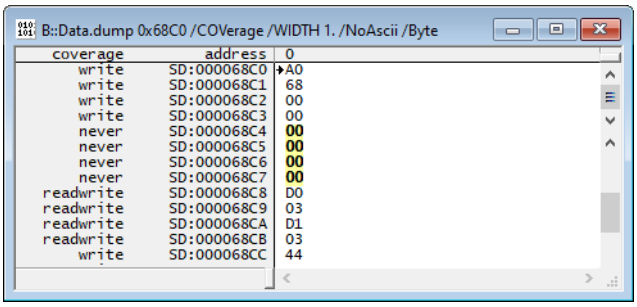
ICacheHits	Highlight memory locations for which instruction cache hits were detected.
DCacheHits	Highlight memory locations for which data cache hits were analyzed.
L2CacheHits	Highlight memory locations for which level 2 cache hits were analyzed.
XICacheHits	-
XDCacheHits	Highlight program memory locations which caused data cache hits.
L2CacheHits	Highlight program memory locations which caused level 2 cache hits.



The following option is used to map the result of trace-based code coverage (**COVerage**) to memory dumps.

COVerage	Highlight program memory location that are never executed respectively data memory locations that are never read/written.

```
Data.dump 0x4e7c /COVerage  /WIDTH 1. /NoAscii /Byte
```



The following option is used to map the results of **CTS** to memory dumps.

CTS	Display **CTS** access information when CTS mode is active.

See also

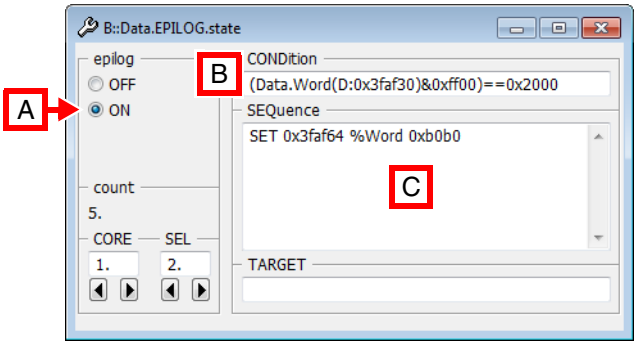
- Data.Assemble
- Data.CHAIN
- Data.Find
- Data.In
- Data.PATTERN
- Data.Print
- Data.STRING
- Data.TABle
- Data.Test
- Data.USRACCESS
- Data.View
- List
- SETUP.DUMP
- SETUP.TIMEOUT
- sYmbol.INFO
- DUMP
- SETUP.ASCIITEXT
- ADDRESS.OFFSET()
- ADDRESS.SEGMENT()
- ADDRESS.STRACCESS()
- ADDRESS.WIDTH()
- Data.Byte()
- Data.Float()
- Data.Long()
- Data.Quad()
- Data.STRing()
- Data.STRingN()
- Data.Word()

▲ 'Release Information' in 'Legacy Release History'

The **Data.EPILOG** command group allows to define a sequence of read/write accesses that are automatically performed directly after the program execution has halted (manual break, breakpoint or end of single step). The complementary command **Data.PROLOG** performs read/write accesses before program execution is continued. It is also possible to store data read with **Data.EPILOG** and restore with **Data.PROLOG**, and vice versa.

The **Data.EPILOG** command group can be used e.g. to manually freeze peripherals, if the processor itself does not provide this feature. Use **Data.EPILOG.SEQuence** and **Data.PROLOG.Sequence** to set up the access sequences.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **Data.EPILOG.state** window.



A For descriptions of the commands in the **Data.EPILOG.state** window, please refer to the **Data.EPILOG.*** commands in this chapter. **Example**: For information about **ON**, see **Data.EPILOG.ON**.

B Conditions can be set up in the **CONDition** field using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** field using the *<data_set_commands>* **SET**, **SETI**, **GETS**, and **SETS**.

Examples:

- Overview including illustration - see **Data.EPILOG.state**.
- Epilog conditions - see **Data.EPILOG.CONDition**.
- Access sequences - see **Data.EPILOG.SEQuence**.

See also

■ Data.ATTACH ■ Data.PROLOG ■ Data.STARTUP ■ Data.TIMER

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.EPILOG.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **Data.EPILOG.SEQuence** will be executed each time after the program execution was stopped.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

Examples:

```
;reads the long at address D:0x3faf30, proceeds a binary AND with
;a constant (here 0xffffffff). If the result is equal to 0x80000000 the
;condition is true and the defined sequence is executed.
Data.EPILOG.CONDition (Data.Long(D:0x3faf30)&0xffffffff)==0x80000000
```

```
;read the word at address D:0x3xfaf30
Data.EPILOG.CONDition (Data.Word(D:0x3faf30)&0xff00)!=0x8000
```

```
;reads the byte at address D:0x3xfaf30
Data.EPILOG.CONDition (Data.Byte(D:0x3faf30)&0xf0)!=0x80
```

See also

■ Data.EPILOG.state ❏ Data.Byte() ❏ Data.Long() ❏ Data.Word()

Format: **Data.EPILOG.CORE** *<core_number>*

Selects the core for which you want to define one or more data epilogs.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: The following example shows how to define a data epilog that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a data epilog
Data.EPILOG.CORE 3.

;Define the data epilog for core 3
Data.EPILOG.CONDition <your_code>
Data.EPILOG.SEQuence  <your_code>
```

For information on how to configure two different data epilogs, see **Data.EPILOG.SELect**.

See also

- Data.EPILOG.state

Data.EPILOG.OFF Switch data epilog off

Format: **Data.EPILOG.OFF**

Disables the execution of the **Data.EPILOG** sequence on program execution halt.

See also

- Data.EPILOG.RESet
- Data.EPILOG.state

Data.EPILOG.ON — Switch data epilog on

Format: **Data.EPILOG.ON**

Enables the execution of the **Data.EPILOG** sequence on program execution halt.

See also

- Data.EPILOG.RESet
- Data.EPILOG.state

Data.EPILOG.RESet — Reset all data epilogs

Format: **Data.EPILOG.RESet**

Switches the **Data.EPILOG** feature off and clears all settings.

See also

- Data.EPILOG.OFF
- Data.EPILOG.ON
- Data.EPILOG.state

Format: **Data.EPILOG.SELect** *<index_number>*

Increments the index number for each new data epilog. This is useful, for example, if you need two separate data epilogs with each data epilog having its own **Data.EPILOG.CONDition**.

TRACE32 automatically assigns the index number 1. to the 1st **Data.EPILOG.SEQuence**. If you require a 2nd, separate data epilog sequence, then increment the *<index_number>* to 2. Otherwise the 2nd data epilog will overwrite the 1st data epilog. You can define a maximum of 10 data epilogs.

Example 1: Two data epilogs with the *same* **Data.EPILOG.CONDition** may have the *same* index number. The backslash **** is used as a line continuation character. No white space permitted after the backslash.

```
;Set the index number to 1.
Data.EPILOG.SELect 1.

;Data epilog sequences shall be executed only if this condition is true:
Data.EPILOG.CONDition (Data.Word(D:0x3faf30)&0xff00)==0x1000

;Define the two data epilog sequences:
Data.EPILOG.SEQuence SET 0x3faf50 %Word 0xA0A0 \
                     SET 0x3faf60 %Word 0xB0B0
```

Example 2: Two data epilogs with *different* **Data.EPILOG.CONDition** settings require two *different* index numbers.

```
;1st data epilog - TRACE32 automatically sets the index number to 1.
Data.EPILOG.SELect 1.

;If this epilog condition is true, ...
Data.EPILOG.CONDition (Data.Word(D:0x3faf30)&0xff00)==0x1000

;... then the 1st epilog sequence will be executed
Data.EPILOG.SEQuence SET 0x3faf50 %Word 0xA0A0

;Increment the index number to define the 2nd data epilog
Data.EPILOG.SELect 2.

;If this epilog condition is true, ...
Data.EPILOG.CONDition (Data.Word(D:0x3faf34)&0xff00)==0x2000

;... then the 2nd epilog sequence will be executed
Data.EPILOG.SEQuence SET 0x3faf54 %Word 0xB0B0
```

See also

- Data.EPILOG.state

Format:	**Data.EPILOG.SEQuence** *<command>* …
<command>:	**SET** *<address>* **%***<format>* *<data>* **SETI** *<address>* **%***<format>* *<data>* *<increment>* **SETS** *<address>* **GETS** *<address>*

Defines a sequence of **Data.Set** commands that are automatically executed by the TRACE32 software directly after the program execution is stopped.

SET Parameters: *<address>* **%***<format>* *<value>*
Write *<value>* with data type *<format>* to *<address>*

SETI Parameters: *<address>* **%***<format>* *<start>* *<increment>*
At the first time performed, write *<start>* to *<address>*.
<start> is incremented by *<increment>* on each successive call.

GETS Parameters: *<address>* **%***<format>*
Reads the value at *<address>* and stores it into an internal data buffer. The internal data buffer can contain multiple records and is reset when the command **Data.PROLOG.Sequence** is called.

SETS Parameters: *<address>* **%***<format>*
If the internal data buffer contains a record for *<address>*, the stored value is written to the processor.

Examples:

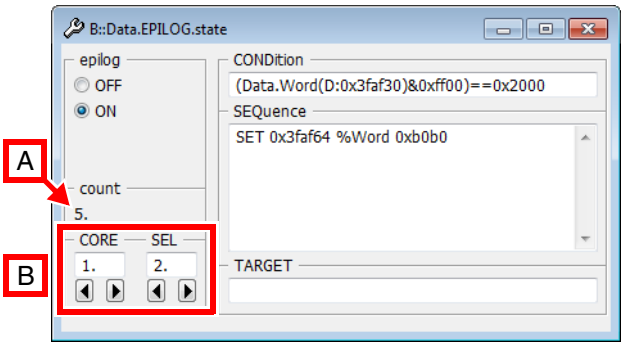
```
;Set peripheral register to 0 when halted, 1 when starting
Data.EPILOG.SEQuence SET 0x3faf50 %Long 0x00000000
Data.PROLOG.SEQuence SET 0x3faf50 %Long 0x00000001

;Set register to 0 when halted, restore original value when starting
Data.EPILOG.SEQuence GETS 0x1230 %Byte SET 0x1230 %Byte 0x00
Data.PROLOG.SEQuence SETS 0x1230 %Byte

;Set (clear) a single bit when starting (stopping)
Data.EPILOG.SEQuence SET 0x3faf50 %Word 0yXXXX1xxxXXXXxxxx
Data.PROLOG.SEQuence SET 0x3faf50 %Word 0yXXXX0xxxXXXXxxxx

;Write 0xa0a0 when starting, increment by 2 for each successive start
Data.PROLOG.SEQuence SETI 0x3faf50 %Word 0xa0a0 2
```

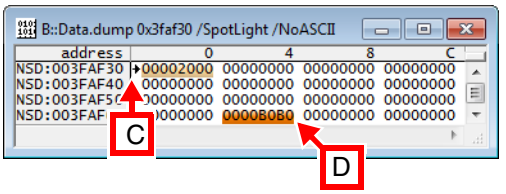
See also

- Data.EPILOG.state

Format: **Data.EPILOG.state**

Opens the **Data.EPILOG.state** window, where you can configure data epilogs.





A Counts the number of times the **Data.EPILOG.SEQuence** command has been executed.

B Lets you create and view the data epilogs of a particular core. This example shows the 2nd data epilog of core 1.
The **CORE** field is grayed out for single-core targets.

C, **D** The **Data.dump** window is just intended to visualize that the **CONDition** [**C**] was true (==0x2000), and thus the **SEQuence** was executed [**D**].

```
Data.EPILOG.state       ;open the window
Data.EPILOG.CORE 1.     ;for core 1, two data epilogs will be defined:

Data.EPILOG.SELect 1.   ;1st data epilog with condition and sequence:
                        ;if condition is true, then execute seq. below
Data.EPILOG.CONDition (Data.Word(D:0x3faf30)&0xff00)==0x1000
Data.EPILOG.SEQuence SET 0x3faf54 %Word 0xa0a0

Data.EPILOG.SELect 2.   ;2nd data epilog with condition and sequence:
                        ;if condition is true, then execute seq. below
Data.EPILOG.CONDition (Data.Word(D:0x3faf30)&0xff00)==0x2000
Data.EPILOG.SEQuence SET 0x3faf64 %Word 0xb0b0

Data.EPILOG.ON          ;activate all data epilogs
Go                      ;start program execution
```

See also

- Data.EPILOG.CONDition
- Data.EPILOG.CORE
- Data.EPILOG.OFF
- Data.EPILOG.ON
- Data.EPILOG.RESet
- Data.EPILOG.SELect
- Data.EPILOG.SEQuence
- Data.EPILOG.TARGET

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.EPILOG.TARGET** *<code_range> <data_range>*

Defines a target program that is automatically started by the TRACE32 software directly after the program execution was stopped.

<code_range> Defines the address range for the target program.

<data_range> Defines the address range used for the data of the target program.

Example:

```
Data.EPILOG.TARGET 0x3fa948--0x3faa07 0x1000--0x1500
```

See also

- Data.EPILOG.state

Format:	**Data.Find** [*<address_range>* [**%***<format>*] *<data>* | *<string>* [**/***<option>*]]
<format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** | **SBYyte** **Float**[**.***<float_rep.>*] **BE** | **LE**
<float_rep>:	**Ieee** | **IeeeRev** | **IeeeS** | **IeeeDbl** | **...**
<option>:	**Back** | **NoFind** | **ALL**

The data/string is searched within the given address range. If it is found, a corresponding message will be displayed.

Without parameters, the **Data.Find** commands will search for the next occurrence of the data/string in the specified address range.

The command can also be executed when using the **Find** button in the **Data.dump … /DIALOG** window.

Byte, **Word**, …	See **"Keywords for <width>"**, page 11.
BE, **LE**	Define the endianness: BigEndian or LittleEndian. The target endianness is used if nothing is specified.

Examples:

```
; search for byte 0x3f in the specified address range
Data.Find 0x100--0xfff 0x3f

; search the next byte 0x3f
Data.Find

; search for specified string
Data.Find 0x100--0xfff "Test"

; search for 32 bit value 0x00001234 in big endian mode
Data.Find 0x100++0xeff %Long %BE 0x1234

; search backward for 16 bit value 0x0089
Data.Find 0x100++0xeff %Word 0x89 /Back

; search for the float 1.45678 in IEEE format
Data.Find 0x4e00--0x4eff %Float.Ieee 1.45678
```

The **Data.Find** command affects the following functions:

FOUND()	Returns TRUE if data/string was found.
TRACK.ADDRESS()	Returns the address of the last found data/string.

```
Data.Find 0x100--0xfff 0x3f

IF FOUND()
     Data.dump TRACK.ADDRESS()
```

```
Data.Find 0x100--0xfff 0x3f

IF FOUND()
     PRINT "Data found at address " TRACK.ADDRESS()
```

The option **/NoFind** sets up the search, but does not process it. This can be beneficial for scripts.

```
OPEN #1 result.txt /Create

&i=1

Data.Find 0x100--0xfff 0x3f /NoFind

RePeat
(
     Data.Find
     WRITE #1 "Address " &i ": " TRACK.ADDRESS()
     &i=&i+1
)
WHILE FOUND()

CLOSE #1

TYPE result.txt

ENDDO
```

See also

- Data.dump
- Data.GOTO
- Data.GREP
- sYmbol.MATCH
- FIND
- WinFIND
- ADDRESS.OFFSET()
- FOUND()
- TRACK.ADDRESS()

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.FindCODE** *<address_range>* *<type>* [*<command>*]
<type>:	**CALL** | **IndirectCALL** | **RETURN** | **JUMP** | *<access>*
<access>:	**ReadWrite** *<address>* | **Read** *<address>* | **Write** *<address>*

Processes the source code in the specified *<address_range>* in order to find the specified instruction type. The command **Data.FindCODE** is mainly used to automatically set the **statistic markers**.

<command> The specified *<command>* is executed on all found program addresses. If you omit *<command>,* all found addresses area printed to the active message **AREA**.

Simple examples:

```
; find all indirect calls in the program address range 0x0++0xffff,
; open a source listing for each found indirect call
Data.FindCODE 0x0++0xffff IndirectCALL "List"

; find all returns in the function sieve and set an onchip breakpoint to
; all found returns
Data.FindCODE sieve RETURN "Break.Set * /Onchip"
```

```
; find all write accesses to the address flags+3 in the function sieve,
; open a source listing for each found write access
Data.FindCODE sieve Write flags+3 "List"

; find all read accesses to the integer variable mstatic1 in the function
; func2, open a source listing for each found read access
Data.FindCODE func2 Read mstatic1 "List"
```

Statistic marker examples:

```
; find all returns in the address range OSLongJmp++0x3F and set a
; statistic marker of the type FEXITCLEANUP to all found program
; addresses
Data.FindCODE OSLongJmp++0x3F RETURN \
"sYmbol.MARKER.Create FEXITCLEANUP *"

; find all write accesses to the address TASK.CONFIG(magic[1]) in the
; function OSTaskInternalDispatch, set a statistic marker of the type
; CORRELATE to all found program addresses
Data.FindCODE OSTaskInternalDispatch Write TASK.CONFIG(magic[1]) \
"sYmbol.MARKER.Create CORRELATE *"

; find all indirect calls in the function OSTaskInternalDispatch and set
; a statistic marker of the type CLEANUP to all found program addresses
Data.FindCODE OSTaskInternalDispatch IndirectCALL \
"sYmbol.MARKER.Create CLEANUP *"
```

Data.GOTO Specify reference address for address tracking

Format: **Data.GOTO** [*<address>*]

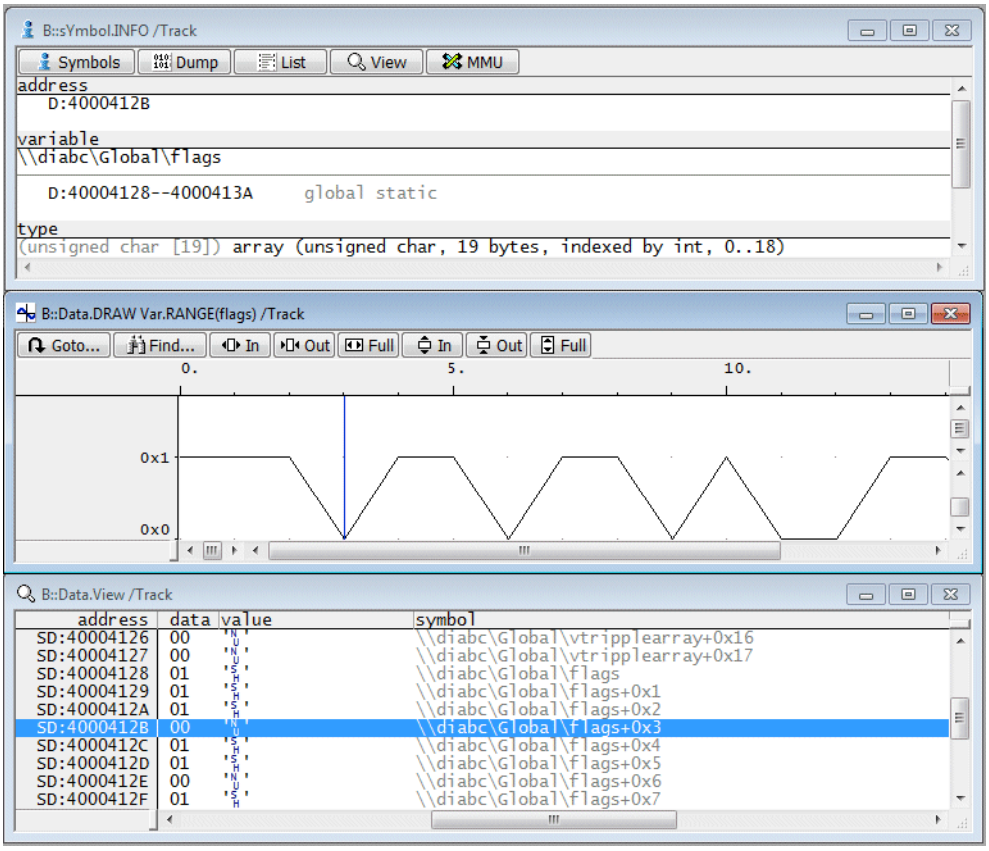
The given address is used for tracking the windows (like **FIND** / **ComPare** commands).

Example1: Tracks all windows that have the **/Track** option to program address func10.

```
List.Mix /Track
sYmbol.INFO /Track
PERF.ListFunc /Track
Data.GOTO func10
```

Example2: Tracks all windows that have the **/Track** option to the datat address `flags+3`.

```
Data.View /Track
sYmbol.INFO /Track
Data.DRAW Var.RANGE(flags) /Track
Data.GOTO flags+3
```



See also

- Data.GREP
- Data.Find
- FIND
- WinFIND
- TRACK.ADDRESS()

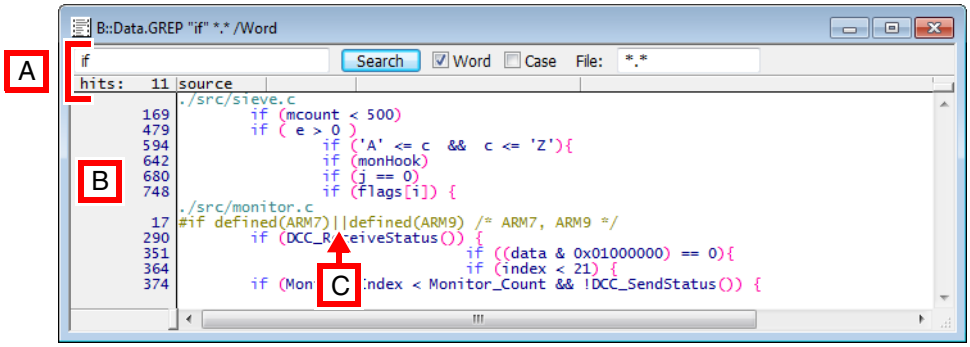
Format:	**Data.GREP** *<string>* [*<file>* [*/<option>*]]
<option>:	**Word** **Case** *<other_options>*

Searches for a specific string in one or all source files; regular expressions are not supported

Word	Search for the whole word.
Case	Perform a case sensitive search.
<other_options>	For descriptions of the other options, see **List**.

Example:

```
Data.GREP "func"              ; search for "func" in all source files

Data.GREP "func" */sieve.c    ; search for "func" only in sieve.c

Data.GREP "if" *.* /Word      ; search for the word "if" in all source
                              ; files
```



A Search *<string>* and number of hits.

B Line numbers of hits.

C Double-clicking a function opens a listing for the selected function in a **List** window. Right-clicking a function opens the **Function** popup menu.

See also

- Data.GOTO
- Data.Find
- APU.GREP
- FIND
- WinFIND
- WinOverlay

[Examples]

Format: **Data.IMAGE** *<address> <horiz> <vert>* [*<scale>*] [**/***<format>* [**/***<option>*]]

<format>:
MONO | **MONOLSB** | **CGA** | **GrayScale8** | **JPEG**
RGB111 | **RGB555** | **RGB555LE** | **RGB565** | **RGB565LE** |
RGB888 | **RGB888LE** | **RGBX888** | **RGBX888LE** | **XXXA888**
RGBCUSTOM | **RGBCUSTOMLE**
YUV420 | **YUV420W** | **YUV420WS**
YUV422 | **YUV422W** | **YUV422WS** | **YUV422P** | **YUV422PS**
Palette256 *<red> <green> <blue> …*
Palette256X6 *<address>*
Palette256X12 *<address>*
Palette256X24 *<address>*

<option>:
BottomUp
FullUpdate
STRIDE *<bytes_per_stride>*
SignedY | **SignedU** | **SignedV**
UVPLANEbase | **UPLANEbase** | **VPLANEbase**
RGBBITS "<[**RGBXA**]>"

Displays graphic bitmap data. Zooming is supported by scrolling the mouse wheel or double-clicking the image. Right-clicking an image allows advanced data operations.

<address>	Base address of the image, e.g.: D:0x10000
<horiz>	Horizontal size of the image (decimal value with postfix '.')
<vert>	Vertical size of the image (decimal value with postfix '.')
<scale>	Initial scale of the image (zoom value)

MONO	Monochrome bitmap (default format). Each byte represents eight consecutive pixels. The MSB is the leftmost bit.
MONOLSB	Monochrome bitmap. Each byte represents eight consecutive pixels. The LSB is the leftmost bit.
CGA	Colors compatible to the CGA (Color Graphics Adapter). In this mode each byte represents two pixels, each nibble can encode 16 predefined colors.
GrayScale8	Each byte contains one pixel with 256 shades of gray.
JPEG	JPEG compressed image.

RGB111 (RGB)	Color display. One byte represents one pixel. Bit 0 is the blue color, bit 1 is green, bit 2 is red. All bits clear displays the background color, all three bits set displays the foreground color (host dependent).
RGB555 RGB555LE (BGR555)	Two bytes make up one pixel. First bit ignored, 5 bit red, 5 bit green, 5 bit blue. For RGB555LE the order is ignore-blue-green-red.
RGB565 RGB565LE (BGR565)	Two bytes make up one pixel. 5 bit red, 6 bit green, 5 bit blue. For RGB565LE the order is blue-green-red.
RGB888 (RGB24) RGB888LE (BGR24)	Three bytes make up one pixel. The first byte contains 256 shades of red, the second byte green and the third byte blue. For RGB888LE the order is blue-green-red.
RGBX888 (RGB32) RGBX888LE (BGR32)	Four bytes make up one pixel. The first byte contains 256 shades of blue, the second byte green and the third byte blue. the fourth byte is ignored. For RGBX888LE the order is blue-green-red-ignore.
XXXA888 (ALPHA)	Four bytes make up one pixel. The first three bytes are ignored (X), the alpha (A) channel is displayed as a grayscale image (256 shades of gray).
RGBCUSTOM	Custom RGB format, needs to be used in conjunction with /RGBBITS option.
YUV420	YUV encoded, three separate planes (4xY,1xU,1xV).
YUV420W	YUV encoded, two planes of 16bit words (4xY,1xUV).
YUV420WS	YUV encoded, two planes of 16bit words, byte swapped (4xY,1xVU).
YUV422	YUV encoded, three separate planes (2xY,1xU,1xV).
YUV422W	YUV encoded, two planes of 16bit words (2xY,1xUV).
YUV422WS	YUV encoded, two planes of 16bit words, byte swapped (2xY,1xVU).
YUV422P	YUV encoded, one plane of 32bit words (Y,U,Y,V).
YUV422PS	YUV encoded, one plane of 32bit words, byte swapped (U,Y,V,Y).
Palette256	Full color display. One byte represents one pixel. The byte selects one of 256 different color values in the palette defined by the parameters.
Palette256X6	Full color display. One bytes represents one pixel. The byte selects one of 256 different color values in the palette read from memory. Each palette value is a byte containing 2 bits for the intensity of each color.

Palette256X12	Full color display. One bytes represents one pixel. The byte selects one of 256 different color values in the palette read from memory. Each palette value is a 16 bit word containing 4 bits for the intensity of each color.
Palette256X24	Full color display. One bytes represents one pixel. The byte selects one of 256 different color values in the palette read from memory. Each palette value is a 32 bit long containing 8 bits for the intensity of each color.
BottomUp	Mirrors the image horizontally.
FullUpdate	Performs a complete redraw each time the window is updated. The default is to update the window step by step to keep the response time of the debugger fast.
STRIDE	Number of bytes for one row of pixels in memory (image width in bytes plus padding bytes).
SignedY	Y values of YUV encoded images are treated as signed values.
SignedU	U values of YUV encoded images are treated as signed values.
SignedV	V values of YUV encoded images are treated as signed values.
UVPLANEbase	Specify a separate base address for the UV-plane of a YUV image (instead of assuming to find it consecutive to the Y-plane). This option is available for YUV formats with two planes, e.g. YUV420W.
UPLANEbase	Specify a separate base address for the U-Plane of a YUV image (instead of assuming to find it consecutive to the Y-plane). This option is available for YUV formats with three planes, e.g. YUV420.
VPLANEbase	Specify a separate base address for the V-Plane of a YUV image (instead of assuming to find it consecutive to the Y- and U-planes). This option is available for YUV formats with two planes, e.g. YUV420.
RGBBITS	Define bits for custom RGB format. The format must be passed as string, containing one or more of the following characters: "RrGgBbXxAa". Each character represents one bit (Red, Green, Blue, Ignore, Alpha/Gray). The memory access is aligned to the next byte. See demo scripts for examples.

Example to show a 50x40 pixel true color bitmap image:

```
; load the image into virtual memory skipping bmp header
Data.LOAD.Binary image.bmp VM:0x0 /OFFSET 0x36


; stride is (50.*3.+3)&~0x3
Data.IMAGE VM:0x0 50. 40. /RGB888LE /BottomUp /STRIDE 152.
```

More examples are available in the ~~/demo directory:

```
PSTEP ~~/demo/practice/image/*.cmm
```

See also

- Data.DRAW
- Data.DRAWFFT
- Data.DRAWXY
- <trace>.DRAW
- Var.DRAW

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.In** *<address>* [*<count>*] [*/<options>*]
<options>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** **BE** | **LE** **Repeat** | **INCrement** | **CORE** *<core_number>*

This command reads data from the specified address and prints it to the message line. The read access occurs either once or the specified number of repetitions. The read address does not increment during the repetitions, unless option /INCrement is set. If the number of repetitions exceeds a certain amount, the output in the message line will be truncated.

Byte, **Word**, …	See **"Keywords for <width>"**, page 11.
BE, LE	Endianness: BigEndian or LittleEndian
Repeat	Repeats the input endlessly, e.g. for external measurements.
INCrement	Address is incremented by *<width>* with each repeated access.
CORE *<number>*	Read memory from the perspective of the specified core /SMP debugging only).

Example:

```
;read a byte from address 0x40
Data.In D:0x40

;read a 32-bit word from address 0x40, repeat 4 times
Data.In D:0x40 4. /Long

;read a 32-bit word from addresses 0x40 and 0x44
Data.In D:0x40 2. /Long /INCrement
```

See also

- Data.dump
- Data.Float()
- Data.STRingN()
- Data.Out
- Data.Long()
- Data.Word()
- Data.View
- Data.Quad()
- Data.Byte()
- Data.STRing()

Data.List Display Source Listing (deprecated)

The commands **Data.List**, **Data.ListAsm**, etc. have been renamed to **List.auto**, **List.Asm**, etc. The old **Data.List*** commands continue to be available.

Format:	**Data.LOAD** *<file>* [*<address>*] [*<range>*] [**/***<load_option>*] **Data.LOAD.auto** *<file>* [*<address>*] [*<range>*] [**/***<load_option>*] **Data.LOAD.** *<file_format><file>* [*<address>*] [*<range>*] [**/***<load_option>*]
<generic_load_option>:	**Verify** | **PVerify** | **NoVerify** | **CHECKLOAD** [*<address_range>*] **ComPare** | **DIFF** | **CHECKONLY** [*<address_range>*] **ZIPLOAD** [*<code_range>* [*<data_range>*]] **DIFFLOAD** [*<address_range>*] **DualPort** (EF) **Byte** | **Word** | **TByte** | **Long** | **PByte** | **HByte** | **SByte** | **Quad** **BSPLIT** *<width> <offset>* **wordSWAP** | **LongSWAP** | **QuadSWAP** | **BitSwap** **VM** | **PlusVM** **NoCODE** | **NosYmbol** | **NoRegister** | **NoBreak** | **NOFRAME** **NoClear** | **More** **PATH** *<dir>* **SOURCEPATH** *<dir>* **StripPATH** | **LowerPATH** **StripPART** *<parts>* | *<part_name>* **StripBeforePART** *<pattern>* **TASK** *<task>* **NAME** *<name>* **MAP** | **DIAG** **Include** | **NoInclude** **COLumns** | **MACRO** **CYGDRIVE** **SingleLine** | **SingleLineAdjacent** **NoTranspose** **…**
<architecture_specific_load_option>:	**LARGE** | **LDT** | **SingleLDT** | **FLAT** | **ProtectedFLAT** [*<code_descriptor><data_descr> <stack_descr>*] (only 386) **SPlit** (only 80196)

The debugger tries to detect the data format of the file automatically when the command **Data.LOAD.auto** (or **Data.LOAD**) is used. The automatic detection is not possible for all formats. In this case please use **Data.LOAD.<file_format>**.

Only the generic options can be used with **Data.LOAD.auto**. All options described below are available for **Data.LOAD.auto** and ALL formats of **Data.LOAD.<file_format>**. There are also options which are only usable for a specific file format. These options are only available if **Data.LOAD.<file_format>** is used (see following commands).

The parameter *<address>* and *<range>* are file format dependent:

<address>	• File format without address information (e.g. binary): base address • File format with address information: address offset
<addressrange>	• If specified, only the data within the address range will be loaded. Data outside this address range will be ignored. • If specified for file formats without address information, the start address of *<range>* is used as base address and *<address>* will be ignored.

Alphabetic List of Generic Load Options

BitSwap	Swap the bits of each byte during load.
Byte **Word** **TByte** **Long** **PByte** **HByte** **SByte** **Quad**	Data is loaded in the specified width: • **Byte** (8-bit accesses) **Word** (16-bit accesses) • **TByte** (24-bit accesses) **Long** (32-bit accesses) • **PByte** (40-bit accesses) **HByte** (48-bit accesses) • **SByte** (56-bit accesses) **Quad** (64-bit accesses) Must be used if the target can only support memory accesses of a fixed width. The default width is determined automatically by TRACE32 to achieve the best download speed.
BSPLIT *<stride>* *<offset>* [*<width>*]	Loads only certain bytes of the memory. • *<stride>* defines a chunk of data which the other two parameters relate to. • *<offset>* defines the offset of the bytes being saved. • *<width>* defines the bus width in bytes. For an illustration of *<stride>*, *<offset>*, and *<width>*, see below. The option **BSPLIT 2 0** loads the lower byte of a 16-bit bus.
CHECKDIFF	Checks the target code with the target agent and reports the result in the **FOUND()** function.
CHECKLOAD	See **CHECKLOAD**.
CHECKONLY	See **CHECKONLY**.
COLumns	Loads information for single stepping columns in HLL mode. May not be available in all file formats.
Compare	See **Compare**.
CutPATH	Deprecated. Cuts name in path to 8 characters.

CYGDRIVE	Use this option to make TRACE32 aware of object files compiled within a Cygwin environment (e.g. the Xilinx MB compiler). This will strip the prefix c:\cygdrive\c\ from source paths so TRACE32 looks for source files at the correct location in the file system.
DIAG	Enable diagnostic messages, which are shown in the **AREA** window during loading.
DIFF	See **DIFF**.
DIFFLOAD	See **DIFFLOAD**.
DualPort (EF)	Data is stored directly to dual-port memory where possible. Data is stored regular if there is no memory mapped at the target address. This option can speed up the download of code by a factor between 2 and 10. It should be used whenever possible, i.e. when the most part of the code is downloaded into emulation memory.
FIXPATH	Deprecated. Remove duplicates of // or \\ from path.
FLASHONLY	Loads the file just to the defined FLASH memories. You can view the defined FLASH memories in the **FLASH.List** window. The number of dropped bytes is displayed in the TRACE32 message line.
FRAME	Consider the stack frame information of the symbol information of the loaded file. (E.g. consider section ".debug_frame" of an ELF file) The stack frame information in the file is used for the **Frame** window. Without this option the TRACE32 tool tries to analyze the function prolog code to get the stack frame. This options is enabled by default for the following CPU families: MMDSP+, Nios II, ARC, C166, Hexagon, APS, Intel X86, Ubicom32 (You can disable it with NOFRAME)
GO	Start target CPU after loading the target program.
GTLDMALOAD *<offset>*	Forces the **Data.LOAD.*** command to use the back-door memory access of the emulation system that is configured by the command **SYStem.GTL.DMANAME**. The **Data.LOAD.*** command can be executed in **SYStem.Mode.Down**, because it does not use the debug capabilities of the CPU. For an example, see below.
HIPERLOAD *<target_ip_addr>*	High performance load. Sends the data via UDP packets to the target - bypassing the JTAG interface.
Include	Activates the loading of source lines, which are generated from include files. By default this option is enabled. Disable it with option **NoInclude**.
LowerPATH	See **LowerPATH**.

LongSWAP	Swaps high and low bytes of a 32-bit word during load.
MACRO	Loads information from C Macros for HLL debugging. May not be available in all file formats.
MAP	Generates memory load map information and checks for overlapping memory writes during download. The load map information can be examine with **sYmbol.List.MAP**. The option can be useful if the load map is questionable. This option is nowadays enabled by default. Disable it with **NOMAP**.
MERGE [\\<*program-name*>]	Merges debug information from the loaded file with already loaded plain symbol information of another program. Useful when debug information was stripped of the work file but is available in another file.
More	This option speeds up the download of large projects consisting of multiple files. The option suppresses the database generation process after loading. The option must be set on all load commands, except the last one.
MultiLine	Allows to show a single source line at multiple target locations. The default for most debug formats.
NAME	Overwrites the program name with the specified one. This option is useful when the same copy of one program is loaded more than once (e.g. in multitask environments).
NoClear	Existing symbols are not deleted. This option is necessary if multiple programs must be loaded (Tasks, Overlays, Banking).
NoCODE	Suppress the code download. Only loads symbolic information.
NOFRAME	Ignore the stack frame information of the symbol information of the loaded file. (E.g. ignore section ".debug_frame" of an ELF file) The stack frame information in the file is used for the **Frame** window. Use this option, if your compiler doesn't produce the correct information. The TRACE32 tool tries then to analyze the function prolog code to get the stack frame.
NoInclude	Deactivates the loading of source lines generated from include files. (By default, these source lines are included.)
NoINCrement	Loads code to a single address (of a FIFO).
NOMAP	During program download the debugger generates usually load map information and checks for overlapping memory writes. The load map information can be examine with **sYmbol.List.MAP**. The option **NOMAP** disables the generation and checking of load map information.
NoRegister	Any startup values for registers (e.g. Program Counter) are not taken from the file.

NosYmbol	No symbols will be loaded (even no program symbol). This option should be used, when pure data files are loaded.

NoTranspose	Module and program names are transcoded per default to avoid reserved characters.
	Examples:
	`my-main` is transcoded to `my_name`
	`mm\init.c` is transcoded to `mm__init.c`
	Path separators are normally transcoded to double underscores, other special characters are transcoded to underscores.
	This can be disabled using the **/NoTranspose** option.

NoVerify	See **NoVerify**.
PATH	See **PATH**.
PlusVM	The code is loaded into target memory plus into the virtual memory.
PVerify	See **PVerify**.
QuadSWAP	Swaps high and low bytes of a 64-bit word during load.
Register	Initialize some registers (e.g. Program Counter) with startup values taken from the file. This is usually enabled by default. Disable it with option **NoRegister**.
SingleLineAdjacent	See **SingleLineAdjacent**.
SOURCEPATH	See **SOURCEPATH**.
StripPATH	See **StripPATH**.
StripPART	See **StripPART**.
TASK	Defines the magic word for the program of this task. This option is only supported for specific processors, which have a built-in MMU (e.g. 68040/60). For more information about the usage of this option, refer to the Processor Architecture Manual.
TYPEMAX *<number>*	Maximum number of types allowed in one single module when type information is compressed.
TypesOnly	Loads only type information.
Verify	See **Verify**.

VM	The TRACE32 software provides a virtual memory (VM:) on the host. With this option the code is loaded into this virtual memory. The virtual memory is mainly used for program flow traces e.g. MPC500/800, ARM-ETM … Since only reduced trace information is sampled, the TRACE32 software also needs the code from the target memory in order to provide a full trace listing. If the on-chip debugging logic of the processor doesn't support memory read while the program is executed a full trace display can only be provided if the program execution is stopped. If the code is loaded into the virtual memory the TRACE32 software can use code from the virtual memory in order to provide a full trace listing.
wordSWAP	Swaps high and low bytes of a 16-bit word during load.
ZIPLOAD	See **ZIPLOAD**.

Details on Generic Load Options

Options which verify that code blocks were written error-free to the target memory	
Verify	Data memory is verified after the complete code has been downloaded to the target. The option also slows down the download process by about three times. See also the **ComPare** option.
CHECKLOAD	Data memory is checked after writing by calculating checksums. Recommended if large files are loaded to targets with a slow upload speed. Checksums over the memory blocks are built by a so-called target agent. The target agent is part of the TRACE32 software and is automatically loaded at the end of the loaded data. If this is not practicable it is also possible to define an at least 64K byte *<address_range>* for the target agent.
PVerify	Partial verify. Same as verify, but only first part of each continuous memory section will be verified. Faster than verify, but still provides some kind of memory checking.
NoVerify	Minimum verification is also turned off. This verification includes checking for existing dual-port memory when loading to dual-port memory and checking for ROM limits when loading to a ROM monitor. With this option all this code outside limits will be silently thrown away.

Examples:

```
Data.LOAD.Elf arm.elf /Verify

Data.LOAD.Elf diabp8.x /CHECKLOAD

Data.LOAD.Elf diabp8.x /CHECKLOAD 0xA0000000++0xFFFF
```

Options that allow to check whether the data in memory match the data in the file. Memory is not changed.	
ComPare	Data is compared against the file by reading the data from memory. Memory is not changed. The comparison stops after the first difference.
CHECKONLY	Data is compared against the file by calculating checksums. Memory is not changed. The comparison stops when checksum is wrong. Recommended if large files are loaded to targets with a slow upload speed. Checksums over the memory blocks are build by a so-called target agent. The target agent is part of the TRACE32 software and is automatically loaded at the end of the data. If this is not practicable it is also possible to define an at least 64K byte *<address_range>* for the target agent.
DIFF	Data is compared against the file, memory is not changed. The result of the compare is available in the **FOUND()** and **TRACK.ADDRESS()** function.

Examples:

```
; check if diabp8.x is already loaded by calculating checksums on data
; in target memory
Data.LOAD.Elf diabp8.x /CHECKONLY

…
; Load code from binary and verify specific sections with the Elf file
Data.LOAD.Binary f.bin 0x0
Data.LOAD.Elf f.elf /NoCODE
Data.LOAD.Elf f.elf sYmbol.SECRANGE(".text") /DIFF /NoRegister /NosYmbol
IF FOUND()
(
     PRINT ADDRESS.OFFSET(TRACK.ADDRESS())
)
…
```

Options to improve the download speed for debug ports with slow download	
DIFFLOAD	Downloads only changed code in a compressed form via a target agent. The target agent is part of the TRACE32 software and is automatically loaded at the end of the data. If this is not practicable it is also possible to define an at least 64K byte *<address_range>* for the target agent. Switching the instruction cache ON before loading improves the download performance. DIFFLOAD is recommended for fast targets with a slow download speed (i.e. lower then 100 KBytes/s).
ZIPLOAD	Data are zipped before the download and unzipped on the target by a so-called target agent. Recommended if large files are loaded to targets with a slow download speed. The target agent is part of the TRACE32 software and is automatically loaded at the end of the data. If this is not practicable it is also possible to define a *<code_range>* and a *<data_range>* respectively for the code and data of the target agent. If no *<data_range>* is specified, the target agent data will be stored after its code. The specified ranges should be within RAM.

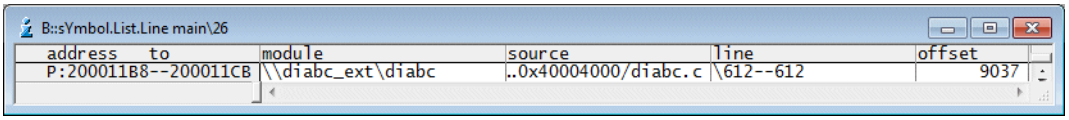
Examples:

```
; first download in standard speed
Data.LOAD.Elf demo.elf /DIFFLOAD
;…
; next download with improved speed
Data.LOAD.Elf demo.elf /DIFFLOAD
;…

; load diabp8.x via ZIPLOAD
Data.LOAD.Elf diabp8.x /ZIPLOAD

Data.LOAD.Elf diabp8.x /DIFFLOAD /ZIPLOAD
```

Options to change the mapping between HLL source code line and blocks of assembler lines	
SingleLineAdjacent	Adjacent blocks of assembler code generated for an HLL line are concentrated.
SingleLine	All blocks of assembler code generated for an HLL line are concentrated.

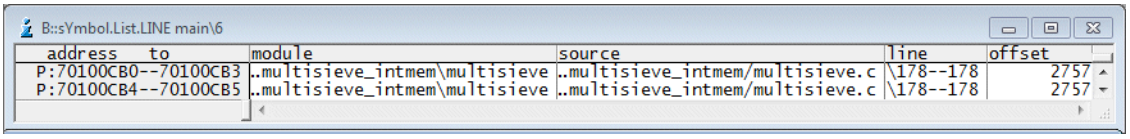
The debug information loaded from *<file>* provides the mapping between HLL source code lines and the blocks of assembler code generated for these lines. Their are mainly three types of mapping:

1. A continuous block of assembler code is generated for an HLL line.


B::sYmbol.List.Line main\26

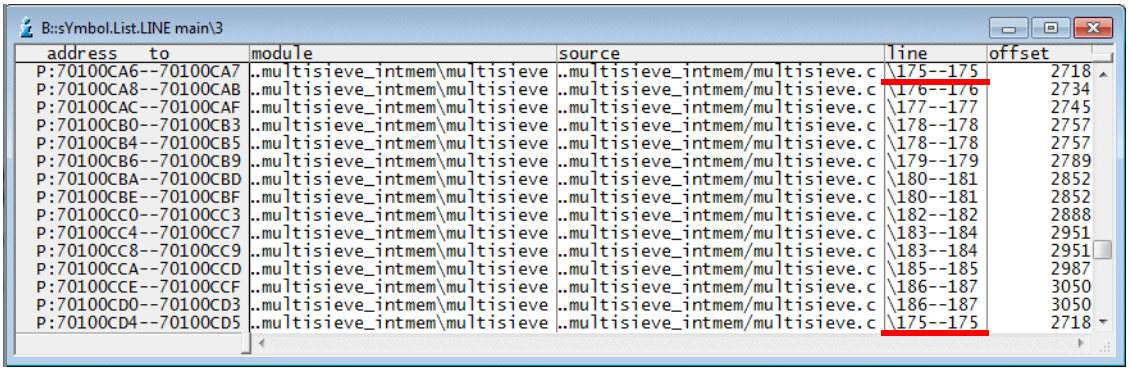
address to	module	source	line	offset
P:200011B8--200011CB	\\diabc_ext\diabc	..0x40004000/diabc.c	\612--612	9037

2. Two or more adjacent blocks of assembler code are generated for an HLL line.


B::sYmbol.List.LINE main\6

address to	module	source	line	offset
P:70100CB0--70100CB3	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\178--178	2757
P:70100CB4--70100CB5	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\178--178	2757

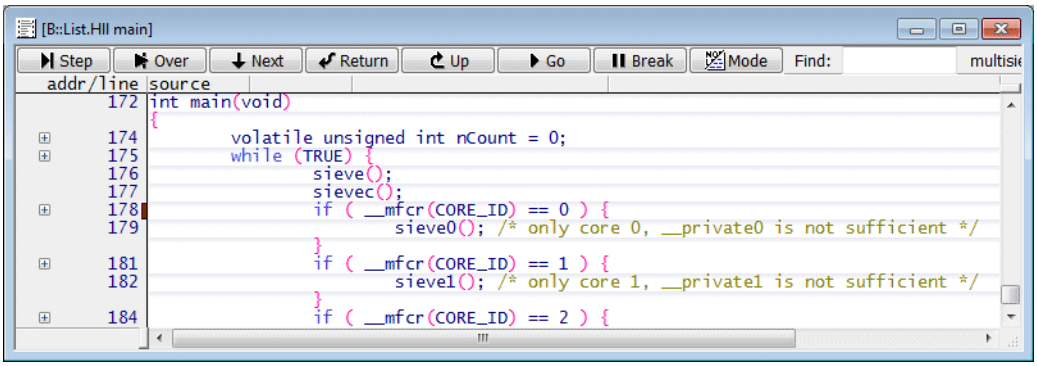
3. Two or more detached blocks of assembler code are generated for an HLL line.


B::sYmbol.List.LINE main\3

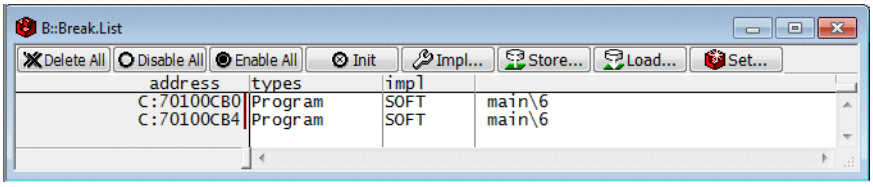
address to	module	source	line	offset
P:70100CA6--70100CA7	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\175--175	2718
P:70100CA8--70100CAB	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\176--176	2734
P:70100CAC--70100CAF	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\177--177	2745
P:70100CB0--70100CB3	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\178--178	2757
P:70100CB4--70100CB5	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\178--178	2757
P:70100CB6--70100CB9	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\179--179	2789
P:70100CBA--70100CBD	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\180--181	2852
P:70100CBE--70100CBF	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\180--181	2852
P:70100CC0--70100CC3	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\182--182	2888
P:70100CC4--70100CC7	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\183--184	2951
P:70100CC8--70100CC9	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\183--184	2951
P:70100CCA--70100CCD	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\185--185	2987
P:70100CCE--70100CCF	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\186--187	3050
P:70100CD0--70100CD3	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\186--187	3050
P:70100CD4--70100CD5	..multisieve_intmem\multisieve	..multisieve_intmem/multisieve.c	\175--175	2718

It has the following effects on debugging if more the on block of assembler code is generated for an HLL line:

- The HLL line is marked with a drill-down box.



- If a breakpoint is set to the HLL line, a breakpoint is set to every block of assembler code.



If the option **SingleLineAdjacent** is used, adjacent blocks of assembler code generated for an HLL line are concentrated.

If the option **SingleLine** is used, all blocks of assembler code generated for an HLL line are concentrated.

The object file (e.g. ELF file) does not contain the source code. It only contains the paths from which the source code can be loaded. The source code paths need to be adjusted if the build host environment differs from the debug host environment.

Option to adjust the debug paths for the source files	
PATH	If the source files are not found with the paths provided by the object file, additional direct directories can be given by this option.The option can be used more than once to include more directories into the search path. The command **sYmbol.SourcePATH** can be used to define more and permanent search directories.
SOURCEPATH	Define a new base directory for the source files. This replaces the current working directory that is taken by default if the source files are not find under the paths provided by the object file.
StripPATH	The file name is extracted from the source paths given in the object file.

Option to adjust the debug paths for the source files	
StripPART	Parts of the file paths provided by the object file are removed. The option takes either a *<number>* or a *<string>* as parameter. *<number>* defines how many parts of the path are removed. *<string>* is searched in the path provided by the object file. Everything until *<string>* is removed from the source path. This allow to specify a new base directory for a complete file tree by using the command **sYmbol.SourcePATH.SetBaseDir**.
StripBeforePART	Strips the file path up to, but excluding the specified *<pattern>*. For an example, see below.
LowerPATH	The file name is converted to lower-case characters.

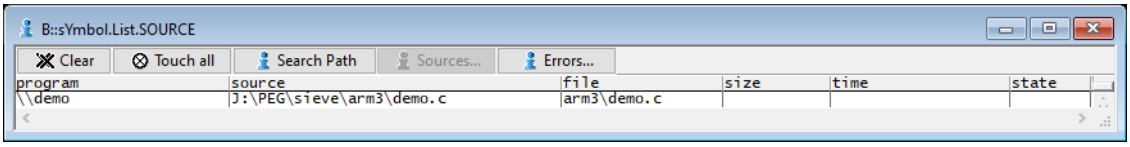
Examples for the options PATH, StripPART, and SOURCEPATH

[Back to PATH] [Back to StripPART] [Back to SOURCEPATH]

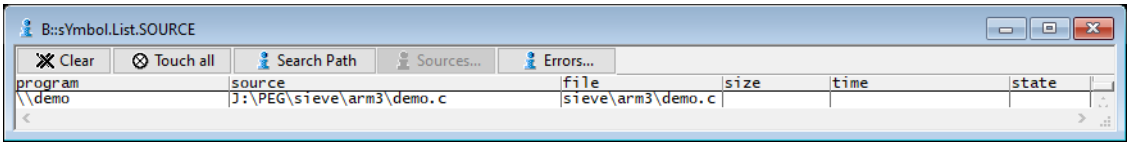
```
Data.LOAD.Elf demo.axf /PATH ~~/demo/quickstartboard/demo_ext
```

```
Data.LOAD.Elf demo.axf /StripPART 4. /SOURCEPATH ~~/demo/hardware/imx53
```

```
Data.LOAD.Elf demo.axf /StripPART 3.
```



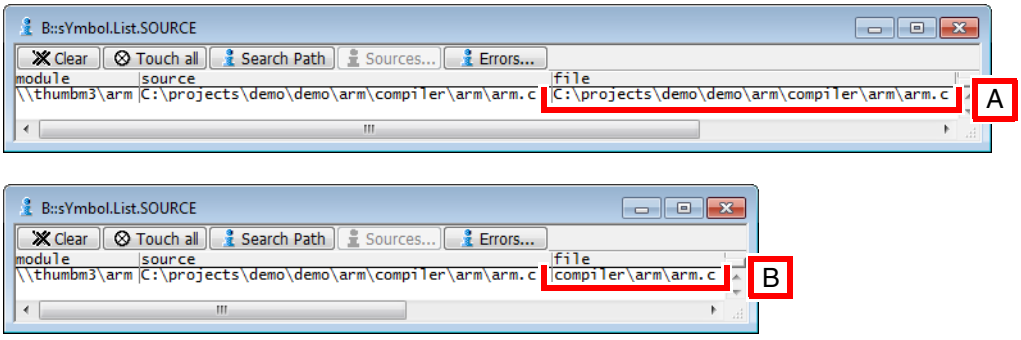

```
Data.LOAD.Elf demo.axf /StripPART "PEG"
```



Example for the option StripBeforePART

[Back]

```
;strip the path up to and excluding the string starting with "co"
Data.LOAD.Elf ~~/demo/arm/compiler/arm/thumbm3.axf /StripBeforePART "co"
sYmbol.List.SOURCE
```

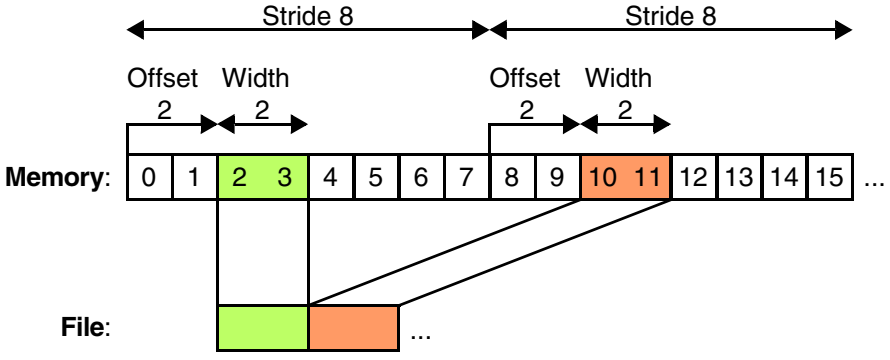


A Without **StripBeforePART**

B Path stripped with **StripBeforePART** "co"

BSPLIT: illustration of <stride>, <offset>, and <width>

[Back]



Example for the option GTLDMALOAD <offset>

[Back]

```
;configure the DMA transactor interface
SYStem.GTL.DMANAME "DMA0"

;connect to emulation system
SYStem.GTL.CONNECT

;load the elf file by using DMA0 starting with offset 0x1000
Data.LOAD.Elf "demo.axf" /GTLDMALOAD 0x1000
```

See also

- Data.LOAD.AIF
- Data.LOAD.AOUT
- Data.LOAD.ASAP2
- Data.LOAD.Ascii
- Data.LOAD.AsciiDump
- Data.LOAD.AsciiHex
- Data.LOAD.AsciiOct
- Data.LOAD.AVocet
- Data.LOAD.BDX
- Data.LOAD.Binary
- Data.LOAD.BounD
- Data.LOAD.CCSDAT
- Data.LOAD.CDB
- Data.LOAD.COFF
- Data.LOAD.ColonHex
- Data.LOAD.COMFOR
- Data.LOAD.CORE
- Data.LOAD.COSMIC
- Data.LOAD.CrashDump
- Data.LOAD.DAB
- Data.LOAD.DBX
- Data.LOAD.Elf
- Data.LOAD.ESTFB
- Data.LOAD.eXe
- Data.LOAD.FIASCO
- Data.LOAD.HiCross
- Data.LOAD.HiTech
- Data.LOAD.HP
- Data.LOAD.ICoff
- Data.LOAD.Ieee
- Data.LOAD.IntelHex
- Data.LOAD.LDR
- Data.LOAD.MachO
- Data.LOAD.MAP
- Data.LOAD.MCDS
- Data.LOAD.MCoff
- Data.LOAD.OAT
- Data.LOAD.Omf
- Data.LOAD.Omf2
- Data.LOAD.OriginHex
- Data.LOAD.PureHex
- Data.LOAD.REAL
- Data.LOAD.ROF
- Data.LOAD.S1record
- Data.LOAD.S2record
- Data.LOAD.S3record
- Data.LOAD.S4record
- Data.LOAD.SAUF
- Data.LOAD.SDS
- Data.LOAD.SPARSE
- Data.LOAD.sYm
- Data.LOAD.SysRof
- Data.LOAD.TEK
- Data.LOAD.TekHex
- Data.LOAD.Ubrof
- Data.LOAD.VersaDos
- Data.LOAD.XCoff
- List
- ADDRESS.isDATA()

▲ 'Release Information' in 'Legacy Release History'
▲ 'Load the Application Program' in 'Training Source Level Debugging'

The following **Data.LOAD.*** commands are format-specific. No automatic detection is performed. All generic options documented for **Data.LOAD.auto** are also available for the format-specific commands. The options documented below are only available for the format-specific commands, not for the generic **Data.LOAD.auto**.

Data.LOAD.AIF — Load Arm image file

Format:	**Data.LOAD.AIF** *<file>* [*<class>*] [**/***<option>*]
<option>:	**Puzzled** **AnySym** **PACK** **RAMINIT** *<generic_load_option>*

Loads a file in the AIF format (Arm Image Format). The debugging information must be in ARMSD format.

Puzzled	If the compiler rearranges the source lines, i.e. the lines will be no longer linear growing, this option must be used.
AnySym	Loads also special symbols that are otherwise suppressed.
PACK	Saves memory space by removing redundant type information. Standard types (e.g. char/long) are assumed to be equal in all modules. Types with the same definition can share the same memory space.
RAMINIT	Loads the data sections at its final position in RAM and fills the BSS section with zeros. Otherwise the data section will be loaded immediately after the code section and the BSS section remains unchanged.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format:	**Data.LOAD.AOUT** *<file>* [*<class>*] [**/***<option>*]
<option>:	*<generic_load_option>*

Loads a file in BSO/Tasking A.OUT format.

<option> For a description of the generic options, click **<generic_load_option>**.

NOTE: This is not the a.out format of the GNU compiler (see **Data.LOAD.DBX** for this format).

See also

- Data.LOAD

Data.LOAD.ASAP2 Load ASAP2 file

Format:	**Data.LOAD.ASAP2** *<file>* [**/***<option>*]
<option>:	*<generic_load_option>*

Loads a file in ASAP2 format.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD
- sYmbol.AddInfo.LOADASAP2
- ▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.LOAD.Ascii** *<file>* *<address>* | *<range>* [*/<option>*]
<option>:	**SKIP** *<offset>* *<generic_load_option>*

Loads a pure data file in word-oriented Ascii file format.

SKIP *<offset>*	If the option **/SKIP** *<offset>* is specified, the first *<offset>* bytes of the file are omitted.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Data.LOAD.AsciiDump Load ASCII file generated from Data.dump window

Format:	**Data.LOAD.AsciiDump** *<file>* [*<address>* | *<range>*] [*/<option>*]
<option>:	**OFFSET** *<value>* *<generic_load_option>*

Loads an Ascii file that was generated from the contents of a **Data.dump** window. To generate the Ascii file, use the command **PRinTer.FILE** and the pre-command **WinPrint**.

OFFSET *<value>*	Offset value in relation to the start *<address>*.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format: **Data.LOAD.AsciiHex** *<file>* [**/***<option>*]
Data.LOAD.AsciiHexA *<file>* [**/***<option>*]
Data.LOAD.AsciiHexB *<file>* [**/***<option>*]
Data.LOAD.AsciiHexC *<file>* [**/***<option>*]
Data.LOAD.AsciiHexP *<file>* [**/***<option>*]
Data.LOAD.AsciiHexS *<file>* [**/***<option>*]

<option>: **OFFSET** *<offset>*

Loads a file in a simple Ascii file format. Refer to **Data.SAVE.AsciiHex** for a description of the file formats.

See also

- Data.LOAD

Data.LOAD.AsciiOct Load octal file

Format: **Data.LOAD.AsciiOct** *<file>* [**/***<option>*]
Data.LOAD.AsciiOctA *<file>* [**/***<option>*]
Data.LOAD.AsciiOctP *<file>* [**/***<option>*]
Data.LOAD.AsciiOctS *<file>* [**/***<option>*]

<option>: **OFFSET** *<offset>*

Loads a file in a simple Ascii file format. Refer to **Data.SAVE.AsciiOct** for a description of the file formats.

See also

- Data.LOAD

Format:	**Data.LOAD.AVocet** *<file>* [*<class>*] [**/***<option>*]
<option>:	**NOHEX** *<generic_load_option>*

Loads a file in Avocet format. Without option the command will load the hex and sym files.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Data.LOAD.BDX Load BDX file

Format:	**Data.LOAD.BDX** *<file> <address>* | *<range>* [**/***<option>*]
<option>:	*<generic_load_option>*

Loads a file in WindRiver visionICE/visionPROBE Binary Download Format (BDX).

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

[Examples]

Format:	**Data.LOAD.Binary** *<file>* *<address>* | *<range>* [**/***<option>*]
<option>:	**SKIP** *<offset>* **UNZIP** *<generic_load_option>*

Loads a plain binary file.

<address>	If *<address>* is specified, the complete file will be loaded to the target *<address>*.
<range>	If *<range>* is specified, the file will be loaded to the range start address until the end of the range, or the end of the file.
SKIP *<offset>*	If the option **/SKIP** *<offset>* is specified, the first *<offset>* bytes of the file are omitted.
UNZIP	Unpacks files compressed with the **ZIP** option of a TRACE32 command, or files compressed by an external tool that uses the gzip archive format.
<option>	For a description of the generic options, click **<generic_load_option>**.

Example 1 - Patch a File

If you need to patch binary files, an elegant and fast way is to use the TRACE32 virtual memory (VM:). The following example shows how the file contents are loaded to and modified in the virtual memory of TRACE32. The result is then saved back to the original file.

```
;Load the binary file to the virtual memory starting at address VM:0
Data.LOAD.Binary "myfile.bin"   VM:0

;Display the virtual memory contents starting at address VM:0
Data.dump    VM:0

;Return the virtual memory content for the specified address
PRINT "0x"   %Hex   Data.Byte(VM:0x04)

;Modify the virtual memory
Data.Set  VM:0x04   0x42

;Save a range of the virtual memory back to the binary file
Data.SAVE.Binary "myfile.bin" VM:0--0x4F
```

Example 2 - Load Directly into RAM

This script shows how to directly load a file into RAM.

Prerequisite: A target board with a boot loader; this example is based on the U-Boot bootloader. Loading required files directly into RAM is a time saver because loading from flash is bypassed. This approach is useful, for example, if you want to quickly test different versions of a kernel.

```
LOCAL &base_path
&base_path="path/to/kernelsources"

SYStem.Mode.Up
Go

WAIT 2.s ;Wait until the boot loader has initialized the target board

TERM.OUT " "    ;Hit any key to stop autoboot and thus
                ;bypass loading from flash
Break           ;Halt the whole system (U-Boot is waiting
                ;for terminal commands)

;1) Load kernel image to RAM address 0x1020000
Data.LOAD.Binary   "&base_path/Linux/uImage" 0x1020000

;2) Load ramdisk to RAM address 0x2300000
Data.LOAD.Binary   "&base_path/Linux/rootfs.ext2.gz.uboot" 0x2300000

;3) Load device tree blob (DTB) to RAM address 0x1800000
Data.LOAD.Binary   "&base_path/Linux/p4080ds.dtb" 0x1800000

;Instruct TRACE32 to load ONLY the debug symbols of the kernel
Data.LOAD "&base_path/vmlinux" /NoReg /NoCODE /StripPART 5.
              /SOURCEPATH &base_path/Linux/Kernel_sources/linux-2.6.34.6

GO ;Resume waiting of U-Boot for terminal commands

;Instruct U-Boot to boot from the RAM addresses to which 1), 2), 3) have
;been loaded. "10." is the ASCII code for LF.
TERM.OUT "bootm 0x1020000 0x2300000 0x1800000" 10.
```

Example 3 - Load a Flash Image

In this script, a flash image is loaded into the FLASH of a target board.

```
//Target-specific code and code for the debugger, e.g. to declare the
//flash layout to the debugger

;Erase the whole flash
FLASH.Erase ALL

;Load image and program it into flash:
;1) Activate all FLASHs for programming
FLASH.Program ALL

;2) Load binary file
Data.LOAD.Binary flash_img.bin D:0xfe000000 /Long

;3) Deactivate FLASH programming
FLASH.Program off

;4) Compare the contents of the FLASH with the file contents
;   The comparison stops after the first difference
Data.LOAD.Binary flash_img.bin D:0xfe000000 /ComPare
```

See also

■ Data.LOAD ■ Data.SAVE.Binary

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.LOAD.BounD** *<file>* [*<access>* [**/***<option>*]

<option>: **IEEE**
MFFP
68881
OLD
Puzzled
<generic_load_option>

The floating-point format is set by means of the IEEE and MFFP options. The compiler options -VDB and -VPOST=NONE should be used.

IEEE, **MFFP**, **68881** Selects the floating-point format used by the compiler.

OLD Load a file from an old complier version. Use this option, if source lines at the start of a function are not set correctly.

Puzzled If the compiler rearranges the source lines, i.e. the lines will be no longer linear growing, this option must be used.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

■ Data.LOAD

Data.LOAD.CCSDAT Load CCSDAT file

Format: **Data.LOAD.CCSDAT** <file> [**/***<option>*]

<option>: *<generic_load_option>*

Loads a file in CSSDAT file format.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

■ Data.LOAD

▲ 'Release Information' in 'Legacy Release History'

[Examples]

Format: **Data.LOAD.CDB** *<file>* [*<class>*] [**/***<option>*]

<option>: **IntelHexFile** *<binary_file>*
NoIntelHexFile
WarningsAll
WarningsNo
<generic_load_option>

Loads debug information and binary code from SDCC-proprietary (Small Device C Compiler) file format called CDB. The file format description is available from SDCC / SourceForge / Free Software Foundation. The debug information and the binary code are saved in two separate files. The load command tries to find the corresponding file and loads debug information and code automatically together (see options to avoid this behavior). The binary part is stored in IntelHex-Format and can also be loaded separately.

IntelHexFile Define a dedicated Intel Hex file, which contains the binary code information. The option **/NoIntelHexFile** will be ignored. If the file does not exist, an error message will appear.
The default is to search the binary file automatically and announce an error message, if no file is found.

NoIntelHexFile No additional file (binary file) will be searched and loaded. Only the defined file will be processed.

WarningsAll All applicable warnings will be display in the **AREA** window. By default a set of warnings will be ignored, which will not lead to a reduced debug capability.

WarningsNo No warnings will be display. All warnings are internally ignored.

<option> For a description of the generic options, click **<generic_load_option>**.

Examples:

```
; Example for loading binary and symbol information separately

Data.LOAD.IH a.ihx /NosYmbol          ; Load binary only

Data.LOAD.CDB a.cdb                   ; Load symbol information only
/NoIntelHexFile

; Example for loading symbols and binary implicitly

Data.LOAD.CDB a.cdb                   ; binary must be named a.ihx

; Example for loading symbols (*.cdb) and binary (*.ihx)
; with one command (explicit)

Data.LOAD.CDB a.cdb /IntelHexFile othername.ihx
```

See also

- Data.LOAD

Format: **Data.LOAD.COFF** *<file>* [*<class>*] [**/***<option>*]

<option>:
FPU
MCS2 | **ICC** | **MOTC11** | **GHILLS** | **GNUCPP**
INT16
SHORT8
ALLLINE
Puzzled
AnySym
CFRONT
GlobTypes
LOGLOAD
<generic_load_option>

Loads a file in the UNIX-COFF format (Common Object File Format). The file format is described in all UNIX manuals. For some processors the command also supports debug information in STABS format.

MCS2 Should be used when loading a file generated by the MCS2-Modula compiler.

MOTC11 Should be used when loading a file generated by the Motorola cc11 compiler.

ICC Should be used when loading a file generated by the Intermetrics compiler.

GHILLS Should be used when loading a file generated by the Greenhills compiler.

GNUCPP Should be used when loading a file generated by the GNU C++ compiler.

ICC Should be used when loading a file generated by the Intermetrics compiler.

CFRONT Should be used when loading a file precompiled by CFRONT.

FPU Indicates the debugger that the code for an FPU was generated by the compiler.

Puzzled If the compiler rearranges the source lines, i.e. the lines will be no longer linear growing, this option must be used.

INT16 Specifies the size of integers to 16 bits.

SHORT8 Specifies the size of shorts to 8 bits.

ALLLINE	Loads HLL source lines in all sections. As a default only lines in the executable section are loaded.
AnySym	Loads also special symbols that are otherwise suppressed.
GlobTypes	Must be set when the debug information is shared across different modules. If the option is required, but not set, the loader will generate an error message requesting for the option.
LOGLOAD	Load using logical addresses contained in the COFF file.
ASMFUNC Ceva-X, TeakLite	Creates extra information for assembler functions.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD
- ▲ 'Release Information' in 'Legacy Release History'

Data.LOAD.ColonHex — Load colon hex file

Format:	**Data.LOAD.ColonHex** *<file>* [*/<option>*]
<option>:	**OFFSET | LineWidth** *<generic_load_option>*

Loads a colon hex file format with ":" as separator.

See also

- Data.LOAD

Format:	**Data.LOAD.COMFOR** *<file>* [**/***<option>*]
<option>:	**PHANTOM** *<generic_load_option>*

The **PHANTOM** option loads also phantom (out-of-sequence) line numbers.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format: **Data.LOAD.CORE** *<file>* [**/***<option>*]

<option>: *<generic_load_option>*

Loads a Linux core dump file into the TRACE32 Instruction Set Simulator. The object file has to be loaded before loading the core file.

<option> For a description of the generic options, click **<generic_load_option>**.

Example:

```
Data.LOAD.Elf object.elf              ;Load the object file

Data.LOAD.CORE corefile /NoClear      ;Load the core dump file
```

See also

■ Data.LOAD

▲ 'Release Information' in 'Legacy Release History'

Data.LOAD.COSMIC Load COSMIC file

Format: **Data.LOAD.COSMIC** *<file>* [*<class>*] [**/***<option>*]

<option>: **INT16**
SCHAR | **SPREC**
MODD | **MODP** | **MODF** (only 68HC16)
IEEE
MMU
REV
ADDBANK
LOGLOAD
<generic_load_option>

Default: MultiLine.

The loader is implemented for 68K, 32K, 68HC11 and 68HC16 families.

INT16	Uses 16 bit integers, instead of 32 bit (only 68K).
IEEE	Uses IEEE floating point format instead of processor specific format.
SCHAR	Char type is signed, instead of unsigned.
MODD, MODP, MODF	Memory models for 68HC16 compiler.
SPREC	Use single precision floating point only.
REV	Reverse bit fields. Must be set when the compiler option was set.
ADDBANK	Add information about the bank number to the module names. Must be used if modules with the same name are duplicated in different banks.
LOGLOA	Loads to logical addresses instead of physical addresses. Only relevant for banked systems.
MMU	Loads information and translation tables for on-chip MMU.
<option>	For a description of the generic options, click **<generic_load_option>**.

NOTE:	If loading a file for the 68HC11K4 processor in banked configuration the MMU command and banking registers of the CPU must be prepared before loading (see emulation probe manual for 68HC11).

See also

- Data.LOAD

Data.LOAD.CrashDump — Load MS Windows Crash Dump file

Format:	**Data.LOAD.CrashDump** <file> [**/***<option>*]
<option>:	*<generic_load_option>*

Loads a Microsoft Windows Crash Dump or Minidump file into the TRACE32 Instruction Set Simulator. The command supports the Crash Dump files of types “Kernel memory Dump” and “Complete memory dump”.

For a complete analysis of the MS Crash Dump, the Windows awareness needs to be used in addition to this command. This helps to retrieve and autoload the Windows kernel debug symbols and sets the context of all the CPUs that are available in the Crash Dump.

For more details about the Windows awareness extension and the MS Crash Dump analysis, please refer to **"OS Awareness Manual Windows Standard"** (rtos_windows.pdf).

<option> For a description of the generic options, click **<generic_load_option>**.

Example:

```
Data.LOAD.CrashDump memory.dmp                    ;Load the crash dump file
```

See also

- Data.LOAD

▲ 'Release Information' in 'Legacy Release History'

Data.LOAD.DAB — Load DAB file

Format: **Data.LOAD.DAB** <file> [/*<option>*]

<option>: *<generic_load_option>*

Loads a file in DAB file format.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.LOAD.DBX** *<file> <code> <data>* [*<sym>*] [**/***<option>*]
<option>:	**CPP** **AnySym** **CFRONT** **GHILLS** **LIMITED** *<generic_load_option>*

Loads a file in DBX-format (sometimes called 'a.out' or Berkeley-Unix file format). The format is used by SUN native compilers and GNU compilers. As the standard format doesn't include any start address the first addresses for code and optionally data must be defined. The third address argument can be used to relocate the symbols when a relocatable program is loaded.

CPP	Must be set when debugging C++ applications.
AnySym	Loads also any special labels (defining file names etc.) which are usually suppressed by the loader.
CFRONT	Load C++ files converted by the AT&T cfront preprocessor.
GHILLS	Load file from Greenhills compiler. For C++ files the **CFRONT** switch is also required.
LIMITED	Doesn't load any type information. Loads the code and the source information only.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format: **Data.LOAD.Elf** *<file>* [*<memory_class>* | *<offset>* | *<range>*] [**/***<option>*]

<option>:
FPU | **NOFPU**
AnySym | **ZeroSym**
NOMERGE | **NOEXTERNALS**
STRIPPREFIX *<prefix>* | **STRIPADANAMES**
PACK
CODESEC | **CODEZERO** | **CODEPROG**
LOGLOAD | **PHYSLOAD**
LOADSEC *<section_name>* | **LOADSEC !***<section_name>*
GlobTypes | **NoGlobTypes**
GHS | **GNU** | **IAR** | **METAWARE** | **MetroWerks** | **MRI** | **WRS** | **CFRONT**
PROTOTYPES
STARTTHUMB
STABS | **DWARF** | **DWARF2**
NOLINES | **NOCONST** | **NOMETHODS** | **NOCALLINFO**
CPP | **GNUCPP**
IA64
DeBuGInfoFILE *<file2>*

<option>: (cont.)
CHILL | **PASCAL**
ALTBITFIELDS
ALTRELOC2
ENUMCONSTS
ABSLINES
ABSLIFETIMES
ForceLines
FUNClines | **IgnoreFUNCLines**
RelPATH | **RelPATH2**
ModulePATH
ChainedStab | **ChainedStab4** | **ChainedAbbrev** | **ChainedLines**
BUGFIX4
RELOC *<section_name>* **AT** *<address>*
RELOC *<section_name>* **AFTER** *<section_name_other>*
RELOC *<section_name>* **LIKE** *<section_name_other>*
RELOCTYPE *<type>*
RELOCSTRIPPED
LOCATEAT *<address>*
RemoveModuleInSection *<section_name>*
RemoveModuleAfterSection *<section_name>*
OVERLAY
NoFILEHDR | **NoPHDRS**
NODEBUG
NODEBUGFRAME | **NOEHFRAME**
NODOUBLE | **NOARGCOERCE**
NOLINKAGENAME

<option>: (cont.)	**IgnoreModuleRange** **DWFILE** | **DWOFILES** **PREFIX** **FILTERBYCORE** *<architecture_specific_load_option>* *<generic_load_option>*
<architecture_specific_load_option>:	**RVCT** | **ABI** *<vers>* (only Arm) **ALTDOUBLE** | **NOALTDOUBLE** (only Arm) **ALTTHUMBSYMBOLS** (only Arm) **REAL** | **REAL32** | **SMALLREAL** | **SMALLREAL32** (only Intel x86) **ALTRELOC** (only M68K and ColdFire) **REV** | **MMU** (only HC11 and HC12) **NMF** *<address>* | **DynamicNMF** (only MMDSP) **LARGE** (only Intel DSP56K)

Load a file in the ELF format. The file format description is available from UNIX International. The debug information can be if DWARF1 or DWARF2 format. For some processors STABS debug information is also supported.

When *<file>* does not contain debug information (stripped file) the debuglink or build-id may be used to automatically search for the debug information in a separate file.

The search strategy is in descending order:

- *<base>*/.build-id/*<xx>*/*<yy...yy>*.debug
  - *<xx>* are the first two hex characters of the build-id
  - *<yy...yy>* are the remaining hex characters of the build-id
  - *<base>* is set by **sYmbol.SourcePATH.SetBaseDir**
- *<path of file>*/*<debuglink>*
  - debuglink is the name stored in the .gnu_debuglink section of the <file>
- *<path of file>*/.debug/*<debuglink>*
  - debuglink is the name stored in the .gnu_debuglink section of the *<file>*
- *<base>*/*<root-path of file>*/*<debuglink>*
  - *<debuglink>* is the name stored in the .gnu_debuglink section of the *<file>*
  - *<root-path of file>* is automatically detected by removing the leading folders one after each other

  Example:

```
<file> = /home/user/sysroot/lib/libc.so

<root-path of file> =
[home/user/sysroot/lib,user/sysroot/lib,sysroot/lib,lib]
```

- *<base>* is set by **sYmbol.SourcePATH.SetBaseDir**

- *<pwd>|<debuglink>*
  - *<debuglink>* is the name stored in the .gnu_debuglink section of the *<file>*
  - *<pwd>* is the present working directory (e.g. `CD /home/user`)

Please refer to **Examples for split debug information** for details.

Options:

<option>	For a description of the generic options, click **<generic_load_option>**.
ABSLIFETIMES	Special option - activates workarounds for compiler/linker issues.
ABSLINES	Special option - activates workarounds for compiler/linker issues.
ALTBITFIELDS	This option might solve problems with regards to the display of bitfields.
ALTRELOC2	Supports special relocation variant.
AnySym	Loads all symbols generated by the compiler (defining file names, local labels etc.) which are usually suppressed by the loader.
BUGFIX4	Option for an alternative interpretation of the DWARF Line Number Information (section ".debug_line"). With this option the offset to a Line Number Program gets calculated from unit_length (==total_length) and header_length (==prologue_length) of the Line Number Program Header (==Statement Program Prologue). Without this option TRACE32 assumes that a Line Number Program starts directly after its header. Try this option only if the Line Number Information in the **sYmbol.List.LINE** window seems to be wrong.
CFRONT	Enable workarounds for Cfront C++ compiler.
PROTOTYPES	Force all functions to include prototype information.
ChainedAbbrev	Special option - activates workarounds for compiler/linker issues.
ChainedLines	Enables workaround for not relocated line tables.
ChainedStab	Special option - activates workarounds for compiler/linker issues.
ChainedStab4	Special option - activates workarounds for compiler/linker issues.
CHILL	Set this option if your program was coded in CHILL (CCITT High Level Language).
CODEPROG	Forces loading from program table of the ELF file.

CODESEC	Normally the code download is done by using the program table of the ELF file. This option selects the Section table for code download. Some linkers produce a buggy Program table.
CODEZERO	If a program header of an ELF file indicates that it is indented for more memory than within the ELF file (p_memsz > p_filesz) than this additional memory should be set to zero. This is usually done by the start-up code of the target application. However with option CODEZERO, the debugger fills this memory with zeroes.
CPP	Must be set when loading an ELF file with symbol information in STABS format for C++.
DeBuGInfoFILE *<file2>*	Loads the debug information from <file2> in case the main file does not contain any debug information. Specifying this option disables the automatic search strategy in TRACE32. If *<file2>* is an empty string, the automatic search strategy is disabled and only the stripped main file is loaded.
DWARF **DWARF2**	Forces debugger to load only debug information in DWARF format (ignoring debug information in STABS format). The default is to load all available debug information independently of the formats.
DWOFILES	Loads the debug information from DWARF Object file.
DWPFILE	Loads the debug information from Dwarf Package file.
FILTERBYCORE	Just loads the debug information for certain cores (when the ELF file includes multiple cores).
ForceLines	Force loading of source code lines which are not “recommended breakpoint locations” according to the DWARF line table. The line table tells the debugger to which target address a line of source code is associated
FPU, NOFPU	Indicates the debugger that the code for FPU or without FPU was generated by the compiler.
FUNClines	Force a source line at first address of function. [build no. 46143 - DVD 08/2013]
IgnoreFUNCLines	Ignore line number debug information of function declaration.
GHS	Enable workarounds for GreenHills compiler.
GlobTypes	Must be set when the debug information is shared across different modules. If the option is required, but not set, the loader will generate an error message requesting for the option.
GNU	Needs to be set for some older GNU compilers.

GNUCPP	Same as **GNU** and **CPP**. Enables GNU specific workarounds for loading of programs written in C++
IA64	Enable IA-64 style symbol demangling of programs written in C++ . This is enabled by default for several modern compiler e.g. GCC vers. 3 and higher, DIAB vers. 5.9 and higher, TASKING VX-toolset, ...
IAR	Enable workarounds for IAR compiler.
IgnoreModul-eRange	Ignores address range information for modules from the DWARF debug.
LOADSEC *<section_name>*	Loads the code of a single ELF section. **Example**: `Data.LOAD.Elf <file> /LOADSEC <section_name>` Suppresses the code of a single ELF section. **Example**: `Data.LOAD.Elf <file> /LOADSEC !<section_name>`
LOGLOAD	Takes the logical address (p_vaddr) of the program table to load the code (instead of the physical address).
METAWARE	Enable workarounds for Synopsys' MetaWare® C/C++ Compiler
MetroWerks	Enable workarounds for MetroWerks Compiler
ModulePATH	Keeps the path name information in the module names. By default the module names are reduced to the pure source name (without path and file extension) whenever possible. The option has no effects on the source file names or directories.
MRI	Must be set for the Microtec compiler.
NOARGCOERCE	Suppresses the load of argument coercing.
NOCALLINFO	Suppresses loaded call debug information (loaded by default).
NOCONST	Suppresses the load of "const" variables.
NODEBUG	Ignore DWARF and STABS debug information.
NODEBUGFRAME	Suppresses the load of debug frame information (“.debug_frame”). Loads only “.eh_frame” when debug info is in separate ELF/DWARF file.
NODOUBLE	Allows using just single precision float.
NOEHFRAME	Suppresses the load of “.eh_frame” section.
NOEXTERNALS	Ignores declaration of external variables in DWARF debug information.

NoFILEHDR	Suppresses the loading of the ELF Header to target memory. Most ELF files do not load the ELF Header to the target memory by default anyway. The loading of the ELF Header should also be configurable via the linker script (scatter file). E.g.: For GCC see 'PHDRS' statement of PHDRS command in GNU Linker Scripts. [build no. 46001 - DVD 08/2013]
NoGlobTypes	Can be set when there is no shared debug information in a file format where the loader expects them (e.g. for Arm).
NOLINES	Suppresses the load of the line table. The line table tells the debugger to which target address a line of source code is associated.
NOLINKAGENAME	Suppresses using the linkage name from the debug info (DW_AT_linkage_name).
NOMERGE	Symbol information of the different formats (DWARF/STABS) are not merged together. The default is to merge the symbol information.
NOMETHODS	Suppresses loading of C++ method debug information.
NoPHDRS	Suppresses the loading of the ELF Program Header Table to target memory. Most ELF files do not load the ELF Program Header Table to the target memory by default anyway. The loading of the ELF Program Header Table should also be configurable via the linker script (scatter file). E.g.: For GCC see 'PHDRS' statement of PHDRS command in GNU Linker Scripts. [build no. 46001 - DVD 08/2013]
OVERLAY	Set when loading • an ELF file containing overlay code sections (declared as such be the debug information in the file) or • an ELF file containing plain code sections (not marked in any special way in the debug information) that overlay code sections of other ELF files (that were loaded or will be loaded). The option makes the ELF-Loaded consider existing relocation sections and the details from the table of declared overlay sections (**sYmbol.OVERLAY.List**) to load the symbols of each overlaying section to a separate virtual memory segment (Each address is virtually extended by an “overlay ID”). If your ELF file does not contain relocation information you have to declare the overlaying sections and source files using **sYmbol.OVERLAY.Create** before loading the ELF file (“File-based Code Overlay Support”). In this case the load options **/NOFRAME** and **/NoClear** are also recommended.

PACK	Saves memory space by removing redundant type information. Standard types (e.g. char/long) are assumed to be equal in all modules. Types with the same definition can share the same memory space.
PASCAL	Manually set programming language to Pascal.
PHYSLOAD	Use the physical address (p_paddr) of the program table to load the program.
PREFIX	Defines the prefix character. Same as **sYmbol.PREFIX** command.
RELOC *<secname>* **AT** *<address>* **RELOC** *<secname>* **AFTER** *<secname_other>* **RELOC** *<secname>* **LIKE** *<secname_other>*	Relocates code/symbols of the specified section to the specified logical address or after the specified section.
RELOCSTRIPPED	For stripped ELF files only (separate debug info file). Relocates the information in debug file according to the information from the stripped ELF file.
RELOCTYPE *<type>*	Relocates sections based on the OS Awareness.

RelPATH	Source files can be compiled with a full or a relative path. **Example**:
	`/home/irohloff/my_project/obj > gcc -c -g ../my_file.c`
	`/home/irohloff/my_project/obj > gcc -c -g /home/irohloff/my_project/my_file.c`
	If the source file was compiled with a relative path, the compilation path is also stored in the *.elf file. TRACE32 combines the compilation path with the relative path to the source file path by default.
	The option /**RelPATH** advises TRACE32 use only the relative path as source file path.
	The option can be combined with other source search path commands to adjust the search path for the debugger in case the source files have been moved.

RelPATH2	The option /RelPATH2 strips away the path information from the DWARF2 line number information. This is usually the directory path to the source file given in the compiler command line.
RemoveModuleAfterSection *<sename>*	Removes dummy modules after section.

RemoveModuleIn-Section *<secname>*	Removes dummy modules in section.
REV	Reverse bit fields. Must be set when the compiler option was set. (only HC11/HC12)
STABS	Forces debugger to load only debug information in STABS format (ignoring debug information in DWARF format). The default is to load all available debug information independently of the formats.
STARTTHUMB	Forces the start PC to thumb mode (as workaround for buggy ELF files).
STRIPADANAMES	Strips off parents from nested function names.
STRIPPREFIX *<prefix>*	Strip given string from the beginning of every ELF symbol. E.g. The symbol "`F_main`" becomes "`main`" with `/STRIPPREFIX "F_"`
WRS	Enable workarounds for WindRiver Diab compiler.
ZeroSym	By default modules linked to address 0x00 are not loaded. The option /**ZeroSym** advises the loader to also load all modules linked to address 0x00.

Architecture Specific Load Options:

Arm architecture:

ABI *<vers>*	Force Arm ABI version.
RVCT	Force IA-64 style symbol demangling of programs written in C++ .

Intel® x86 architecture:

REAL	Memory model (only used for x86 REAL-mode debug info in stabs format).
REAL32	Memory model (only used for x86 REAL-mode debug info in stabs format).
SMALLREAL	Memory model (only used for x86 REAL-mode debug info in stabs format).
SMALLREAL32	Memory model (only used for x86 REAL-mode debug info in stabs format).

M68K and Coldfire architecture:

ALTRELOC	Supports special relocation variant.

HC11 and HC12 architecture:

MMU	Loads information and translation tables for onchip MMUs
REV	Reverse bit fields. Must be set when the compiler option was set.

MMDSP architecture:

NMF *<address>*	NMF framework support.
DynamicNMF	Special option (not active architecture).

DSP56K architecture:

LARGE	Large memory model.

Examples

Example for *<memory_class>*:

```
; If loading the code to the target memory is not working, you can
; inspect the code by loading it to the virtual memory

Data.LOAD.Elf demo.elf VM:

Data.List VM:                          ; Display a source listing based on
                                       ; the code in the virtual memory

sYmbol.List.MAP                        ; Display the addresses to which
                                       ; the code/data was written
```

Example for *<offset>* for the TriCore:

```
; The program was linked for address 0x83000000, which is cached external
; memory, but FLASH programming is not working on cached memory.
; To solve this situation the program has to be programmed to 0xA3000000
; which is in not cached external memory

…

FLASH.Program ALL

Data.LOAD.Elf demo.elf 0x20000000     ; Add offset for loading

FLASH.Program off
```

Examples for the option **/RELOC**:

```
; relocate the code section of the file mymodul.o
; to the address 0x40000000
Data.LOAD.Elf mymodul.o /NoCODE /NoClear /RELOC .text AT 0x4000000

; relocate the const. section of the file mymodul.o
; after the code section
Data.LOAD.Elf mymodul.o /NoCODE /NoClear /RELOC .text AT 0x4000000 \
/RELOC .const AFTER .text

; relocate the const. section of the file mymodul.o
; the same delta like the code section
Data.LOAD.Elf mymodul.o /NoCODE /NoClear /RELOC .text AT 0x4000000 \
/RELOC .const LIKE .text
```

Example for *<range>*:

```
; The elf files contains program for the FLASH and data loaded to RAM,
; the data loaded to RAM might disturb the target-controlled FLASH
; programming.
; To solve this situation the code is loaded only to the specified
; address range.

…

FLASH.Program ALL

Data.Load.Elf demo.elf 0xa3000000++0x3fffff

FLASH.Program off
```

Example for *<offset> <range>*:

```
Data.LOAD.Elf demo.elf 0x1000000 0x13f9900++0xff

; Please be aware that the code is first moved by <offset> so the <range>
; has to be specified by using its new addresses
```

Example for loading the program to a virtual machine.

```
Data.LOAD.Elf ../FreeRTOS/FreeRTOS.elf N:3:::0 /NoClear /NoCODE
```

Examples for split debug information

Example 1: gnu-debuglink in same folder:

```
; $ gcc -g -o foo foo.c
; $ objcopy --only-keep-debug foo foo.debug ;
; $ strip -g foo
; $ objcopy --add-gnu-debuglink=foo.debug foo
Data.LOAD.Elf foo ; loads './foo' and './foo.debug'
```

Example 2: gnu-debuglink in .debug subfolder:

```
; $ gcc -g -o foo foo.c
; $ mkdir  .debug
; $ objcopy --only-keep-debug foo .debug/foo.debug
; $ strip -g foo
; $ objcopy --add-gnu-debuglink=foo.debug foo
Data.LOAD.Elf foo  ; loads ./foo and ./.debug/foo.debug
```

Example 3: gnu-debuglink in /usr/lib/debug subfolder:

```
; $ gcc -g -o /usr/local/bin/foo foo.c
; $ objcopy --only-keep-debug /usr/local/bin/foo  \
;                                 /usr/lib/debug/usr/local/bin/foo.debug
; $ strip -g /usr/local/bin/foo
; $ objcopy --add-gnu-debuglink=foo.debug /usr/local/bin/foo
sYmbol.SourcePATH.SetBaseDir /usr/lib/debug
Data.LOAD.Elf /usr/local/bin/foo
; loads /usr/local/bin/foo and /usr/lib/debug/usr/local/bin/foo.debug
```

Example 4: build-id in /usr/lib/debug:

```
; $ gcc -g -Wl,--build-id -o foo foo.c
; $ objcopy --only-keep-debug foo  \
;                            /usr/lib/debug/.buildid/<xx>/<yy..yy>.debug
; # where <xx> are the first two hex characters of the build-id
; # where <yy...yy> are the remaining hex characters of the build-id
; $ strip -g foo
sYmbol.SourcePATH.SetBaseDir /usr/lib/debug
Data.LOAD.Elf foo
; loads ./foo and /usr/lib/debug/.build-id/<xx>/<yy..yy>.debug
```

Example 5: explicitly set debug information file e.g. no gnu-debuglink, no build-id available:

```
; $ gcc -g -o foo foo.c
; $ objcopy --only-keep-debug foo foo.bar
; $ strip -g foo
Data.LOAD.Elf foo /DeBuGInfoFILE foo.bar ; loads ./foo and ./foo.bar
```

See also

■ Data.LOAD ■ SETUP.DropCoMmanD

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.LOAD.ESTFB** *<file>* [*<offset>* | *<range>*] [**/***<option>*]
<option>:	*<generic_load_option>*

Loads an EST flat binary file. An EST flat binary is a binary file with a 32 byte header which defines start and end address of the included binary data.

<offset>	If *<offset>* is specified, the load address is increased by *<offset>* bytes. Positive and negative offsets are possible.
<range>	If *<range>* is specified, only the parts within *<range>* are loaded.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Data.LOAD.eXe Load EXE file

Format:	**Data.LOAD.eXe** *<file>* [*<access>* | *<address>*] [**/***<option>*]
<option>:	**AnySym** **CPP** **ENUMCONSTS** **RELOC** *<section_name>* **AT** *<address>* **RELOC** *<section_name>* **AFTER** *<section_name>* **FPU** **LOCATEAT** *<address>* **NoMMU** (default, except for 8086, 80186 and 80286) *<generic_load_option>*

Default: ENUMCONSTS.

Loads files in EXE-format. The command accepts different formats for symbolic information. Plain MS-DOS EXE formats require a base address for the starting segment. Files from Paradigm Locate or PE-Files from Pharlap require no load address.

<option>	For a description of the options, see **Data.LOAD.Elf**.
<generic_load_option>	For a description, click **<generic_load_option>**.

The following formats are accepted:

Real Mode	**Debug-Format**	**Compiler**
DOS-EXE	CodeView 4	MSVC 16-bit edition, DOS File
WIN-EXE	CodeView 4	MSVC 16-bit, Windows Executable (only symbols)
DOS-EXE	CodeView 3	MS-C, Logitech Modula
DOS-EXE	Borland	Borland-C/C++ 2.x-3.x
PARADIGM-AXE	Borland	Borland-C/C++ 2.x-3.x and Paradigm Locater
Protected Mode		
PHARLAP-P3	CodeView 4	MSVC 32-bit edition and Pharlap Locater
Windows(CE)	CodeView 4/5	MSVC 32-bit edition
Windows(CE)	PECOFF	MSVC 32-bit edition
Windows(CE)	PDB	MSVC (x86,Arm,SH,PowerPC)
SymbianOS	STABS	GCC
Windows	PDB	MSVC (x64)

See also

■ Data.LOAD ■ SETUP.DropCoMmanD

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.LOAD.FIASCO** *<file>* [*<address>* | *<range>*] [**/***<option>*]
<option>:	**NoCRCcheck** **CRCcheck** *<generic_load_option>*

Default: NoCRCcheck.

Loads a data file of the FIASCO BB5 file format (*.fpsx).

<option>	For a description of the generic options, click **<generic_load_option>**.
NoCRCcheck	The check sums in the file are ignored.
CRCcheck	The check sums in the file are taken into account. If one or more errors are encountered in the check sums, a warning message is displayed in the TRACE32 message line. Detailed warning messages are printed to the **AREA** window.

See also

- Data.LOAD

Data.LOAD.HiCross Load HICROSS file

Format:	**Data.LOAD.HiCross** *<file>* [*<class>*] [**/***<option>*]
<option>:	**M2** **DBGTOASM** *<generic_load_option>*

Loads a file from Hiware Modula2 or C Cross Development System. The file name is the name of the absolute file, all other files are searched automatically.

<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format:	**Data.LOAD.HiTech** *<file>* [*<class>*] [**/***<option>*]
<option>:	**NOHEX** **NOSDB** **Puzzled** *<generic_load_option>*

Load a file in HI-TECH object format. The code is loaded in S-Record format. When the code is not in S-Record (Motorola) format, it must be loaded separate with the appropriate command and the symbol file can be loaded with the **/NOHEX** option.

NOHEX	Don't load code, load only symbol files.
NOSDB	Don't load the HLL-debugging information.
Puzzled	This option should be used, when the global optimizer of the compiler is activated.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format: **Data.LOAD.HP** *<file>* [*<class>* | *<offset>*] [*/<option>*]

<option>:
NOX
NOL
NOA
NoMMU
PACK
WARN
MODULES
<generic_load_option>

Loads a file in HP-64000 format. All three file types (.X/.L/.A) are loaded if existing. The command **sYmbol.LSTLOAD.HPASM** allows source debugging in assembler files. The optional address value defines an offset for the code of the program. This offset can be used to load a file into a different bank on banked 8-bit systems.

NOX	Doesn't load the absolute code file. The file name must be the name of the symbol file (.L).
NOL	Doesn't load any symbols.
NOA	Doesn't load local symbols from '.A' files.
NoMMU	Doesn't set up the MMU table. The MMU table is usually loaded with information from the sections of the file (64180, 80186,etc.).
PACK	Compress symbol information by saving the same labels in different modules only once.
MODULES	Takes the object file from a different location. Try this option if local symbols from modules are missing.
WARN	Issues a warning message when '.A' files are not found. The default is to silently ignore these files.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format:	**Data.LOAD.ICoff** *<file>* [*<class>*] [**/***<option>*]
<option>:	**FPU** **MOD** **Puzzled** *<generic_load_option>*

Loads files in Introl ICOFF format.

<option>	For a description of the generic options, click **<generic_load_option>**.
FPU	Tells the debugger that code for the FPU was generated by the compiler.
MOD	Adjusts the loader for INTROL MODULA2 files.
Puzzled	If the compiler rearranges the source lines, i.e. the lines are no longer linear growing, this option has to be used.

See also

- Data.LOAD

Format: **Data.LOAD.Ieee** *<file>* [*<address>* | *<access>*] [**/***<option>*]

<option>: **FPU** | **NOCLR** | **STandarD**
InLine | **NoInLine**
INT16
NoMATCH
MSECtion
NOFRAME
LIMITED
CFRONT
PREFIX *<char>*
ZP2 | **MCC3** | **C** | **ALSYS** | **XDADA** | **A5** (only 68K)
NoMMU (only x86)
LARGE (only C166)
<generic_load_option>

The access class can be used to select a different access class for the saving of code or for the symbols, e.g. the code can be saved directly in emulation memory (**E:**). The address parameter is used as an offset to the addresses in the IEEE file. It is only useful for loading different memory banks on a banked system.

<option> For a description of the generic options, click **<generic_load_option>**.

ALSYS Must be used when loading ALSYS IEEE files.

C Forces the language to 'C'. This option must be used, when the compiler generates a wrong 'compiler-id', i.e. the displayed language is PL/M or ADA.

CFRONT Load C++ files converted by a CFront preprocessor. NOTE: This option should not be used for Microtec C++ files.

FPU Tells the debugger that code for an FPU (Floating Point Unit) was generated by the compiler.

INT16 Specifies the size of integers to 16 bits.

LIMITED Doesn't load any type information. Loads the code and the source information only.

MCC3 Must be used when loading MCC68K 3.0 files.

MSECtion Assembler module sections are included in the section table. As a default the IEEE sections are included in the table. If the compiler generates the assembler module information, this information will be more exact.

NOCLR If the 'NOCLR' option was selected in the Microtec C compiler, this option has to be set in order to display enumeration types properly.

NOFRAME Ignore the stack frame information in the IEEE file. The stack frame information in the file is used for the **Frame.view** window only. Use this option, if your compiler doesn't produce the correct information. The TRACE32 tool try's then to analyze the function prolog code to get the stack frame.

NoInLine Suppresses the lines generated by the compiler when inline optimizing in activated (-Oi). The code generated by a call to an inlined function is then executed as one line.

NoMATCH If the loader detects externals with type information it tries to combine them with globals of the same name without type information. This matching process can be turned off with this option.

NoMMU Suppress the generation of the address translation table for the MMU command. This table is used to recreate logical addresses from physical addresses seen on the address bus (80x86).

STandarD If the 'STANDARD' option was selected in Microtec PAS68K, then the associated underscores can be removed at the time of loading by setting this option.

ZP2 Must be set, when the ZP2 option was used to compile the file.

See also

■ Data.LOAD

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.LOAD.IntelHex** *<file>* [*<addressrange>*] [**/***<option>*]
<option>:	**OFFSET** *<offset>* *<generic_load_option>*

The file is shifted (by the factor of the Offset) and loaded.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

■ Data.LOAD ■ Data.SAVE.IntelHex

▲ 'Release Information' in 'Legacy Release History'

Data.LOAD.LDR Load META-LDR file

Meta

Format:	**Data.LOAD.LDR** *<file>* [*<addressrange>*] [**/***<option>*]
<option>:	*<generic_load_option>*

Load LDR file for Meta architecture. Please refer to **"Meta Debugger"** (debugger_meta.pdf) for more information.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

■ Data.LOAD

[Examples]

Format: **Data.LOAD.MachO** *<file>* [*<class>*] [**/***<option>*]

<option>: **DebugFile** *<binary_file>*
NoDebugFile
DWARF | **STABS**
NOCONST
NOMERGE
UUID
IgnoreARCH
ARCHNumber *<value>*
IgnoreModuleRange
<generic_load_option>

Load a file in the Mach-O file format. The file format description is available from Apple Inc. The debug information can be in DWARF2 or STABS format. For some compiler (e.g. GCC) both formats are combined in one file. Binary and symbol information could be found separated in two files with an identical UUID. The load command tries to find silently a corresponding debug file and load its symbol information.

<option> For a description of the generic options, click **<generic_load_option>**.

ARCHNumber Loads the entry with the specified number of an universal binary (FAT). Counting starts from zero.
Default is to load the first matching architecture (target <-> Mach-O-file) of the universal binary.

DebugFile Define a dedicated file, which contains the symbol information. The option **/NoDebugFile** will be ignored. If the file does not exist, an error message will appear.
The default is to search the debug file automatically and does not announce anything, if no file is found. The UUIDs instead will always be checked between the two files.

DWARF, STABS Forces debugger to load only debug information in DWARF respectively STABS format. The default is to load all available debug information independently of the formats.

IgnoreARCH The architecture field of the Mach-O-file will be ignored and no warning will be emitted, if does not match.
If the Mach-O-file is an universal binary (FAT), the first entry (number=0) will be loaded regardless of its and the others architecture-codes.

IgnoreModuleRange Ignores address range information for modules from the DWARF debug.

NOCONST Suppresses the load of "const" variables.

NoDebugFile No additional file (debug file) will be searched. Only the defined file will be processed. No UUIDs will be compared. This allows to load even non-correspondent files.
The **NoDebugFile** option will be set implicitly by the generic load options **NoCODE** and **NosYmbol** (see example).

NOMERGE Symbol information of the different formats (DWARF/STABS) are not merged together. The default is to merge the symbol information.

UUID Only the universally unique identifier (UUID) of a Mach-O file is read, shown and saved. Target code, registers or symbols will not be changed. The function **MACHO.LASTUUID()** dispenses this UUID. After every Mach-O load command the UUID is saved and could be read via the function MACHO.LASTUUID().
If no UUID is found in a Mach-O file, MACHO.LASTUUID() will dispense "no UUID read" independently rather UUID option is set or not. But only with the UUID option set, a failure will be returned by the load command. This option should be used exclusively, because if used all other options are ignored.

Examples:

Example for loading binary and symbol information separately:

```
Data.LOAD.MachO a.out /NosYmbol       ; Load binary only

Data.LOAD.MachO sym.out /NoCODE       ; Load symbol information only

; Example for loading binary and symbols with one command

Data.LOAD.MachO a.out /DebugFile sym.out
```

Example for usage of **UUID** option:

```
PRINT MACHO.LASTUUID()                ; "no UUID read" will be displayed,
                                      ; because no Mach-O file was loaded

Data.LOAD.MachO a.out /NosYmbol       ; Load binary only

PRINT MACHO.LASTUUID()                ; UUID of a.out will be displayed
                                      ; in area window, like: "ba4718af-
                                      ; 884c-6b81-b7e8-5d771938ac83"

Data.LOAD.MachO sym.out /UUID         ; UUID of sym.out will be displayed
                                      ; in area window and could be
                                      ; compared with those of a.out.
                                      ; Nothing will be loaded.

Data.LOAD.MachO sym.out /NoCODE       ; If both are equal, load symbols
```

See also

■ Data.LOAD ❏ MACHO.LASTUUID()

Data.LOAD.MAP Load MAP file

Format:	**Data.LOAD.MAP** *<file>* [*<class>*] [**/***<option>*]
<option>:	*<generic_load_option>*

Loads a .MAP file from the Logitech Modula2 Cross Development System.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

■ Data.LOAD

Format:	**Data.LOAD.MCDS** *<file>* [*<class>*] [**/***<option>*]
<option>:	*<generic_load_option>*

Loads a file from Hiware Modula2 Cross Development System. The file name is the name of the absolute file, all other files are searched automatically. The source line numbers will be loaded only if the source files are found.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Data.LOAD.MCoff Load MCOFF file

Format:	**Data.LOAD.MCoff** *<file>* [*<range>* | *<class>*] [**/***<option>*]
<option>:	**ALLLINE** *<generic_load_option>*

Loads a file in the MCOFF format (Motorola Common Object File Format). The format is generated by the GNU-56K DSP compiler.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format:	**Data.LOAD.OAT** *<file>* [*<address>*] [**/***<option>*]
<option>:	**LOCATEAT** *<address>*

Loads *.oat files generated by the Android RunTime (ART).

LOCATEAT Relocates the symbols to the specified start *<address>*.

See also

- Data.LOAD
- ▲ 'Release Information' in 'Legacy Release History'

Format: **Data.LOAD.OMF** *<file>* [*<class>*] [**/***<option>*]

<option>:
PLM | **PAS** | **C** | **CPP** | **ADA**
MIX (only 8086, 8051)
LST (only 8051, 8086)
SRC
MIXASM | **MIXPLM** | **MIXPAS** (only 8086)
REAL (only 8086)
NoMMU (only 8086)
MRI | **IC86** | **IC86OLD** | **ParaDigm** | **CADUL** (only 8086)
RevArgs (only 8086)
PLAIN (only 8086, 8096)
EXT | **SPJ** (only 8051)
MMU | **MMU1** (only 8051)
SMALL | **LARGE** (only 8051)
ASMVAR (only 8051)
UBIT (only 8051)
NoGlobal (only 8086)
DOWNLINE
PACK
<generic_load_option>

The implementation of this command is processor specific.

PLM With this option the file extension can be set to '.plm'. The source lines in the object file must relate directly to the source file (no list file).

PAScal Same as above, but for PASCAL files.

C Same as above for 'C' files (only for 80186).
NOTE: For Microtec MCC86, Intel IC86 and Paradigm compilers/converters are extra options available.

MIX Assumes a mixed object file, generated from PLM/86 and another compiler. The switch must be used in combination with another 'language' switch. The PL/M source is loaded from the listing file.

LST Loads line number information from the listfile of PL/M compilers. This option must be set when loading file generated by Intel 8051 or 8086 PL/M compilers.

MIXASM Assumes a mixed object file, generated by a standard compiler and an assembler. The switch must be used in combination with another 'language' switch. If set, the source search path is extended to search first the high level source file (e.g. '.c') and then the assembler source file ('.asm').

MIXPLM	Assumes a mixed object file, generated by a standard compiler and a PL/M compiler. The switch must be used in combination with another 'language' or compiler switch. If set, the source search path is extended to search first the high level source file (e.g. '.c') and then the PL/M source file ('.plm'). When using Intel PL/M, the object files must be converted by the 'cline' utility.
MIXPAS	Assumes a mixed object file, generated by a standard compiler and an pascal compiler. The switch must be used in combination with another 'language' switch. If set, the source search path is extended to search first the high level source file (e.g. '.c') and then the pascal source file ('.pas').
REAL	Assumes that all selectors outside the GDT range are REAL or VIRTUAL-86 mode addresses.
MRI	Loads extended OMF files, as generated by the Microtec MCC86 compiler. This extensions include register variables, bitfields in structures and the name of the source files. The stack traceback is adapted to the MCC86 stack format.
IC86	Load OMF files from Intel IC86. The stack traceback is adapted to the Intel IC86 stack format.
ParaDigm	Loads extended OMF files from PARADIGM LOCATE. The extensions include source file names, register variables and enumeration values.
PLAIN	This option must be used, if the 'BLKDEF' and 'BLKEND' records in the OMF file are not correctly nested.
EXT	Must be set, when loading an extended OMF-51 file (KEIL), i.e. if the nesting of the blocks and the code section in the file reflect the original nesting of the source.
MMU	Generates MMU translation information for KEIL-51 banked linker. Bank 0 is placed to logical address 0x10000, Bank 1 to 0x20000 a.s.o. The first 64K (0x0--0xffff) are transparently translated or used for the common area.
MMU1	Same as above, but different translation. Bank 1 is placed to logical address 0x10000, Bank 2 to 0x20000 a.s.o. The first 64K (0x0--0xffff) are transparently translated or used for the common area. Bank 0 is not allowed in this configuration.
SMALL, LARGE	Define the memory access class for EQU symbols to be either X: or I:.
ASMVAR	Generates HLL variable information for assembler variables.
UBIT	Unsigned bit fields.

NoGlobal Suppresses the global symbols of the file. This can speed up download and save memory when the globals are redundant.

<option> For a description of the generic options, click **<generic_load_option>**.

The following compilers are accepted:

Format	Compiler	Remarks
OMF-51	Intel-PL/M Intel-C51 Keil-C51 SPJ-C KSC/System51	Use **LST** or **PLM** option Use **C** option Use **EXT** and **Puzzled** option, includes extended information. Use the 'OBJECTEXTEND' option to compile the files. Use **MMU** or **MMU1** option when loading files from BL51. Use **SPJ** option. Use **PAS** option.
OMF-96	Intel-C96	Use **SPLIT** option when code and data are two separate memory spaces.
OMF-86	Intel-PL/M Intel-iC86 Microtec Paradigm	Use **LST** or **PLM** or **MIX** or **MIXPLM** option. Use **IC86** or **IC86OLD** option, **RevArgs** option when PASCAL calling conventions are used. Use **MRI** option, includes extended type and source file information. Use **ParaDigm** option, includes extended type and source file information.
OMF-386	Pharlap SSI/Intel SSI/CodeView SSI/Metaware	No option required, includes extended register variables information. Use the '-regvars' option to produce register variable information. No option required. The Codeview debugging format provides more information than the intel format info. No option required. SPF Format from SPLINK. **CPP** switch required for C++.
OMF-166	Keil-C166	Use **Puzzled** option, includes extended type information.

See also

- Data.LOAD
- Data.SAVE.Omf

Format:	**Data.LOAD.Omf2** *<file>* [**/***<option>*]
<option>:	*<generic_load_option>*

Loads OMF-251 files.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Data.LOAD.OriginHex Load special hex files

Format:	**Data.LOAD.OriginHex** *<file> <addressrange>* [**/***<option>*]
<option>:	**OFFSET** *<value>* *<generic_load_option>*

Loads a file in special hex file format.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format:	**Data.LOAD.PureHex** *<file> <address>* | *<range>* [*/<option>*]
<option>:	**SKIP** *<offset>* *<generic_load_option>*

Loads a file in hex-byte format. The file format contains no address information. The input file should contain ASCII hexadecimal data in one or multiple lines.

SKIP *<offset>*	If the option **/SKIP** *<offset>* is specified, the first *<offset>* bytes of the file are omitted.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD
- ▲ 'Release Information' in 'Legacy Release History'

Data.LOAD.REAL Load R.E.A.L. file

Format:	**Data.LOAD.REAL** *<file>* [*/<option>*]
<option>:	*<generic_load_option>*

Loads a file in R.E.A.L. object file format.

<option>	For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format: **Data.LOAD.ROF** *<file>* [*<code>*] [*<data>*] [**/***<option>*]

<option>:
NoSTB
NoDBG
NoMOD
MAP
FPU
CPP
CFRONT
<generic_load_option>

Code and *data* defines the addresses of the code and data regions. With the command **sYmbol.RELOCate** these addresses can be moved after loading the file. The loader loads the three files produced by the compiler (code, symbols, HLL). The symbol files will be searched first on the actual path and then in the subdirectory 'STB'. The option **PATH** should be used to define the path to the source files if the files are compiled on an OS-9 host.

FPU	If code for the FPU has been generated this option should be used.
NoMOD	Is used for loading the symbols only. The file name has to be the name of the symbol file (.stb).
NoSTB	The loading of symbols is suppressed.
NoDBG	The loading of HLL information is suppressed.
MAP	The '.map' file (produced by the linker on request) is used to get the symbols instead of the '.stb' file. The '.map' file includes the absolute symbols, which are not inside the '.stb' file.
<option>	For a description of the generic options, click **<generic_load_option>**.

Limitations

The data symbols are loaded to absolute addresses, i.e. only one copy of the data will contain the symbols. Within the disassembler the base register's relative address offset will only display the correct symbol, when the current base register value has the correct value for this module.

See also

- Data.LOAD

Format:	**Data.LOAD.S1record** *<file>* [*<addressrange>*] [**/***<option>*]
<option>:	**FLAT** **OFFSET** *<offset>* **RECORDLEN** *<value>* *<generic_load_option>*

Load an SREC file containing S19-style 16-bit address records.

If a single address is selected this address will define an address offset like the option **OFFSET**. A given address range will suppress the loading of data and symbols outside of this defined destination address area.

Options available for SREC formats:

FLAT	Loads S-Record files to linear address spaces for those CPUs not supporting linear logical address spaces.
OFFSET	Changes the address value to value plus offset. The Srecord will be loaded to the address plus offset value.
RECORDLEN	Defines the number of data bytes per line in the Srecord file. Decimal values have to be given with decimal point behind.
<option>	For a description of the generic options, click **<generic_load_option>**.

The file may contain also symbolic information, which needs the following format:

```
$$
$$_MODULNAME1
__SYMBOLNAME1 $00000000_
__SYMBOLNAME2 $12345678_
$$_MODULNAME2
__SYMBOLNAME3 $AB0000CF_
```

The character '_' stands for BLANK (0x20). The address has to be entered in 8 digits.

See also

- Data.LOAD
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record

Format: **Data.LOAD.S2record** *<file>* [*<addressrange>*] [**/***<option>*]

<option>: **FLAT**
OFFSET *<offset>*
RECORDLEN *<value>*
<generic_load_option>

Load an SREC file containing S28-style 24-bit address records. The description of options, further information and examples are here.

See also

- Data.LOAD
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record

Data.LOAD.S3record Load S3-Record file

Format: **Data.LOAD.S3record** *<file>* [*<addressrange>*] [**/***<option>*]

<option>: **FLAT**
OFFSET *<offset>*
RECORDLEN *<value>*
<generic_load_option>

Load an SREC file containing S37-style / 32-bit address records. The description of options, further information and examples are here.

See also

- Data.LOAD
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.LOAD.S4record** *<file>* [*<addressrange>*] [**/***<option>*]

<option>: **FLAT**
OFFSET *<offset>*
RECORDLEN *<value>*
<generic_load_option>

Load an SREC file containing S47-style / 64-bit address records. The description of options, further information and examples are here.

See also

- Data.LOAD
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record

Data.LOAD.SAUF Load SAUF file

Format: **Data.LOAD.SAUF** *<file>* [**/***<option>*]

<option>: **CFRONT**
<generic_load_option>

Loads SAUF file format.

See also

- Data.LOAD
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record

Format: **Data.LOAD.SDS** *<file>* [*<address>*] [**/***<option>*]

<option>: **FPU**
NOCONST
PACK
<generic_load_option>

Loads files in Software Development Systems (SDSI) or Uniware format. The address parameter can be used to load via dual port access or to define a different load address for banked applications.

FPU Tells the debugger that code for the FPU was generated by the compiler.

PACK Saves memory space by removing redundant type information. Standard types (e.g. char/long) are assumed to be equal in all modules. Types with the same definition can share the same memory space. This option may save approx 40% of the memory space required without packing.

NOCONST Suppresses the load of "const" variables. These are often removed from the optimizer anyway.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record

Data.LOAD.SPARSE Load SPARSE file

Format: **Data.LOAD.SPARSE** *<file>* [**/***<option>*]

<option>: *<generic_load_option>*

Loads SPARSE image.

See also

- Data.LOAD
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record

Format:	**Data.LOAD.sYm** *<file>* [*<address>*] [**/***<option>*]
<option>:	**LOC** **NOLOC** *<generic_load_option>*

Loads simple symbol files.

The debug information is contained in different types of files. The SYM files (*.sym) contain the global symbols, the optional LOC files (*.loc) contain the local symbols for each module.

<option>	For a description of the generic options, click **<generic_load_option>**.
<file>	Specify the global SYM file as *<file>*. Depending on its content, the LOC files are loaded automatically. If not, the option **LOC** activates the loader for local symbol information and line numbers.

The command accepts the following formats as main symbol files:

PLAIN SYMBOLS

```
1234_SYMBOLNAME1 <TAB> 5678_SYMBOLNAME2
F000_SYMBOLNAME3
```

The hex number (one to 8 digits) is followed by a blank and the symbol name. Multiple symbol names in one line are separated by TAB's (0x9).

ZAX

```
$$ progname
     symbol 1234H
$$ module
     symbol 5678H
     symbol $5678
```

LOC

The local symbol file is compatible to the TRACE80 emulators.

The following example show a LOC file (*.LOC) for a “C” file defining source line #183 at program relative address 0x00AD and one data label at data address 0x0242.The source code must always precede the line definition

```
:c
; vfloat = -1.0;
00AD' 183
0242" mstatic1
```

The module base addresses (code start and end and data start) must be in the global symbol file (*.SYM):

```
1000 [main
1fff ]main
2000 ["main
```

See also

■ Data.LOAD

Data.LOAD.SysRof — Load RENESAS SYSROF file

Format:	**Data.LOAD.SysRof** *<file>* [*<access>*] [*/<option>*]
<option>:	*<generic_load_option>*

Loads a file in Renesas SYSROF object file format.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

■ Data.LOAD ■ Data.SAVE.S1record ■ Data.SAVE.S2record ■ Data.SAVE.S3record

▲ ’Release Information’ in ’Legacy Release History’

Format:	**Data.LOAD.TEK** *<file>* [*<address>*] [**/***<option>*]
<option>:	**NoMMU** *<generic_load_option>*

The optional *address* parameter can be used to load to a different memory class (like E:) or to supply an offset for loading banked applications.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Data.LOAD.TekHex Load TEKTRONIX HEX file

Format:	**Data.LOAD.TekHex** *<file>* [*<address>*] [**/***<option>*]
<option>:	**OFFSET** *<generic_load_option>*

The optional *address* parameter can be used to load to a different memory class (like E:) or to supply an offset for loading banked applications.

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format:	**Data.LOAD.Ubrof** *<file>* [*<address>*] [**/***<option>*]
<option>:	**ICC3S** | **ICC3L** **XSP** **NoMMU** **LARGE** **EXTPATH** *<.extension>* *<generic_load_option>*

Default: MultiLine.

If the option '-r' is used as a compiler option, the source text will be loaded directly from the object file, whereby the option '-rn' the source text will be loaded as usual. The optional *address* parameter can be used to load to a different memory class (like E:) or to supply an offset for loading banked applications.

COLumns	With this option the column debugging is activated.
ICC3S, **ICC3L**	Loads files from ICC8051 3.0.
XSP	Generates virtual stack pointer information for optimized stack frames (68HC12, H8). This is a workaround for not sufficient information in the debug file.
NoMMU	Doesn't load the MMU tables from the file contents. This table is used to translate physical addresses to their logical counterparts (only 64180).
LARGE	This option must be set when a large memory model is used.
EXTPATH	Defines an alternate file extension for the source files, if the .c file is not found.
<option>	For a description of the generic options, click **<generic_load_option>**.

See also

■ Data.LOAD

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.LOAD.VersaDos** *<file>* [*<access_class>*] [**/***<option>*]

<option>: **NoLO**
NoDB
<generic_load_option>

Loads the *.lo file first, which contains the code, and then the *.db symbol file (if existent). The file name of the *.lo file must be given.

<option> For a description of the generic options, click **<generic_load_option>**.

NoDB With the option **NoDB** the loading of the symbols is suppressed.

NoLO The option **NoLO** suppresses the loading of the data file. The symbol file name has to be given as an argument in this case.

See also

- Data.LOAD

Data.LOAD.XCoff Load XCOFF file

Format: **Data.LOAD.XCoff** *<file>* [*<class>*] [**/***<option>*]

<option>: *<generic_load_option>*

Loads a file in the IBM-RS6000/XCOFF format (PowerPC).

<option> For a description of the generic options, click **<generic_load_option>**.

See also

- Data.LOAD

Format: **Data.MSYS** *<dll_file> <cmdline>*

Starts a flashdisk support utility to program, view or format M-Systems flashdisks. The utility is supplied by M-Systems in form of a DLL module. The syntax of the command depends on the DLL module.

Data.Out Write port

Format: **Data.Out** *<address>* [**%**[*<accessformat>*.]*<dataformat>*] *<data>* [**/***<option>*]

<accessformat>: **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

<dataformat>: **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** **BE** | **LE**

<option>: **Repeat** | **CORE** *<core_number>*

As opposed to the **Data.Set** command, the address is not increased during write-to. If the CPU structure decides between IO and DATA area, the IO area is selected on default (Z80, 186 …).

Byte, **Word**, … See **"Keywords for <width>"**, page 11.

Repeat Repeats the input endlessly, e.g. for external measurements.

Examples:

```
Data.Out IO:0x10 0x33              ; write one byte to I/O space

Data.Out D:0x10 0x33               ; write one byte to memory-mapped I/O

Data.Out IO:0x10 "ABC" 0x33        ; writes 4 characters to io port

Data.Out IO:0x10 "ABCD" /Repeat    ; continuously writes data to the port

; write 32-bit bytewise
Data.Out IO:0x10 %Byte.Long 0x12345678
```

See also

■ Data.In ■ Data.Set ■ Data.Test ❑ ADDRESS.OFFSET()
❑ ADDRESS.SEGMENT() ❑ ADDRESS.STRACCESS() ❑ ADDRESS.WIDTH()

Data.PATTERN — Fill memory with pattern

[Examples]

Format: **Data.PATTERN** *<addressrange>* [*/<option>*]

<option>:
Verify | **ComPare** | **DIFF**
ByteCount | **WordCount** | **LongCount**
ByteShift | **WordShift** | **LongShift**
RANDOM | **PRANDOM**
Byte | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
PlusVM
SEED *<value>*
LFSR32 *<taps>*

Fills the memory with a predefined pattern for the specified address range.

Verify	Verify the data by a following read operation.
ComPare	Pattern is compared against memory. Memory is not changed. The comparison stops after the first difference
DIFF	Pattern is compared against memory. Memory is not changed. The result of the compare is available in the **FOUND()** function.
ByteCount, **WordCount**, **LongCount**	Pattern is an incrementing count of 8, 16, or 32 bit.
ByteShift, **WordShift**, **LongShift**	Pattern is an left rotating bit of 8, 16, or 32 bit.
RANDOM	Pattern is a random sequence.
PRANDOM	Pattern is a pseudo random sequence.
Byte, **Word**, …	Specify memory access size. See **"Keywords for <width>"**, page 11. If no access size is specified, the debugger uses the optimum size for the processor architecture.
PlusVM	The data patter is written into the target memory plus into the virtual memory.

LFSR32 *<taps>*

Pattern is generated by a 32-bit Linear-Feedback-Shift-Register. *<taps>* is a 32-bit number.
The pseudo code used to generate the sequence is

```
if (lfsr & 0x80000000)
  lfsr = (lfsr<<1)^(taps|0x1);
else
  lfsr = (lfsr<<1);
```

SEED *<value>*

This option is used together with the **PRANDOM** or **LFSR32** option in order to initialize the pseudo random generator or the initial LFSR value with a different seed.

Example 1

This example shows how to test memory address translations (MMU) with a pattern:

```
Data.PATTERN A:0x0--0x7ffff           ; fill memory with pattern
…                                     ; (A: enforces physical address)
Data.dump 0x0                         ; display logical view of memory
Data.dump 0x4000
…
Data.PATTERN 0x0--0x7fff /ComPare     ; compare memory against pattern
```

Example 2

The **Data.PATTERN** *<addressrange>* **/DIFF** command is used together the following functions:

FOUND()	Returns TRUE if a difference was found in the comparison.
TRACK.ADDRESS()	Returns the address of the first difference.

```
…
;fill memory with a predefined pattern
Data.PATTERN 0x0++0xffff

;any write access or code manipulation
…

;compare predefined pattern against memory to check if the previous
;write access or code manipulation has had an impact on the pattern
Data.PATTERN 0x0++0xffff /DIFF

IF FOUND()
     PRINT "Error found at address " TRACK.ADDRESS()
…
```

Example 3

```
;Same as /LongShift
Data.PATTERN VM:0x0000--0xffff /LFSR32 0x1 /SEED 0x1

;LFSR as pseudo random sequence, starting with 0x12345678
;Note: There are many other <taps> values, which result in long number
;sequences.
Data.PATTERN VM:0x0000--0xffff /LFSR32 0x10904081 /SEED 0x12345678
```

See also

■ Data.dump ■ Data.Set ■ Data.Test

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.Print** [[**%***<format>*][*<address>* | *<range>*] …] [**/***<option>* …]

<format>:
Decimal [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
DecimalU [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Hex [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
HexS [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
OCTal [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Ascii [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Binary [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Float[.*<float_rep>*[.*<endianness>*]]
sYmbol [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Var
DUMP [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Byte [.*<endianness>* [.*<bitorder>*]]
Word [.*<endianness>* [.*<bitorder>*]]
Long [.*<endianness>* [.*<bitorder>*]]
Quad [.*<endianness>* [.*<bitorder>*]]
TByte [.*<endianness>* [.*<bitorder>*]]
PByte [.*<endianness>* [.*<bitorder>*]]
HByte [.*<endianness>* [.*<bitorder>*]]
SByte [.*<endianness>* [.*<bitorder>*]]

<width>: **DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

<endianness>: **DEFault** | **LE** | **BE**

<bitorder>: **DEFault** | **BitSwap**

<option>:
CORE *<core_number>*
COVerage
CTS
Track
FLAG *<flag>*
CFlag *<cflag>*
Mark *<break>*

<flag>: **Read** | **Write** | **NoRead** | **NoWrite**

<cflag>: **OK** | **NoOK** | **NOTEXEC** | **EXEC**

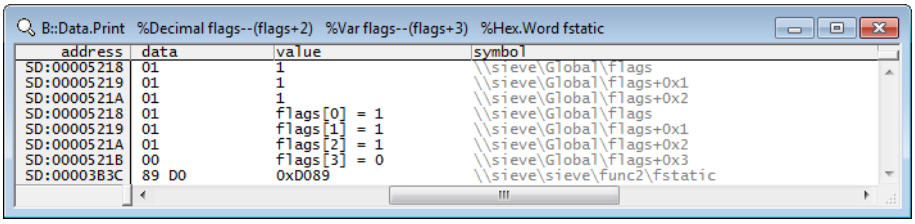
<break>: **Program** | **HII** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

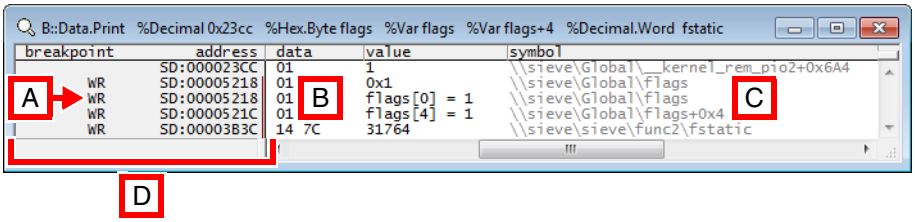
Displays the bare memory content on multiple address ranges as a list. If the single address format is selected, only one word at this address will be displayed. When selecting an address range the defined data range can be dumped.

Decimal, DecimalU,...	Refer to **"Keywords for <format>"**, page 10
Byte, Word, ...	Refer to **"Keywords for <width>"**, page 11
DEFault, BE, LE	Define byte-order display direction: default target endianness, Big Endian or Litte Endian
DEFault, BitSwap	**BitSwap** allows to display data in reverse bit-order in each byte. If **BitSwap** is used together with BE or LE, the byte order will not change, otherwise **BitSwap** will also reverse the byte-order.
CORE *<core>*	Display memory from the perspective of the specified core /SMP debugging only).
COVerage	Highlight data memory locations that have never been read/written.
Track	Track the window to the reference address of other windows.
Mark *<break>*	Highlight memory locations for which the specified breakpoint is set.
CTS	Display **CTS** access information when CTS mode is active.

Examples:

```
Data.Print 0x1000--0x10ff           ; display fixed range

Data.Print Var.RANGE(flags)         ; display range defined by object
                                    ; name

Data.Print %Binary Register(ix)     ; display data byte referenced by IX

Data.Print %Var                     ; display indexed by HLL pointer
Var.VALUE(dataptr)
```





A Read/Write breakpoint (created with **Break.Set**).

B Hex data.

C Symbolic address.

D Scale area.

The scale area contains Flag and Breakpoint information, memory classes and addresses. The state line displays the currently selected address, both in hexadecimal and symbolic format. By double-clicking a data word, a **Data.Set** command can be executed on the current address.

By holding down the right mouse button, the most important memory functions can be executed via the **Data Address** pull-down menu. If the **Mark** option is on, the relevant bytes will be highlighted. For more information, see **Data.dump**.

See also

- Data.dump
- ADDRESS.SEGMENT()
- Data.Float()
- Data.STRingN()
- Data.TABle
- ADDRESS.STRACCESS()
- Data.Long()
- Data.Word()
- Data.View
- ADDRESS.WIDTH()
- Data.Quad()
- ADDRESS.OFFSET()
- Data.Byte()
- Data.STRing()

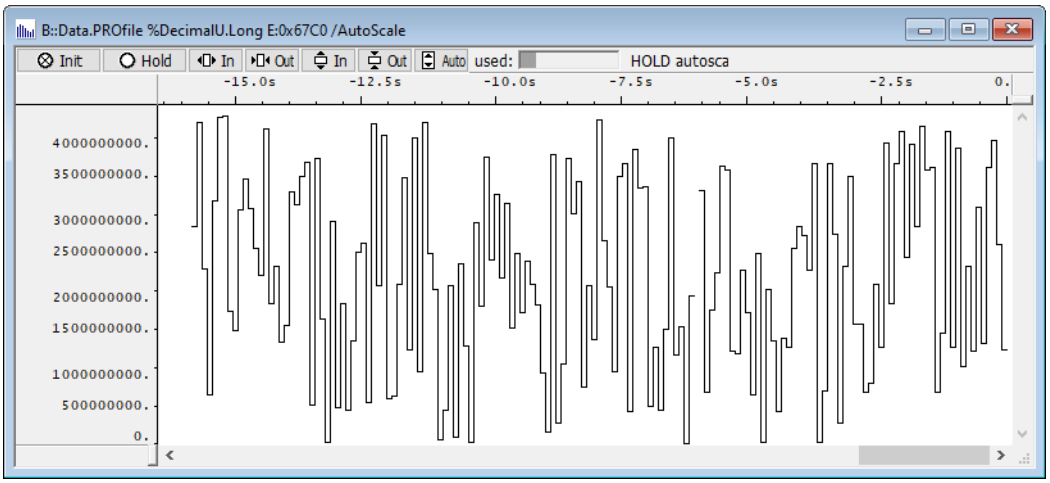
Format:	**Data.PROfile** [**%***<format>*][*<address>* …] [*<gate>*] [*<scale>*] [**/***<option>*]
<format>:	**Decimal.**[*<width>*[**.***<endianness>*]] **DecimalU.**[*<width>*[**.***<endianness>*]] **Hex.**[*<width>*[**.***<endianness>*]] **HexS.**[*<width>*[**.***<endianness>*]] **OCTal.**[*<width>*[**.***<endianness>*]] **Float**[**.***<float_rep>*[**.***<endianness>*]] **Byte** [**.***<endianness>*] **Word** [**.***<endianness>*] **Long** [**.***<endianness>*] **Quad** [**.***<endianness>*] **TByte** [**.***<endianness>*] **PByte** [**.***<endianness>*] **HByte** [**.***<endianness>*] **SByte** [**.***<endianness>*]
<width>:	**DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<gate>:	**0.1s** | **1.0s** | **10.0s**
<scale>:	**1.** … **32768.**
<option>:	**AutoInit** | **AutoArm** | **AutoScale**

The value at the specified memory location(s) is displayed graphically. The display requires **run-time memory access** if the data value should be displayed while the program execution is running. The display is updated and shifted every 100 ms. The *<gate>* parameter allows to change this shift rate.

AutoInit	The results are re-initialized each time the program execution is started.
AutoArm	Update is started and stopped with the program execution.
AutoScale	The graph is automatically scaled according to the sampled data values.

Example 1:

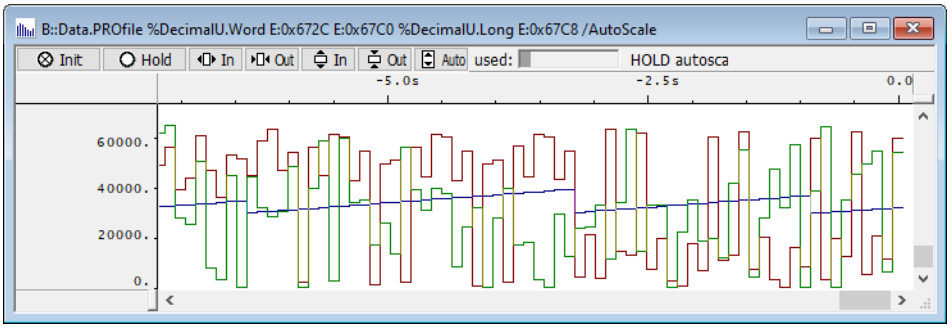
```
Data.PROfile %DecimalU.Long E:0x67C0 /AutoScale
```



Buttons	
Init	Restart display
Hold	Stop update/re-start update
In, Out	Zoom in/out horizontally and vertically
Auto	Enable auto scale

Example 2: Up to three data values can be displayed. The following color assignment is used: first data value red, second data value green, third data value blue.

```
Data.PROfile %DecimalU.Word E:0x672C E:0x67C0 %DecimalU.Long E:0x67C8
/AutoScale
```



See also

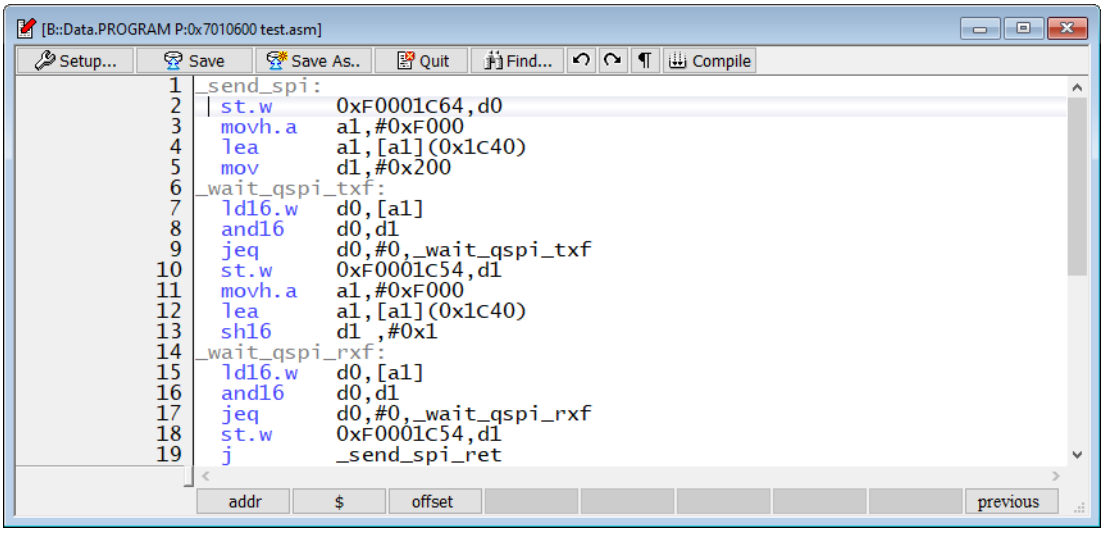
- Data.DRAW
- Var.PROfile

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.PROGRAM** [*<address>* | *<addressrange>* [*<file>* [*<line>*]]]

This command creates a window for editing and assembling a short assembler program. Without a specified file name, the file t32.asm is generated.

If the **Compile** button is used, syntax errors and undefined labels will be detected. The resulting program will be assembled for the specified address and saved to memory. The labels entered will be added to the symbol list.



See also

- Data.Assemble
- Data.ReProgram
- Data.Set
- SETUP.DropCoMmanD
- SETUP.EDITOR

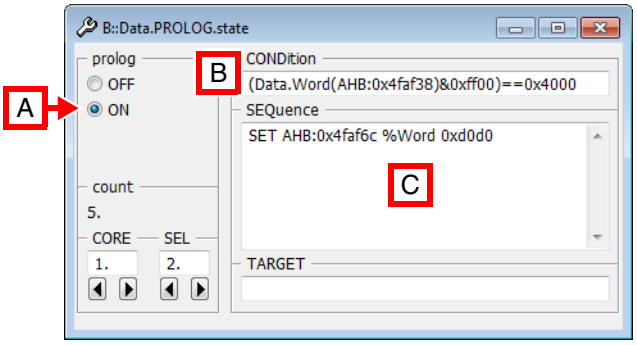
▲ 'Text Editors' in 'PowerView User's Guide'
▲ 'Release Information' in 'Legacy Release History'

The **Data.PROLOG** command group allows to define a sequence of read/write accesses that are automatically performed directly before the program execution is continued with **Go** or **Step**.

The **Data.PROLOG** command group can also be used, for example, to manually freeze peripherals, if the processor itself does not provide this feature.

The complementary command **Data.EPILOG** performs read/write accesses after program execution halted. It is also possible to store data read with **Data.EPILOG** and restore with **Data.PROLOG**, and vice versa.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **Data.PROLOG.state** window:



A For descriptions of the commands in the **Data.PROLOG.state** window, please refer to the **Data.PROLOG.*** commands in this chapter. Example: For information about **ON**, see **Data.PROLOG.ON**.

B Conditions can be set up in the **CONDition** field using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** field using the *<data_set_commands>* **SET**, **SETI**, **GETS**, and **SETS**.

Examples:

- Overview including illustration - see **Data.PROLOG.state**.
- Prolog conditions - see **Data.PROLOG.CONDition**.
- Access sequences - see **Data.PROLOG.SEQuence**.

See also

■ Data.ATTACH ■ Data.EPILOG ■ Data.STARTUP ■ Data.TIMER

▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.PROLOG.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **Data.PROLOG.SEQuence** will be executed each time before the program execution is started by **Go** / **Step**.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

Examples:

```
;reads the long at address D:0x3faf30, performs a binary AND with
;a mask (here 0xffffffff). If the result is equal to 0x80000000, then the
;condition is true and the defined sequence is executed.
Data.PROLOG.CONDition (Data.Long(D:0x3faf30)&0xffffffff)==0x80000000
```

```
;reads the word at address D:0x3xfaf30
Data.PROLOG.CONDition (Data.Word(D:0x3faf30)&0xff00)!=0x8000
```

```
;reads the byte at address D:0x3xfaf30
Data.PROLOG.CONDition (Data.Byte(D:0x3faf30)&0xf0)!=0x80
```

See also

■ Data.PROLOG.state ❏ Data.Byte() ❏ Data.Long() ❏ Data.Word()

Format: **Data.PROLOG.CORE** *<core_number>*

Selects the core for which you want to define one or more data prologs.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: This script shows how to define a data prolog that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a data prolog
Data.PROLOG.CORE 3.

;Define the data prolog for core 3
Data.PROLOG.CONDition <your_code>
Data.PROLOG.SEQuence  <your_code>
```

For information on how to configure two different data epilogs, see **Data.PROLOG.SELect**.

See also

- Data.PROLOG.state

Data.PROLOG.OFF Switch data prolog off

Format: **Data.PROLOG.OFF**

Disables the execution of the **Data.PROLOG** sequence on program execution start.

See also

- Data.PROLOG.state

Data.PROLOG.ON — Switch data prolog on

Format: **Data.PROLOG.ON**

Enables the execution of the **Data.PROLOG** sequence on program execution start.

See also

- Data.PROLOG.state

Data.PROLOG.RESet — Reset all data prologs

Format: **Data.PROLOG.RESet**

Switches the **Data.PROLOG** feature off and clears all settings.

See also

- Data.PROLOG.state

Format: **Data.PROLOG.SELect** *<serial_number>*

Increments the index number for each new data prolog. This is useful, for example, if you need two separate data prologs with each data prolog having its own **Data.PROLOG.CONDition**.

TRACE32 automatically assigns the index number 1. to the 1st **Data.PROLOG.SEQuence**. If you require a 2nd, separate data prolog sequence, then increment the *<index_number>* to 2. Otherwise the 2nd data prolog will overwrite the 1st data prolog. You can define a maximum of 10 data prologs.

Example 1: Two data prologs with the *same* **Data.PROLOG.CONDition** may have the *same* index number. The backslash **** is used as a line continuation character. No white space permitted after the backslash.

```
;Set the index number to 1.
Data.PROLOG.SELect 1.

;Data PROLOG sequences shall be executed only if this condition is true:
Data.PROLOG.CONDition (Data.Word(D:0x4faf34)&0xff00)==0x4000

;Define the two data PROLOG sequences:
Data.PROLOG.SEQuence SET 0x4faf54 %Word 0xC0C0 \
                     SET 0x4faf64 %Word 0xD0D0
```

Example 2: Two data prologs with *different* **Data.PROLOG.CONDition** settings require two *different* index numbers.

```
;1st data prolog - TRACE32 automatically sets the index number to 1.
Data.PROLOG.SELect 1.

;If this prolog condition is true, ...
Data.PROLOG.CONDition (Data.Word(D:0x4faf38)&0xff00)==0x2000

;... then the 1st prolog sequence will be executed
Data.PROLOG.SEQuence SET 0x4faf58 %Word 0xE0E0

;Increment the index number to define the 2nd data prolog
Data.PROLOG.SELect 2.

;If this prolog condition is true, ...
Data.PROLOG.CONDition (Data.Word(D:0x4faf38)&0xff00)==0x3000

;... then the 2nd prolog sequence will be executed
Data.PROLOG.SEQuence SET 0x4faf58 %Word 0xF0F0
```

See also

- Data.PROLOG.state

Format:	**Data.PROLOG.SEQuence** *<data_set_command>* …
<data_set_ command>:	**SET** *<address>* **%***<format>* *<data>* **SETI** *<address>* **%***<format>* *<data>* *<increment>* **SETS** *<address>* **GETS** *<address>*

Defines a sequence of *<data_set_commands>* that are automatically executed by the TRACE32 software directly before the program execution is started by **Go** / **Step**.

SET Parameters: *<address>* **%***<format>* *<value>*
Write *<value>* with data type *<format>* to *<address>*

SETI Parameters: *<address>* **%***<format>* *<start>* *<increment>*
At the first time performed, write *<start>* to *<address>*.
<start> is incremented by *<increment>* on each successive call.

GETS Parameters: *<address>* **%***<format>*
Reads the value at *<address>* and stores it into an internal data buffer.
The internal data buffer can contain multiple records and is reset when the command **Data.PROLOG.SEQuence** is called.

SETS Parameters: *<address>* **%***<format>*
If the internal data buffer contains a record for *<address>*, the stored value is written to the processor.

Examples:

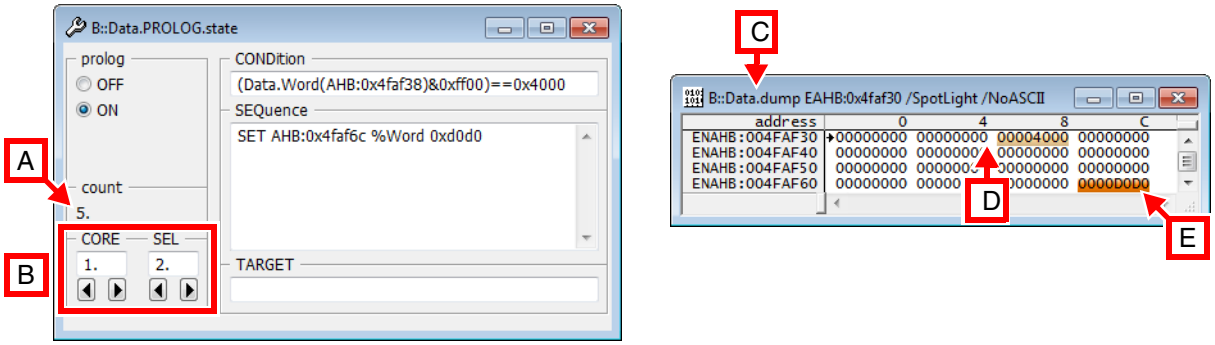
```
;Write 0xa0a0 when starting, increment by 2 for each successive start
Data.PROLOG.SEQuence SETI 0x3faf50 %Word 0xa0a0 2

;Set peripheral register to 0 when halted, 1 when starting
Data.EPILOG.SEQuence SET 0x3faf50 %Long 0x00000000
Data.PROLOG.SEQuence SET 0x3faf50 %Long 0x00000001

;Set register to 0 when halted, restore original value when starting
Data.EPILOG.SEQuence GETS 0x1230 %Byte SET 0x1230 %Byte 0x00
Data.PROLOG.SEQuence SETS 0x1230 %Byte

;Set (clear) a single bit when starting (stopping)
Data.EPILOG.SEQuence SET 0x3faf50 %Word 0yXXXX1xxxXXXXxxxx
Data.PROLOG.SEQuence SET 0x3faf50 %Word 0yXXXX0xxxXXXXxxxx
```

See also

- Data.PROLOG.state

Format: **Data.PROLOG.state**

Opens the **Data.PROLOG.state** window, where you can configure data prologs.



A Counts the number of times the **Data.PROLOG.SEQuence** command has been executed.

B Lets you create and view the data prologs of a particular core. This example shows the 2nd data prolog of core 1. The **CORE** field is grayed out for single-core targets.

C … **E** The **Data.dump** window is just intended to visualize what happens behind the scenes:

- The access class E: in the **Data.dump** window is required if you want the window to display memory while the program is running; refer to [**C**].
- The **CONDition** [**D**] is true (==0x4000), and thus the **SEQuence** is executed [**E**] before the program execution is started with the **Go** command.

```
Data.PROLOG.state       ;open the window
Data.PROLOG.CORE 1.     ;for core 1, two data prologs will be defined:

Data.PROLOG.SELect 1.   ;1st data prolog with condition and sequence:
                        ;if condition is true, then execute seq. below
Data.PROLOG.CONDition (Data.Word(AHB:0x4faf38)&0xff00)==0x3000
Data.PROLOG.SEQuence SET AHB:0x4faf5c %Word 0xc0c0

Data.PROLOG.SELect 2.   ;2nd data prolog with condition and sequence:
                        ;if condition is true, then execute seq. below
Data.PROLOG.CONDition (Data.Word(AHB:0x4faf38)&0xff00)==0x4000
Data.PROLOG.SEQuence SET AHB:0x4faf6c %Word 0xd0d0

Data.PROLOG.ON          ;activate all data prologs
Go                      ;start program execution
```

See also

- Data.PROLOG.CONDition
- Data.PROLOG.CORE
- Data.PROLOG.OFF
- Data.PROLOG.ON
- Data.PROLOG.RESet
- Data.PROLOG.SELect
- Data.PROLOG.SEQuence
- Data.PROLOG.TARGET

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.PROLOG.TARGET** *<code_range> <data_range>*

Defines a target program that is automatically started by the TRACE32 software directly before the program execution is started by **Go** / **Step**.

<code_range> Defines the address range for the target program.

<data_range> Defines the address range used for the data of the target program.

Example:

```
Data.PROLOG.TARGET 0x3fa948--0x3faa07 0x1000--0x1500
```

See also

- Data.PROLOG.state

Format: **Data.REF** [*<address>*] [**/***<option>*]

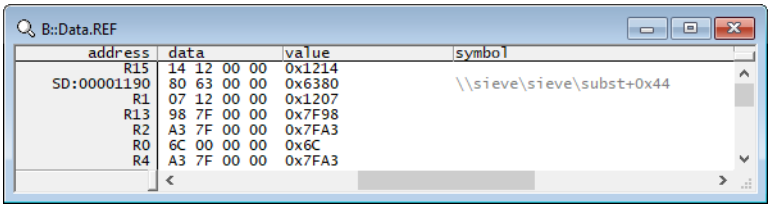
<option>: **CORE** *<core_number>*
COVerage
CTS
Track
FLAG *<flag>*
CFlag *<cflag>*
Mark *<break>*

<flag>: **Read** | **Write** | **NoRead** | **NoWrite**

<cflag>: **OK** | **NoOK** | **NOTEXEC** | **EXEC**

<break>: **Program** | **Hll** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

Displays all values (registers or memory locations) that are referenced by the current assembler instruction. This command is not implemented for all processors.



CORE *<core>* Display memory from the perspective of the specified core /SMP debugging only).

COVerage Highlight data memory locations that have never been read/written.

Track Track the window to the reference address of other windows.

Mark *<break>* Highlight memory locations for which the specified breakpoint is set.

CTS Display **CTS** access information when CTS mode is active.

Format: **Data.ReProgram** [*<address>* | *<addressrange>* [*<file>*]]

Assembles instructions from a file into memory. It is similar to the **Data.PROGRAM** command.

The **Data.Assemble** command can be used to patch single instructions.

See also

- Data.PROGRAM

Data.ReRoute Reroute function call

Format: **Data.ReRoute** *<address_range>* *<old_destination>* *<new_destination>* [*<exclude_range>*]

Replaces, within the specified address range, all function calls to *<old_destination>* by function calls to *<new_destination>*. Function calls within *<exclude_range>* are not replaced.

Examples:

```
; replace all function calls to 0x3fa96c by 0x3fa9e4
; within the address range 0x3f9900--0x3fae1f
Data.ReRoute 0x3f9900--0x3fae1f 0x3fa96c 0x3fa9e4
```

```
; replace all function calls to func5 by func7 within the module diabc
Data.ReRoute diabc func5 func7
```

```
; replace all function calls to malloc by T32_malloc within the code
; segment of the currently loaded program
; don't replace the calls to malloc in the module t32mem
Data.ReRoute sYmbol.SECRANGE(.text) malloc T32_malloc \t32mem
```

[Examples]

Format:	**Data.SAVE.***<format>* *<file>* [*<addressrange>*] [*/<option>*]
<option>:	*<format_specific_save_options>* *<generic_save_options>*

Saves the data from the specified address range in a file with the specified file format.

<file> File name and (optional) path

<addressrange> Address range to be saved. It is possible to use access classes, e.g. A:

<option>
- For descriptions of the format-specific save options, refer to the respective **Data.SAVE.<format>** command.
- For descriptions of the generic save options, see table **below**.

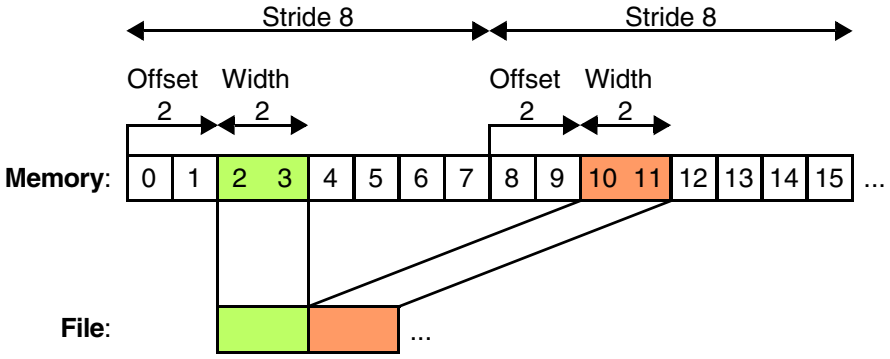
List of Generic Save Options

BSPLIT *<stride><offset>* [*<width>*]	Saves only certain bytes of the memory. • *<stride>* defines a chunk of data to which the other two parameters refer to. • *<width>* defines the bus width in bytes. • *<offset>* defines the offset of the bytes being saved. • For an illustration of *<stride>*,*<offset>*, and *<width>*, see **below**. The option **BSPLIT 2 0** saves the lower byte of a 16-bit bus.
Byte **Word** **TByte** **Long** **PByte** **HByte** **SByte** **Quad**	Data is saved in the specified width: • **Byte** (8-bit accesses) **Word** (16-bit accesses) • **TByte** (24-bit accesses) **Long** (32-bit accesses) • **PByte** (40-bit accesses) **HByte** (48-bit accesses) • **SByte** (56-bit accesses) **Quad** (64-bit accesses) Must be used if the target can only support memory accesses of a fixed width. The default width is determined automatically by TRACE32 to achieve the best upload speed.
LongSWAP	Swaps high and low bytes of a 32-bit word during save.
QuadSWAP	Swaps high and low bytes of a 64-bit word during save.
SkipErrors	Skips memory that cannot be read. Otherwise TRACE32 would abort the command if a bus error occurs while saving the specified *<range>*. No data is saved for addresses that cause bus errors, provided the format allows this.

wordSWAP	Swaps high and low bytes of a 16-bit word during save.
BITSWAP	Swaps the bits of each byte during save.
NoINCrement	Saves code to a single address (of a FIFO).

BSPLIT: illustration of <stride>, <offset>, and <width>

[Back]



Examples

```
; Save data from uncached and physical address 0x00000000--0x0001ffff
; to s3record file.
Data.SAVE.S3record flashdump.s3 ANC:0x00000000--0x0001ffff
```

```
; Save data from supervisor data memory 0xc0000000--0xcfffffff
; to binary file and compress.
Data.SAVE.Binary memorydump.bin.zip SD:0xc0000000--0xcfffffff /ZIP
```

See also

- Data.SAVE.Ascii
- Data.SAVE.AsciiHex
- Data.SAVE.AsciiOct
- Data.SAVE.BDX
- Data.SAVE.Binary
- Data.SAVE.CCSDAT
- Data.SAVE.DAB
- Data.SAVE.Elf
- Data.SAVE.ESTFB
- Data.SAVE.IntelHex
- Data.SAVE.Omf
- Data.SAVE.PureHex
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record
- Data.SAVE.S4record

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.SAVE.Ascii** *<file>* [*<addressrange>*] [**/***<option>*]

<option>: **Hex** | **Decimal** | **DecimalU** | **BINary** |
Float. [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] |
Append

Saves an *<addressrange>* as a pure data file in word-oriented ASCII file format. The output file includes one byte in each line.

Append Appends data to an existing file.

<option> Saves the file in one of the following formats:

DecimalU
- Unsigned Decimal

Decimal
- Signed Decimal

Hex
- Hexadecimal value

Binary
- Binary value

Float
- Floating point value

See also

- Data.SAVE.<format>

Data.SAVE.AsciiHex Save hex file

Format: **Data.SAVE.AsciiHex** *<file>* [*<addressrange>*] [**/***<option>*]
Data.SAVE.AsciiHexP *<file>* [*<addressrange>*] [**/***<option>*]
Data.SAVE.AsciiHexA *<file>* [*<addressrange>*] [**/***<option>*]
Data.SAVE.AsciiHexS *<file>* [*<addressrange>*] [**/***<option>*]
Data.SAVE.AsciiHexC *<file>* [*<addressrange>*] [**/***<option>*]
Data.SAVE.AsciiHexB *<file>* [*<addressrange>*] [**/***<option>*]

<option>: **OFFSET** *<offset>*
<generic_save_options>

Saves a file in a simple ASCII file format.

<option> For a description of the generic options, see **<generic_save_options>**.

Examples:

TRACE32 Command Line	Content of file 'x.txt'
D.SAVE.AH x.txt d:0++1f	***<STX>***$A0000, DA F2 DA 33 69 8C 83 B4 F7 6E 59 E8 48 7D 90 64 85 29 75 66 84 F1 A4 05 52 34 51 CA 36 B0 04 73 ***<ETX>***$S1009
D.SAVE.AHP x.txt d:0++1f	***<STX>***$A0000, DA%F2%DA%33%69%8C%83%B4%F7%6E%59%E8%48%7D%90%64% 85%29%75%66%84%F1%A4%05%52%34%51%CA%36%B0%04%73% ***<ETX>***$S1009
D.SAVE.AHA x.txt d:0++1f	***<STX>***$A0000, DA'F2'DA'33'69'8C'83'B4'F7'6E'59'E8'48'7D'90'64' 85'29'75'66'84'F1'A4'05'52'34'51'CA'36'B0'04'73' ***<ETX>***$S1009
D.SAVE.AHS x.txt d:0++1f	***<DC2>***$A0000, DA'F2'DA'33'69'8C'83'B4'F7'6E'59'E8'48'7D'90'64' 85'29'75'66'84'F1'A4'05'52'34'51'CA'36'B0'04'73' ***<DC4>***$S1009
D.SAVE.AHC x.txt d:0++1f	***<STX>***$A0000. DA,F2,DA,33,69,8C,83,B4,F7,6E,59,E8,48,7D,90,64, 85,29,75,66,84,F1,A4,05,52,34,51,CA,36,B0,04,73, ***<ETX>***$S1009
D.SAVE.AHB x.txt d:0++1f	DA F2 DA 33 69 8C 83 B4 F7 6E 59 E8 48 7D 90 64 85 29 75 66 84 F1 A4 05 52 34 51 CA 36 B0 04 73

Key: ***<STX>***=(char)0x02, ***<ETX>***=(char)0x03, ***<DC2>***=(char)0x12, ***<DC4>***=(char)0x14
Lines end with <CR><LF>=(char)0x0D(char)0x0A, added after the byte if (addr & 0x0F == 0x0F).
Address prefix is $A, Checksum $S (where available) is 16bit sum of bytes.

See also

- Data.SAVE.<format>

Format: **Data.SAVE.AsciiOct** *<file>* [*<addressrange>*] [**/***<option>*]
Data.SAVE.AsciiOctP *<file>* [*<addressrange>*] [**/***<option>*]
Data.SAVE.AsciiOctA *<file>* [*<addressrange>*] [**/***<option>*]
Data.SAVE.AsciiOctS *<file>* [*<addressrange>*] [**/***<option>*]

<option>: **OFFSET** *<offset>*
<generic_save_options>

Saves a file in a simple ASCII file format.

<option> For a description of the generic options, see **<generic_save_options>**.

Examples:

```
; Command
Data.SAVE.AsciiOct out.txt 0x0--0x1f
; out.txt:
<STX>$A000000,
356 335 314 273 252 002 003 004 005 006 007 010 011 012 013 014
015 016 017 020 021 022 023 024 025 026 027 030 031 032 033 034
<ETX>$S100261

; Command
Data.SAVE.AsciiOctP out.txt 0x0--0x1f
; out.txt:
<STX>$A000000,
356%335%314%273%252%002%003%004%005%006%007%010%011%012%013%014%
015%016%017%020%021%022%023%024%025%026%027%030%031%032%033%034%
<ETX>$S100262

; Command
Data.SAVE.AsciiOctA out.txt 0x0--0x1f
; out.txt:
<STX>$A000000,
356'335'314'273'252'002'003'004'005'006'007'010'011'012'013'014'
015'016'017'020'021'022'023'024'025'026'027'030'031'032'033'034'
<ETX>$S100262

; Command
Data.SAVE.AsciiOctA out.txt 0x0--0x1f
; out.txt:
<DC2>$A000000,
356'335'314'273'252'002'003'004'005'006'007'010'011'012'013'014'
015'016'017'020'021'022'023'024'025'026'027'030'031'032'033'034'
<DC4>$S100262
```

Key: ***<STX>***=(char)0x02, ***<ETX>***=(char)0x03, ***<DC2>***=(char)0x12, ***<DC4>***=(char)0x14Lines end with <CR><LF>=(char)0x0D(char)0x0A, added after the byte if (addr & 0x0F == 0x0F).
Address prefix is $A, Checksum $S (where available) is 16bit sum of bytes.

See also

- Data.SAVE.<format>
- ▲ 'Data Access' in 'EPROM/FLASH Simulator'

Data.SAVE.BDX — Save BDX file

Format:	**Data.SAVE.BDX** *<file>* *<addressrange>* [**/***<option>*]
<option>:	*<generic_save_options>*

Saves a file in BDX format (binary format).

<option> — For a description of the generic options, see **<generic_save_options>**.

See also

- Data.SAVE.<format>

Data.SAVE.Binary — Save binary file

Format:	**Data.SAVE.Binary** *<file>* [*<addressrange>*] [**/***<option>*]
<option>:	**ZIP** | **Append** *<generic_save_options>*

The contents of the entire address range are saved if no address parameter has been defined! The save procedure may be interrupted at any time (Control-C).

<option> — For a description of the generic options, see **<generic_save_options>**.

See also

- Data.SAVE.<format>
- Data.LOAD.Binary

Format:	**Data.SAVE.CCSDAT** *<file> <addressrange>* [*/<option>*]
<option>:	**OFFSET** *<offset>* *<generic_save_options>*

Saves memory in CCSDAT file format.

<option> For a description of the generic options, see **<generic_save_options>**.

See also

- Data.SAVE.<format>
- ▲ 'Release Information' in 'Legacy Release History'

Data.SAVE.DAB Save DAB file

Format:	**Data.SAVE.DAB** *<file> <addressrange>* [*/<option>*]
<option>:	*<generic_save_options>*

Saves memory in DAB file format.

<option> For a description of the generic options, see **<generic_save_options>**.

See also

- Data.SAVE.<format>
- ▲ 'Release Information' in 'Legacy Release History'

Format:	**Data.SAVE.Elf** *<file> <addressrange>* [**/***<option>*]
<option>:	**ELF32** | **ELF64** *<generic_save_options>*

Saves binary data in ELF format.

<option>	For a description of the generic options, see **<generic_save_options>**.

See also

- Data.SAVE.<format>

Data.SAVE.ESTFB Save EST flat binary file

Format:	**Data.SAVE.ESTFB** *<file> <addressrange>* [**/***<option>*]
<option>:	*<generic_save_options>*

Saves memory in EST flat binary file format.

<option>	For a description of the generic options, see **<generic_save_options>**.

See also

- Data.SAVE.<format>

Format: **Data.SAVE.IntelHex** *<file> <addressrange>* [**/***<option>*]

<option>: **ADDR** *<address_size>*
Append
TYPE2
TYPE4
OFFSET *<offset>*
<generic_save_options>

Saves a file in an IntelHex format.

ADDR Defines the *<address_size>* of the INTEL-HEX file.

Append Appends data to an existing file.

TYPE2 Defines 20 bits for the address field.

TYPE4 Defines 32 bits for the address field.

OFFSET Gives the offset to the address range to store.

<option> For a description of the generic options, see **<generic_save_options>**.

See also

■ Data.SAVE.<format> ■ Data.LOAD.IntelHex

Data.SAVE.Omf Save OMF file

Format: **Data.SAVE.Omf** *<file> <addressrange>* [**/***<option>*]

<option>: *<generic_save_options>*

Saves memory in OMF file format. The command is implemented for the OMF-96 format.

<option> For a description of the generic options, see **<generic_save_options>**.

See also

■ Data.SAVE.<format> ■ Data.LOAD.Omf

Format:	**Data.SAVE.PureHEX** *<file> <addressrange>* [*/<option>*]
<option>:	*<generic_save_options>*

Saves memory in pure HEX file format. The output file includes the saved memory contents as ASCII hexadecimal data. After each 18 characters a CR (0x0D) and a LF (0x0A) characters are added.

<option> For a description of the generic options, see **<generic_save_options>**.

See also

- Data.SAVE.<format>
- ▲ 'Release Information' in 'Legacy Release History'

[Examples]

Format:	**Data.SAVE.S1record** *<file>* *<range>*[||*<ranges>* …] [**/***<option>*]
<option>:	**OFFSET** | **RECORDLEN** | **Append** | **SkipErrors** *<generic_save_options>*

Saves memory content as SREC S19 style / 16-bit address record. The following options are available:

Append	Appends data to an existing file.
OFFSET	Changes the address value to value plus offset. The Srecord will be loaded to the address plus offset value.
RECORDLEN	Defines the number of data bytes per line in the Srecord file. Decimal values have to be given with decimal point behind.
<option>	For a description of the generic options, see **<generic_save_options>**.

Example 1

```
Data.SAVE.S3record mydata.s3 0x1000++0fff
```

Example 2

TRACE32 allows to specify more than one address <*range*>. The ranges are separated by two pipe symbols ||, no space allowed. The example below is for demo purposes only.

```
Data.dump VM:0x1000

;initialize a TRACE32 virtual memory (VM:) area with a test pattern
Data.PATTERN VM:0x1000++3ff /WordCount

;save three non-contiguous ranges to one S3 file
Data.SAVE.S3record ~~~\s3multirange.s3 VM:0x1000++0f||\
VM:0x1040++1f||VM:0x1110++0f

Data.CLEARVM VM:0x1000++0xFFFF ;clear the 1st 64 kB block of the virtual
memory

;load the S3 file back to the virtual memory
Data.LOAD.S3record ~~~\s3multirange.s3 /VM /OFFSET 0x200
```

See also

- Data.SAVE.S2record
- Data.SAVE.S3record
- Data.SAVE.S4record
- Data.SAVE.<format>
- Data.LOAD.S1record
- Data.LOAD.S2record
- Data.LOAD.S3record
- Data.LOAD.S4record
- Data.LOAD.SAUF
- Data.LOAD.SDS
- Data.LOAD.SPARSE
- Data.LOAD.SysRof

Data.SAVE.S2record — Save S2-record file

[Examples]

Format:	**Data.SAVE.S2record** *<file> <range>*[||*<ranges>* …] [**/***<option>*]
<option>:	**OFFSET** | **RECORDLEN** | **Append** | **SkipErrors** *<generic_save_options>*

Saves memory content as SREC S28 style / 24-bit address record. The description of options, further information and examples are here.

See also

- Data.SAVE.S1record
- Data.LOAD.S1record
- Data.LOAD.SAUF
- Data.SAVE.S3record
- Data.LOAD.S2record
- Data.LOAD.SDS
- Data.SAVE.S4record
- Data.LOAD.S3record
- Data.LOAD.SPARSE
- Data.SAVE.<format>
- Data.LOAD.S4record
- Data.LOAD.SysRof

Data.SAVE.S3record — Save S3-record file

[Examples]

Format:	**Data.SAVE.S3record** *<file> <range>*[||*<ranges>* …] [**/***<option>*]
<option>:	**OFFSET** | **RECORDLEN** | **Append** | **SkipErrors** *<generic_save_options>*

Saves memory content as SREC S37 style / 32-bit address record. The description of options, further information and examples are here.

See also

- Data.SAVE.S1record
- Data.LOAD.S1record
- Data.LOAD.SAUF
- Data.SAVE.S2record
- Data.LOAD.S2record
- Data.LOAD.SDS
- Data.SAVE.S4record
- Data.LOAD.S3record
- Data.LOAD.SPARSE
- Data.SAVE.<format>
- Data.LOAD.S4record
- Data.LOAD.SysRof

[Examples]

Format:	**Data.SAVE.S4record** *<file>* *<range>*[||*<ranges>* …] [**/***<option>*]
<option>:	**OFFSET** | **RECORDLEN** | **Append** | **SkipErrors** *<generic_save_options>*

Saves memory content as SREC S47 style / 64-bit address record. The description of options, further information and examples are here.

See also

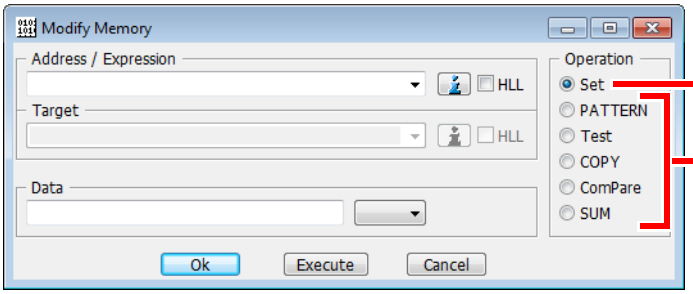
- Data.SAVE.S1record
- Data.SAVE.S2record
- Data.SAVE.S3record
- Data.SAVE.<format>

[Examples]

Format:	**Data.Set** [*<address>* | *<range>*] {[**%***<format>*] *<value>*} [{**/***<option>*}]
<format>:	[*<access_format>*.]*<data_format>* **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] **BE** | **LE** **BitSwap**
<access_format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<data_format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<value>:	*<data>* | "*<string>*" [**0**]
<option>:	**Verify** | **ComPare** | **DIFF** | **PlusVM** | **CORE** *<core_number>*

Write data to memory. If no byte access is possible for an address location, the write is performed in the smallest possible width.

If you run **Data.Set** without command line arguments, then the **Modify Memory** dialog opens.



The data set function may be called by mouse-click (left button) to a data field. By choosing an address range, memory can be filled with a constant.

0	Writes the "*<string>*" as a zero-terminated string to memory.
BE, **LE**	Defines byte-order display direction: Big Endian or Little Endian.
BitSwap	Reverses the bit-order within a byte. This will not affect the byte-order.
Byte (default), **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	Data size for integer or string constants. See **"Keywords for <width>"**, page 11.

ComPare	Compares the data against memory, don't write to memory.
CORE *<number>*	Performs write operation on the specified hardware thread.
DIFF	Data is compared against memory. Memory is not changed. The result of the compare is available in the **FOUND()** function.
Float	Data format for floating point constants.
Verify	Sets and verifies complete block of data by a following read operation.

Example 1

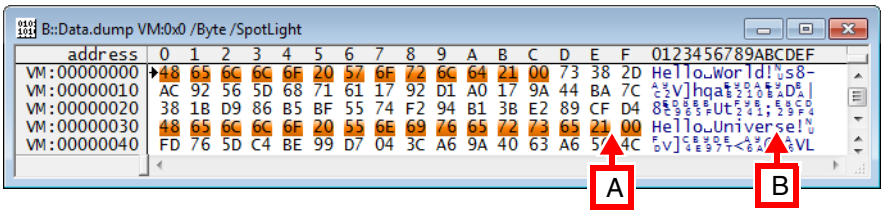
Various **Data.Set** operations with and without the use of PRACTICE functions:

```
Data.Set 0x100 "hello world" 0x0          ; set string to memory

Data.Set 0x100 %Long 0x12345678           ; write long word

Data.Set 0x0--0x0ffff 0x0                 ; init memory with 0

Data.Set Var.RANGE(flags) 0x0             ; fill field with constant

Data.Set 0x100--0x3ff %Long 0x20000000    ; fill with long word

Data.Set 0x100--0x3ff %Word 0x2000 /ComPare

                                          ; verify that memory contains
                                          ; this data

Data.Set 0x100 %Float.IeeeDbl 1.25        ; set floating point in
                                          ; IEEE-Double format

PRINT Data.Short(D:0x200)                 ; prints short at address
PRINT Data.Short(D:0x200)                 ; D:0x200

Data.Set 0x4128 %BE %Byte.Long            ; write the 32-bit <data>
0x12345678                                ; bytewise to memory

                                          ; equivalent command
Data.Set 0x4128 0x12 0x34 0x56 0x78

Data.Set 0x4128 %BE %Byte.Long 0xab       ; write the 32-bit <data>
                                          ; bytewise to memory

Data.Set 0x4128 0x00 0x00 0x00 0xab       ; equivalent command
```

Example 2

Shows how to write a zero-terminated string to memory. In this case, a zero-terminated string is written to the TRACE32 virtual memory.

```
Data.Set VM:0x0  "Hello World!" 0       ;set two zero-terminated strings
Data.Set VM:0x30 "Hello Universe!" 0    ;to the TRACE32 virtual memory
```



A In the byte-formatted output, **00** indicates a zero-terminated string.

B In the ASCII-formatted output, **NU** indicates a zero-terminated string.

Example 3

The **Data.Set** *<addressrange>* **/DIFF** command is used together with the following functions:

FOUND()	Returns TRUE if a difference was found in the comparison.
TRACK.ADDRESS()	Returns the address of the first difference.

```
…
Data.Set 0x0++0xffff 123
Data.Set 0x0++0xffff 123 /DIFF

IF FOUND()
     PRINT "Error found at address " TRACK.ADDRESS()

…
```

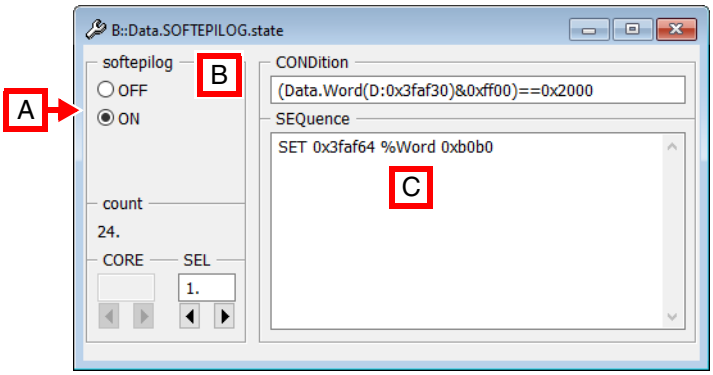
See also

■ Data.Out ■ Data.PATTERN ■ Data.PROGRAM ■ Data.Test

▲ 'Release Information' in 'Legacy Release History'

The **Data.SOFTEPILOG** command group allows to define a sequence of read/write accesses that are automatically performed directly after a **software** breakpoint is written to the memory by the debugger. The complementary command **Data.SOFTPROLOG** performs read/write accesses before a software breakpoint is written to the memory.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **Data.SOFTEPILOG.state** window.



A For descriptions of the commands in the **Data.SOFTEPILOG.state** window, please refer to the **Data.SOFTEPILOG.*** commands in this chapter. **Example**: For information about **ON**, see **Data.SOFTEPILOG.ON**.

B Conditions can be set up in the **CONDition** field using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** field using the *<data_set_commands>* **SET**, **SETI**, **GETS**, and **SETS**.

Data.SOFTEPILOG.CONDition — Define condition for data softepilog

Format:	**Data.SOFTEPILOG.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **Data.SOFTEPILOG.SEQuence** will be executed directly after a software breakpoint is written to the memory.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

Data.SOFTEPILOG.CORE — Select core for data softepilog

Format:	**Data.SOFTEPILOG.CORE** *<core_number>*

Selects the core for which you want to define one or more data softepilogs.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: The following example shows how to define a data softepilog that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a data softepilog
Data.SOFTEPILOG.CORE 3.

;Define the data softepilog for core 3
Data.SOFTEPILOG.CONDition <your_code>
Data.SOFTEPILOG.SEQuence  <your_code>
```

For information on how to configure two different data softepilogs, see **Data.SOFTEPILOG.SELect**.

Data.SOFTEPILOG.OFF Switch data softepilog off

Format: **Data.SOFTEPILOG.OFF**

Disables the execution of the **Data.SOFTEPILOG** sequence.

Data.SOFTEPILOG.ON Switch data softepilog on

Format: **Data.SOFTEPILOG.ON**

Enables the execution of the **Data.SOFTEPILOG** sequence.

Data.SOFTEPILOG.RESet Reset all data softepilogs

Format: **Data.SOFTEPILOG.RESet**

Switches the **Data.SOFTEPILOG** feature off and clears all settings.

Data.SOFTEPILOG.SELect Increment the index number to the next epilog

Format: **Data.SOFTEPILOG.SELect** *<index_number>*

Increments the index number for each new data softepilog. This is useful, for example, if you need two separate data softepilogs with each data softepilog having its own **Data.SOFTEPILOG.CONDition**.

TRACE32 automatically assigns the index number 1. to the 1st **Data.SOFTEPILOG.SEQuence**. If you require a 2nd, separate data softepilog sequence, then increment the *<index_number>* to 2. Otherwise the 2nd data softepilog will overwrite the 1st data softepilog. You can define a maximum of 10 data softepilogs.

Format: **Data.SOFTEPILOG.SEQuence** *<command>* …

<command>: **SET** *<address>* **%***<format>* *<data>*
SETI *<address>* **%***<format>* *<data>* *<increment>*
SETS *<address>*
GETS *<address>*

Defines a sequence of **Data.Set** commands that are automatically executed by the TRACE32 software directly after writing the software breakpoint into the memory.

SET Parameters: *<address>* **%***<format>* *<value>*
Write *<value>* with data type *<format>* to *<address>*

SETI Parameters: *<address>* **%***<format>* *<start>* *<increment>*
At the first time performed, write *<start>* to *<address>*.
<start> is incremented by *<increment>* on each successive call.

GETS Parameters: *<address>* **%***<format>*
Reads the value at *<address>* and stores it into an internal data buffer. The internal data buffer can contain multiple records and is reset when the command **Data.SOFTEPILOG.Sequence** is called.

SETS Parameters: *<address>* **%***<format>*
If the internal data buffer contains a record for *<address>*, the stored value is written to the processor.

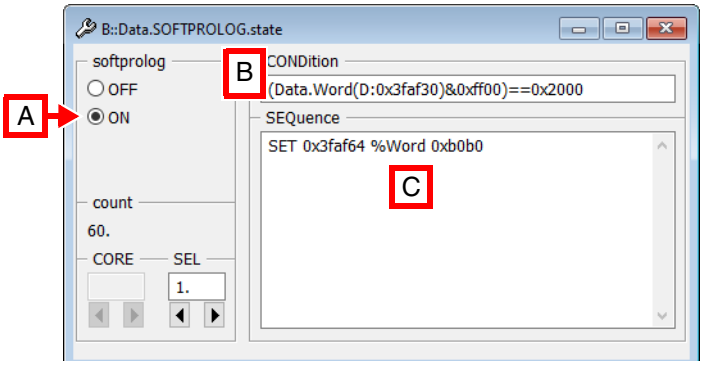
Data.SOFTEPILOG.state Display data softepilogs

Format: **Data.SOFTEPILOG.state**

Opens the **Data.SOFTEPILOG.state** window, where you can configure data softepilogs.

The **Data.SOFTPROLOG** command group allows to define a sequence of read/write accesses that are automatically performed before a **software** breakpoint is written to the memory by the debugger. The complementary command **Data.SOFTEPILOG** performs read/write accesses directly after a software breakpoint is written to the memory.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **Data.SOFTPROLOG.state** window.



A For descriptions of the commands in the **Data.SOFTPROLOG.state** window, please refer to the **Data.SOFTPROLOG.*** commands in this chapter. **Example**: For information about **ON**, see **Data.SOFTPROLOG.ON**.

B Conditions can be set up in the **CONDition** field using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** field using the *<data_set_commands>* **SET**, **SETI**, **GETS**, and **SETS**.

Data.SOFTPROLOG.CONDition Define condition for data softprolog

Format:	**Data.SOFTPROLOG.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **Data.SOFTPROLOG.SEQuence** will be executed before a software breakpoint is written to the memory.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

Format: **Data.SOFTPROLOG.CORE** *<core_number>*

Selects the core for which you want to define one or more data softprologs.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: The following example shows how to define a data softprolog that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a data softprolog
Data.SOFTPROLOG.CORE 3.

;Define the data softprolog for core 3
Data.SOFTPROLOG.CONDition <your_code>
Data.SOFTPROLOG.SEQuence  <your_code>
```

For information on how to configure two different data softprologs, see **Data.SOFTPROLOG.SELect**.

Data.SOFTPROLOG.OFF Switch data softprolog off

Format: **Data.SOFTPROLOG.OFF**

Disables the execution of the **Data.SOFTPROLOG** sequence.

Data.SOFTPROLOG.ON Switch data softprolog on

Format: **Data.SOFTPROLOG.ON**

Enables the execution of the **Data.SOFTPROLOG** sequence.

Format: **Data.SOFTPROLOG.RESet**

Switches the **Data.SOFTPROLOG** feature off and clears all settings.

Data.SOFTPROLOG.SELect Increment the index number to the next prolog

Format: **Data.SOFTPROLOG.SELect** *<index_number>*

Increments the index number for each new data softprolog. This is useful, for example, if you need two separate data softprolog with each data softprolog having its own **Data.SOFTPROLOG.CONDition**.

TRACE32 automatically assigns the index number 1. to the 1st **Data.SOFTPROLOG.SEQuence**. If you require a 2nd, separate data softprolog sequence, then increment the *<index_number>* to 2. Otherwise the 2nd data softprolog will overwrite the 1st data softprolog. You can define a maximum of 10 data softprolog.

Data.SOFTPROLOG.SEQuence Define softprolog sequence

Format: **Data.SOFTPROLOG.SEQuence** *<command>* …

<command>:
SET *<address>* **%***<format>* *<data>*
SETI *<address>* **%***<format>* *<data>* *<increment>*
SETS *<address>*
GETS *<address>*

Defines a sequence of **Data.Set** commands that are automatically executed by the TRACE32 software before writing the software breakpoint into the memory.

SET Parameters: *<address>* **%***<format>* *<value>*
Write *<value>* with data type *<format>* to *<address>*

SETI Parameters: *<address>* **%***<format>* *<start>* *<increment>*
At the first time performed, write *<start>* to *<address>*.
<start> is incremented by *<increment>* on each successive call.

GETS Parameters: *<address>* **%***<format>*
Reads the value at *<address>* and stores it into an internal data buffer.
The internal data buffer can contain multiple records and is reset when the command **Data.SOFTPROLOG.Sequence** is called.

SETS Parameters: *<address>* **%***<format>*
If the internal data buffer contains a record for *<address>*, the stored value is written to the processor.

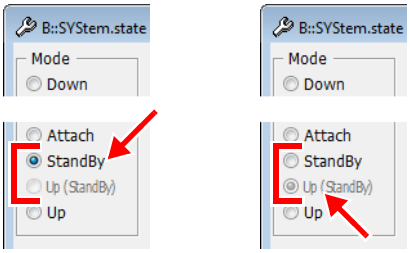
Data.SOFTPROLOG.state — Display data softprologs

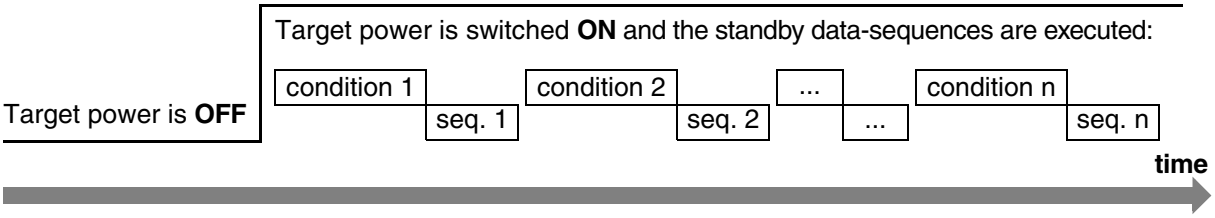
Format: **Data.EPILOG.state**

Opens the **Data.SOFTPROLOG.state** window, where you can configure data softprologs.

Using the **Data.STANDBY** command group, you can define one or more sequences that perform write operations to registers or memory. For example, you can use the **Data.STANDBY** command group to deactivate a watchdog.

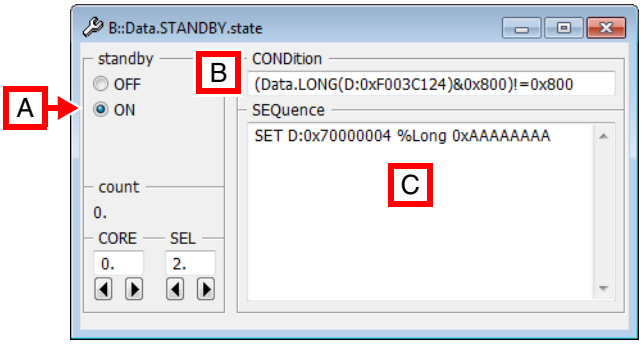
These standby data-sequences are executed automatically as soon as target power is switched on while the debugger is in **StandBy** mode (**SYStem.Mode StandBy**):





Each sequence can optionally depend on a condition, see also [**B**] in the figure below.

For configuration of standby data-sequences, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **Data.STANDBY.state** window:



A For descriptions of the commands in the **Data.STANDBY.state** window, please refer to the **Data.STANDBY.*** commands in this chapter.
Example: For information about **ON**, see **Data.STANDBY.ON**.

B Simple conditions can be set up in the **CONDition** field using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Standby data-sequences can be set up in the **SEQuence** field using the memory modification commands **SET**, **SETI**, **GETS**, and **SETS**.

A good way to familiarize yourself with the **Data.STANDBY** command group is to start with the following example.

Example

This demo script illustrates how to define standby data-sequences for two cores.

```
Data.STANDBY.state              ;optional step: open the window

Data.STANDBY.CORE   0           ;let's define two sequences for core 0
Data.STANDBY.SELect 1           ;sequence 1 on core 0

;no condition is specified for sequence 1 on core 0:
Data.STANDBY.SEQuence SET D:0x70000000 %Long 0x55555555

;set the index number to 2, else the first sequence would be
;overwritten by the 2nd sequence
Data.STANDBY.SELect 2

;the 2nd sequence consisting of two SET sequences shall be executed
;only if this condition is true:
Data.STANDBY.CONDition (Data.LONG(D:0xF003C124)&0x800)!=0x800

;define the two SET sequences for the condition above:
Data.STANDBY.SEQuence SET D:0x70000004 %Long 0xAAAAAAAA \
                      SET D:0x70000014 %Long 0xBBBBBBBB

Data.STANDBY.CORE   1           ;let's define a sequence for core 1
Data.STANDBY.SELect 1           ;sequence 1 on core 1

;no condition is specified for sequence 1 on core 1:
Data.STANDBY.SEQuence SET D:0x70000010 %Long 0x11111111

Data.STANDBY.ON                 ;we are now ready to activate the
                                ;standby data-sequences

SYStem.Mode StandBy             ;switch to StandBy mode
```

As soon as target power is switched **ON**, the standby data-sequences are executed.

See also

- Data.STANDBY.CONDition
- Data.STANDBY.CORE
- Data.STANDBY.OFF
- Data.STANDBY.ON
- Data.STANDBY.RESet
- Data.STANDBY.SELect
- Data.STANDBY.SEQuence
- Data.STANDBY.state
- Data.STARTUP

▲ 'Release Information' in 'Legacy Release History'

Only for PowerPC MPC5xxx, TriCore

Format:	**Data.STANDBY.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which a standby data-sequence will be executed automatically. To define the standby data-sequence, use the command **Data.STANDBY.SEQuence**.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

Example:

```
;reads the long at address 0xF003C124. If the result is not equal to
;0x800,...
Data.STANDBY.CONDition (Data.LONG(D:0xF003C124)&0x800)!=0x800

;... then the standby data-sequence is executed.
Data.STANDBY.SEQuence SET D:0x70000004 %Long 0xAAAAAAAA
```

See also

- Data.STANDBY
- Data.STANDBY.state

Data.STANDBY.CORE — Assign sequence to core

Only for PowerPC MPC5xxx, TriCore

Format: **Data.STANDBY.CORE** *<core_number>*

Selects the core for which you want to define one or more standby data-sequences.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: The following example shows how to define a standby data-sequences that is executed on core 3 of a multicore chip.

```
;select the core for which you want to define a standby data-sequence
Data.STANDBY.CORE 3.

;define the standby data-sequence for core 3
Data.STANDBY.CONDition <your_code>
Data.STANDBY.SEQuence  <your_code>
```

For information on how to configure two separate standby data-sequences, see **Data.STANDBY.SELect**.

See also

- Data.STANDBY
- Data.STANDBY.state

Data.STANDBY.OFF — Switch all sequences off

Only for PowerPC MPC5xxx, TriCore

Format: **Data.STANDBY.OFF**

Switches the **Data.STANDBY** feature off.

See also

- Data.STANDBY
- Data.STANDBY.state

Data.STANDBY.ON — Switch all sequences on

Only for PowerPC MPC5xxx, TriCore

Format: **Data.STANDBY.ON**

Switches the **Data.STANDBY** feature on.

See also

- Data.STANDBY
- Data.STANDBY.state

Data.STANDBY.RESet — Clear all settings

Only for PowerPC MPC5xxx, TriCore

Format: **Data.STANDBY.RESet**

Switches the **Data.STANDBY** feature off and clears all settings.

See also

- Data.STANDBY
- Data.STANDBY.state

Only for PowerPC MPC5xxx, TriCore

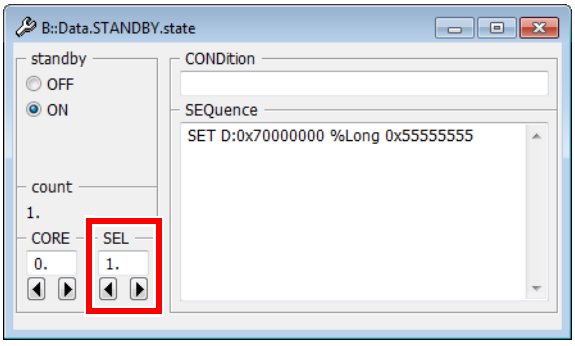
Format: **Data.STANDBY.SELect** *<index_number>*

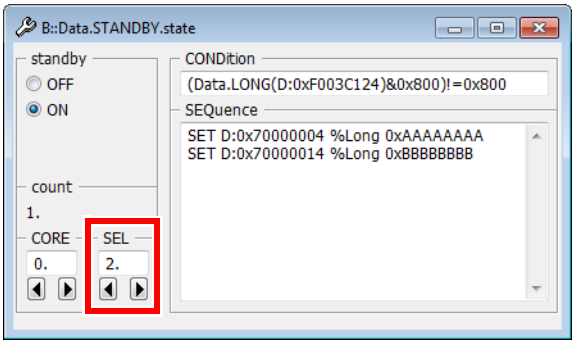
Selects the sequence that is configured by the subsequent **Data.STANDBY.*** commands. This is useful, for example, when you are working with multiple sequences and conditions.

Sequence **1.** is automatically selected after start-up. You can define up to 10 sequences.

Example: This script defines two separate sequences with the index numbers **1.** and **2.**. Sequence 2, in turn, consists of two **SET** sequences that depend on the *same* condition. The backslash **** is used as a line continuation character. No white space permitted after the backslash.

```
;optional step for you: set the index number to 1.
Data.STANDBY.SELect 1.

;no condition is specified for this sequence:
Data.STANDBY.SEQuence SET D:0x70000000 %Long 0x55555555

;set the index number to 2, else the first sequence would be
;overwritten by the 2nd sequence
Data.STANDBY.SELect 2.

;the 2nd sequence consisting of two SET sequences shall be executed
;only if this condition is true:
Data.STANDBY.CONDition (Data.LONG(D:0xF003C124)&0x800)!=0x800

;define the two SET sequences for the condition above:
Data.STANDBY.SEQuence SET D:0x70000004 %Long 0xAAAAAAAA \
                      SET D:0x70000014 %Long 0xBBBBBBBB
```





See also

- Data.STANDBY
- Data.STANDBY.state

Only for PowerPC MPC5xxx, TriCore

Format:	**Data.STANDBY.SEQuence** *<command>* …
<command>:	**SET** *<address>* **%***<format>* *<data>* **SETI** *<address>* **%***<format>* *<data>* *<increment>* **SETS** *<address>* **GETS** *<address>*

Defines a standby data-sequence consisting of memory modification commands that are automatically executed when TRACE32 leaves the **StandBy** mode and switches to **SYStem.Mode Up (StandBy)**.

SET	Write *<data>* to *<address>*.
SETI	Write *<data>* to *<address>*. Then *<data>* is incremented by *<increment>*.
GETS	Save the data at *<address>*.
SETS	Write the data that was saved with a previous GETS back to *<address>*.

Example: This script defines a standby data-sequence consisting of two SET sequences. The backslash **** is used as a line continuation character. No white space permitted after the backslash.

```
Data.STANDBY.SEQuence SET D:0x70000004 %Long 0xAAAAAAAA \
                      SET D:0x70000014 %Long 0xBBBBBBBB
```

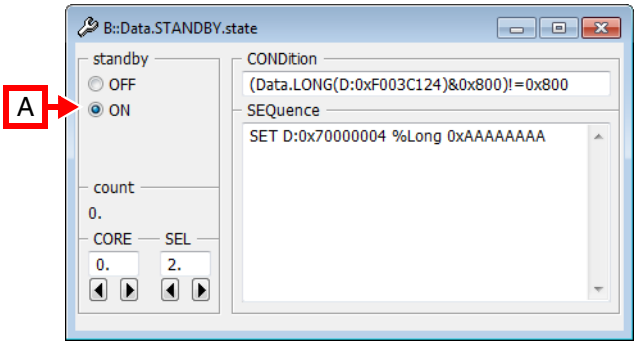
See also

- Data.STANDBY
- Data.STANDBY.state

Only for PowerPC MPC5xxx, TriCore

Format:	**Data.STANDBY.state**

Opens the **Data.STANDBY.state** window, where you can configure standby data-sequences.



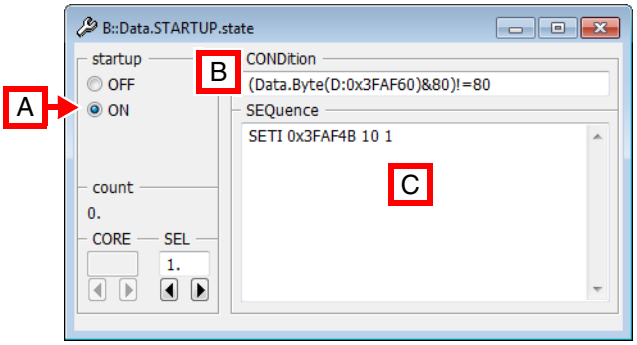
A For descriptions of the commands in the **Data.STANDBY.state** window, please refer to the **Data.STANDBY.*** commands in this chapter.
Example: For information about **ON**, see **Data.STANDBY.ON**.

See also

- Data.STANDBY
- Data.STANDBY.CONDition
- Data.STANDBY.CORE
- Data.STANDBY.OFF
- Data.STANDBY.ON
- Data.STANDBY.RESet
- Data.STANDBY.SELect
- Data.STANDBY.SEQuence

The **Data.STARTUP** command group allows to define a sequence of **Data.Set** commands that are executed when the debugger is activated with **SYStem.Mode Up**.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **Data.STARTUP.state** window:



A For descriptions of the commands in the **Data.STARTUP.state** window, please refer to the **Data.STARTUP.*** commands in this chapter.
Example: For information about **ON**, see **Data.STARTUP.ON**.

B Conditions can be set up in the **CONDition** field using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** field using the *<data_set_commands>* **SET**, **SETI**, **GETS**, and **SETS**.

See also

- Data.STARTUP.CONDition
- Data.STARTUP.CORE
- Data.STARTUP.OFF
- Data.STARTUP.ON
- Data.STARTUP.RESet
- Data.STARTUP.SELect
- Data.STARTUP.SEQuence
- Data.STARTUP.state
- Data.STANDBY
- Data.ATTACH
- Data.EPILOG
- Data.PROLOG

Format:	**Data.STARTUP.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **Data.STARTUP.SEQuence** will be executed periodically.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

Examples:

```
; reads the long at address D:0x3faf30, proceeds a binary AND with
; a constant (here 0xffffffff). If the result is equal to 0x80000000 the
; condition is true and the defined sequence is executed.
Data.STARTUP.CONDition (Data.Long(D:0x3faf30)&0xffffffff)==0x80000000
```

```
; read the word at address D:0x3xfaf30
Data.STARTUP.CONDition (Data.Word(D:0x3faf30)&0xff00)!=0x8000
```

```
; reads the byte at address D:0x3xfaf30
Data.STARTUP.CONDition (Data.Byte(D:0x3faf30)&0xf0)!=0x80
```

See also

- Data.STARTUP
- Data.STARTUP.state

Format: **Data.STARTUP.CORE** *<core_number>*

Selects the core for which you want to define one or more data startup sequences.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: This script shows how to define a startup sequence that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a startup sequence
Data.STARTUP.CORE 3.

;Define the startup sequence for core 3
Data.STARTUP.CONDition <your_code>
Data.STARTUP.SEQuence  <your_code>
```

For information on how to configure two different startup sequences, see **Data.STARTUP.SELect**.

See also

- Data.STARTUP
- Data.STARTUP.state

Data.STARTUP.OFF — Switch startup sequence off

Format: **Data.STARTUP.OFF**

Switches the **Data.STARTUP** feature off.

See also

- Data.STARTUP
- Data.STARTUP.state

Data.STARTUP.ON — Switch startup data sequence on

Format: **Data.STARTUP.ON**

Switches the **Data.STARTUP** feature on.

See also

- Data.STARTUP
- Data.STARTUP.state

Data.STARTUP.RESet — Reset startup data sequence

Format: **Data.STARTUP.RESet**

Switches the **Data.STARTUP** feature off and clears all settings.

See also

- Data.STARTUP
- Data.STARTUP.state

Format: **Data.STARTUP.SELect** *<serial_number>*

Increments the index number for each new startup sequence. This is useful, for example, if you need two separate startup sequences with each sequence having its own **Data.STARTUP.CONDition**.

TRACE32 automatically assigns the index number 1. to the first **Data.STARTUP.SEQuence**. If you require a second, separate startup sequence, then increment the *<index_number>* to 2. Otherwise the second startup sequence will overwrite the first one. You can define a maximum of 10 startup sequences.

Example 1: Two startup sequences with the *same* **Data.STARTUP.CONDition** may have the *same* index number. The backslash **** is used as a line continuation character. No white space permitted after the backslash.

```
;Set the index number to 1.
Data.STARTUP.SELect 1.

;Startup sequences shall be executed only if this condition is true:
Data.STARTUP.CONDition (Data.Word(D:0x4faf34)&0xff00)==0x4000

;Define the two startup sequences:
Data.STARTUP.SEQuence SET 0x4faf54 %Word 0xC0C0 \
                      SET 0x4faf64 %Word 0xD0D0
```

Example 2: Two startup sequences with *different* **Data.STARTUP.CONDition** settings require two *different* index numbers.

```
;1st startup sequence - TRACE32 automatically sets the index number to 1.
Data.STARTUP.SELect 1.

;If this startup condition is true, ...
Data.STARTUP.CONDition (Data.Word(D:0x4faf38)&0xff00)==0x2000

;... then the 1st startup sequence will be executed
Data.STARTUP.SEQuence SET 0x4faf58 %Word 0xE0E0

;Increment the index number to define the 2nd startup sequence
Data.STARTUP.SELect 2.

;If this data startup condition is true, ...
Data.STARTUP.CONDition (Data.Word(D:0x4faf38)&0xff00)==0x3000

;... then the 2nd startup sequence will be executed
Data.STARTUP.SEQuence SET 0x4faf58 %Word 0xF0F0
```

See also

- Data.STARTUP
- Data.STARTUP.state

Format:	**Data.STARTUP.SEQuence** *<command>* …
<command>:	**SET** *<address>* **%***<format>* *<data>* **SETI** *<address>* **%***<format>* *<data>* *<increment>* **SETS** *<address>* **GETS** *<address>*

Defines a sequence of **Data.Set** commands that are executed when the emulation system in activated.

SET	Write *<data>* to *<address>*.
SETI	Write *<data>* to *<address>*. Then *<data>* is incremented by *<increment>*.
GETS	Save the data at *<address>*.
SETS	Write the data that was saved with a previous GETS back to *<address>*.

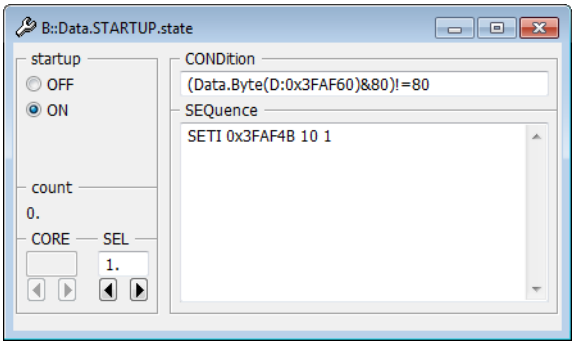
Examples:

```
Data.STARTUP.SEQuence SET 0x3faf50 %Word 0xa0a0

Data.STARTUP.SEQuence SETI 0x3faf50 %Word 0xa0a0 2

Data.STARTUP.SEQuence SETS 0x3faf60

Data.STARTUP.SEQuence GETS 0x3faf60
```

See also

- Data.STARTUP
- Data.STARTUP.state

Format: **Data.STARTUP.state**



See also

- Data.STARTUP
- Data.STARTUP.CONDition
- Data.STARTUP.CORE
- Data.STARTUP.OFF
- Data.STARTUP.ON
- Data.STARTUP.RESet
- Data.STARTUP.SELect
- Data.STARTUP.SEQuence

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.STRING** [**%CONTinue**] *<addressrange>*

Displays a string in the selected **AREA** window. The character set is host specific.

CONTinue Adds the string to the current output line in the selected **AREA** window without inserting a newline character.

Example:

```
AREA.Create TERMINAL                    ; create an area
AREA.Select TERMINAL                    ; select it for output and input
AREA TERMINAL                           ; show the area in a window
Data.STRING SD:0x1000--0x1fff           ; display data in the window
```

See also

■ Data.dump ■ Data.WRITESTRING ❑ Data.STRing()

▲ 'Release Information' in 'Legacy Release History'

Data.SUM Memory checksum

[Examples]

Format: **Data.SUM** *<range>* [*/<format>*]

<format>: **Byte** | **Word** | **Long**
RotByte | **RotWord** | **RotLong**
XorWord9 | **RotWord9**
CRC16 | **CRC32** | **CRC**
Even | **Odd**
InvByte | **InvWord** | **InvLong**
ByteSWAP

The *format* option allows to select an algorithm to determine the check sum. The default setting is **XorWord9**. The resulting checksum is available for PRACTICE by the **Data.SUM()** function.

Checksum	Algorithms
Byte	Sum bytes (returns a 32-bit checksum)
Word	Sum words (returns a 32-bit checksum)
Long	Sum longs (returns a 32-bit checksum)
RotByte	Sum byte and rotate byte left circular (returns a 8-bit checksum)
RotWord	Sum word and rotate word left circular (returns a 16-bit checksum)
RotLong	Sum long and rotate long left circular (returns a 32-bit checksum)
CRC16	16-bit CRC algorithm (ZMODEM variant)
CRC32	32-bit CRC algorithm (PKZIP variant)
CRC *<params>*	Usage: /CRC *<checksum_width> <polynom>* *<input reflection 0|1> <output reflection 0|1>* *<CRC init> <output XOR value>* *<polynom_width>* : Size of CRC Checksum in bits *<polynom>* : Polynom (without MSB which has to be 1). *<input_reflection>* : if not zero, input will be reflected (LSB of a byte will be processed first) *<output_reflection>*: if not zero, output will be reflected (bit ordering will be reversed) *<crc_init>* : Initial value, which is XORed to input *<output_xor>* : Value which will be XORed to output /CRC16 is a synonym for /CRC 16. 0x1021 0 0 0x0 0x0 The official CRC16 CCITT algorithm should be the same as /CRC 16. 0x1021 1 1 0x0 0x0 /CRC32 is a synonym for /CRC 32. 0x04C11DB7 1 1 0xFFFFFFFF 0xFFFFFFFF This is the CRC used in the various “ZIP” utilities. The Unix “cksum” utility uses the same as /CRC 32. 0x04C11DB7 0 0 0x0 0xFFFFFFFF **NOTE**: The cksum utility adds the length of the text in little endian before calculating the CRC. If you want to get the same result you need to emulate this behavior by also adding the length of the text to the end of the text.
Even	Sum even bytes (returns a 32-bit checksum)

Odd	Sum odd bytes (returns a 32-bit checksum)
RotWord9	Rotate words special (OS-9 compatible) (returns a 16-bit checksum)
XorWord9	exor of all words, start-value = 0xffff (OS-9 compatible) (returns a 16-bit checksum)
InvByte	Sum bitwise inverted bytes (returns a 32-bit checksum)
InvWord	Sum bitwise inverted words (returns a 32-bit checksum)
InvLong	Sum bitwise inverted longs (returns a 32-bit checksum)
ByteSWAP	Swap the endianness before adding the bytes. This allow to calculate a little-endian checksum on a big-endian system and vice versa.

Examples:

```
; checksum over EPROM
…
Data.SUM 0x0--0x0ffff
IF Data.SUM()!=3426
     STOP "Error Eprom"
…
```

```
; checksum across memory, OS-9 compatible
Data.SUM 0x1002--0x1bff /XorWord9
```

```
; calculate a 32-bit check sum, byte summarizing the memory contents
; bytewise
Data.SUM 0x0--0x1fffb /Byte

; place resulting checksum in memory
Data.Set 0x1fffc %Long Data.SUM()
```

See also

■ Data.Test ❑ Data.SUM()

Format: **Data.TABle** *<base>*|*<range> <size> <element>* [*<element>*...] [**/***<option>* …]

<element>: [**%***<format>*] [*<offset>*|*<offrange>*]

<format>:
Decimal [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
DecimalU [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Hex [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
HexS [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
OCTal [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Ascii [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Binary [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Float[.*<float_rep>*[.*<endianness>*]]
sYmbol [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Var
DUMP [.*<width>* [.*<endianness>* [.*<bitorder>*]]]
Byte [.*<endianness>* [.*<bitorder>*]]
Word [.*<endianness>* [.*<bitorder>*]]
Long [.*<endianness>* [.*<bitorder>*]]
Quad [.*<endianness>* [.*<bitorder>*]]
TByte [.*<endianness>* [.*<bitorder>*]]
PByte [.*<endianness>* [.*<bitorder>*]]
HByte [.*<endianness>* [.*<bitorder>*]]
SByte [.*<endianness>* [.*<bitorder>*]]

<width>: **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

<endianness>: **DEFault** | **LE** | **BE**

<bitorder>: **DEFault** | **BitSwap**

<option>:
CORE *<core_number>*
COVerage
CTS
Track
FLAG *<flag>*
CFlag *<cflag>*
Mark *<break>*

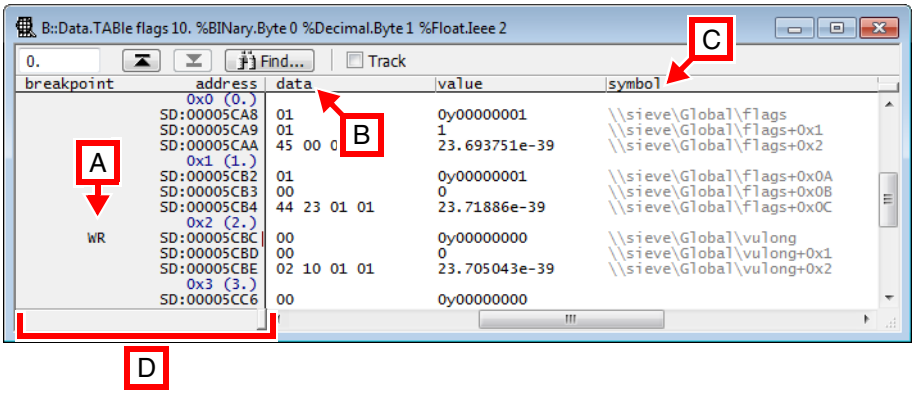
<flag>:	**Read** | **Write** | **NoRead** | **NoWrite**
<cflag>:	**OK** | **NoOK** | **NOTEXEC** | **EXEC**
<break>:	**Program** | **Hll** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

Displays an array without high-level information. If an address is given, it will specify the base address of an array of unlimited size. A range specifies an array of limited size.

<base>, <range>	Base address or address range of data structure
<size>	Size in bytes of a single element of the data structure
<offset> *<offrange>*	relative byte offset of the element
Decimal, DecimalU,...	Refer to **"Keywords for <format>"**, page 10
Byte, Word, ...	Refer to **"Keywords for <width>"**, page 11
DEFault, BE, LE	Define byte-order display direction: default target endianness, Big Endian or Litte Endian
DEFault, BitSwap	**BitSwap** allows to display data in reverse bit-order in each byte. If **BitSwap** is used together with BE or LE, the byte order will not change, otherwise **BitSwap** will also reverse the byte-order.
CORE *<core>*	Display memory from the perspective of the specified core /SMP debugging only).
COVerage	Highlight data memory locations that have never been read/written.
Track	Track the window to the reference address of other windows.
Mark *<break>*	Highlight memory locations for which the specified breakpoint is set.
CTS	Display **CTS** access information when CTS mode is active.

```
; Displays an array starting at symbol 'xarray' with the size of 14 bytes
; for each element.
; The first two long-words are display in hexadecimal.
; The next two bytes as word in decimal and the last four bytes are
; assumed to be an IEEE floating point number.

Data.TABle xarray 14. %Hex.Long 0x0--0x7 %Decimal.Word 0x8 %Float.Ieee
0x0a
```

Sample window for displaying an array.



A Read/Write breakpoint (created with **Break.Set**).

B Hex data.

C Symbolic address.

D Scale area.

The scale area contains Flag and Breakpoint information, memory classes and addresses. The state line displays the currently selected address, both in hexadecimal and symbolic format. By double-clicking a data word, a **Data.Set** command can be executed on the current address.

By holding down the right mouse button, the most important memory functions can be executed via the **Data Address** pull-down menu. If the **Mark** option is on, the relevant bytes will be highlighted. For more information, see **Data.dump**.

See also

- Data.CHAIN
- Data.dump
- Data.Print
- Data.View

Format:	**Data.TAG** *<address> <patcharea> <tagarea>* [**/INTR**]

The command patches binary code to generate one tag for statistical analysis. Similar to **Data.TAGFunc** command, but generates no symbols and no breakpoints.

See also

- Data.TAGFunc

Data.TAGFunc

Format:	**Data.TAGFunc** [*<group>* | *<range>*] *<patch>* [*<tags>*] [**/***<options>*]
<tags>:	*<tags_entry>* [*<tags_exit>* [*<tags_parameter>*]]
<options>:	**INTR** **Parameter**

The command patches binary code to generate the tags required for statistical performance analysis (e.g. **Analyzer.STATistic.Func**) or function parameter trace and trigger. The optional group argument defines which modules or programs should be modified. The extra code generated is placed within the range defined by patch area. The tags parameter define the placement of the tag variables. Without this argument the tags will be placed at the end of the patch area. On processors with data cache the tag variables must be placed in a not cached area. When a second tag range is defined, the exit point tags are placed in an extra memory area. The third tag range is used for the parameter tags (if used). The command generates also the required breakpoints and symbols for the analysis. Functions are only patched if there is enough space for the modification. Depending on the processor different strategies are used to jump from the program to the patch area. Placing the patch area at a location that can be reached by short branches or jumps can result in more possible patches. Functions which can't be patched are listed in the **AREA** window. The **INTR** option marks the functions as interrupt functions for the statistic analysis. The **Parameter** option generates tags to trace or trigger on function parameters and return values. The patches and symbols generated by this command can be removed by the **Data.UNTAGFunc** command.

This command can be used for the following features:

- Detailed performance analysis with pipelined CPUs. This avoids the prefetching problem.
- Performance analysis with instruction caches enabled. The tags must be placed into a non-cached area in this case.
- Function parameter trace and trigger. Traces all function parameters and return values. The tags can also be used to trigger on specific parameter values or return values. This also adds a system call parameter trace to procedure based operating systems (when the kernel routines are tagged).
- Function call and parameter history. The last parameters and return values for each function can be viewed. This feature is also possible with the low cost BDM/Monitor debuggers.

```
Data.LOAD.Ieee mccp.x /Puzzled
; load the application
Data.TAGFunc , 0x08000--0x0bfff /Parameter
; modify the whole program
Analyzer.ReProgram perf
; program the analyzer
Go
; start measurement
…

Break
; stop measurement
Analyzer.STATistic.TREE
; display results (call tree form)
Analyzer.List FUNCVar TIme.REF
; display parameters (nesting)
```

```
Data.TAGFunc int0--int10 0x8000--0x8fff 0x10000--0x100ff /INTR
; tag interrupts
Data.TAGFunc main--last 0x9000--0xffff 0x10100--0x1ffff
; tag regular funcs
```

See also

- Data.TAG
- Data.UNTAGFunc

[Examples]

Format:	**Data.Test** *<address_range>* [**/***<option>*]
<option>:	**Toggle** **Prime** **RANDOM** | **PRANDOM** **AddrBus** **Repeat** [*<count>*] **WriteOnly** | **ReadOnly** **NoBreak** **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

Performs an integrity test of the memory in the specified *<address_range>* and prints a message indicating success or failure of the test.

Depending on the options, the test detects problems with:

- Read and/or write accesses
- Address line failures
- Aliases addresses (mapping addresses beyond the capacity of a memory to low addresses)

The memory test can be aborted at any time by pressing the **STOP** button on the TRACE32 main toolbar.

NOTE:	The **Data.Test** command is not meant to detect where the target system has implemented memory. Only use it as a pure integrity check.

Toggle (default)	Memory contents are read in one block at a time, and inverted twice, thereby not altering memory contents. **NOTE**: problems with aliases addresses are not detected by this test.
Prime	The defined range is completely filled with a test pattern and is subsequently verified. **NOTE**: The length of the test pattern is a prime, but not the data itself. Original memory contents are lost. This test detects address line failures or mirrored partitions within a memory. Can be combined with **WriteOnly** or **ReadOnly**.
RANDOM	Pattern is a random sequence.

PRANDOM	Pattern is a pseudo random sequence. Can be combined with **WriteOnly** or **ReadOnly**.
Repeat	Memory test is repeated several times. If no parameter is used, the test continues to repeat until stopped manually.
WriteOnly	Memory write only.
ReadOnly	Memory read only.
NoBreak	Even in the case of memory error, the memory test does not abort.
Byte, **Word**, …	Specify memory access size. See **"Keywords for <width>"**, page 11. If no access size is specified, the debugger uses the optimum size for the processor architecture.

The options **WriteOnly** and **ReadOnly** are useful if there are additional operations to be performed between writing and reading (verification), e.g.

- Changing the configuration of a memory controller, e.g. for a different access timing
- Enabling the read access
- Programming the FLASH memory (see example below)

To ensure that the read data is verified with the corresponding write data, **WriteOnly** and **ReadOnly** can only be combined with options that generate predictable data, e.g. **Prime** and **PRANDOM**.

Examples:

```
Data.Test 0x0--0x0ffff /Prime                ; Memory test where the
                                             ; length of the test pattern
                                             ; is a prime.

Data.Test 0x0--0x0ffff /Repeat               ; Memory test until memory
                                             ; error occurs or abort by
                                             ; means of keyboard.

Data.Test 0x0--0x0ffff /Prime /Repeat 3.     ; Memory test where the
                                             ; length of the test pattern
                                             ; is a prime.
                                             ; The memory test is
                                             ; repeated 3 times.
```

Test for FLASH memory (write and read-back identical pattern).

```
…                                            ; FLASH declaration

FLASH.ReProgram ALL

Data.Test 0x0--0x0ffff /WriteOnly /Prime     ; make only write cycles
```

```
FLASH.ReProgram off

Data.Test 0x0--0x0ffff /ReadOnly /Prime      ; make only read cycles

…
```

The **Data.Test** command affects the following functions:

FOUND()	Returns TRUE if a memory error was found.
TRACK.ADDRESS()	Returns the address of the first error.

```
…

Data.Test 0x0++0xffff /Prime

IF Found()
     PRINT "Error found at address " TRACK.ADDRESS()

…
```

See also

- Data.dump
- Data.Out
- Data.PATTERN
- Data.Set
- Data.SUM
- FLASHFILE.TEST
- SETUP.TIMEOUT
- FOUND()

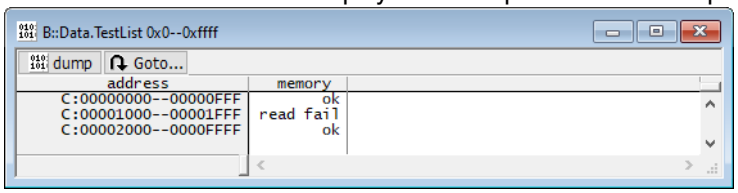
▲ 'Release Information' in 'Legacy Release History'

Format: **Data.TestList** [*<address_range>*] [*/<option>* …]

<option>: **64K** | **1M**

Data.TestList is non-destructive test to find out which memory type is at which address in your target. By default, the smallest resolution is 4K. By choosing an *<option>*, the specified overall *<address_range>* is divided into 64K or 1M sized ranges of which only the first 16K are tested.

The **Data.TestList** window displays one line per result for the specified *<address_range>*.



The following results are possible:

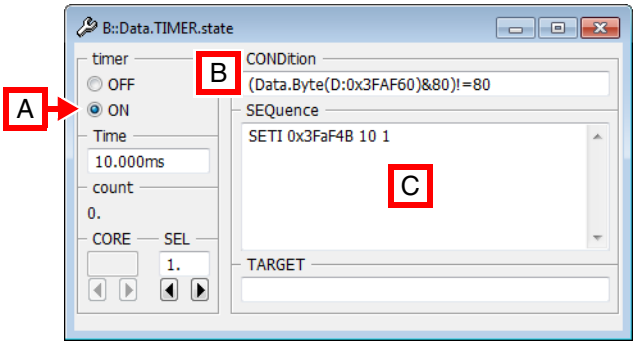
ok	RAM
read only	ROM/FLASH
read fail **write fail**	no memory

The command **Data.TestList** may cause a “debug port fail” error if the peripherals are accessed.

The **Data.TIMER** command group allows to define a sequence of **Data.Set** commands that are executed periodically. This command group can be used e.g. to trigger a watchdog while the program execution is stopped.

The command is only active when the core is halted in debug mode.

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **Data.TIMER.state** window:



A For descriptions of the commands in the **Data.TIMER.state** window, please refer to the **Data.TIMER.*** commands in this chapter. Example: For information about **ON**, see **Data.TIMER.ON**.

B Conditions can be set up in the **CONDition** field using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** field using the *<data_set_commands>* **SET**, **SETI**, **GETS**, and **SETS**.

See also

- Data.TIMER.CONDition
- Data.TIMER.CORE
- Data.TIMER.ERRORSTOP
- Data.TIMER.OFF
- Data.TIMER.ON
- Data.TIMER.RESet
- Data.TIMER.SELect
- Data.TIMER.SEQuence
- Data.TIMER.state
- Data.TIMER.TARGET
- Data.TIMER.Time
- Data.ATTACH
- Data.EPILOG
- Data.PROLOG

Format:	**Data.TIMER.CONDition** *<condition>*
<condition>:	*<memory_access>* **&** *<mask>* **==** *<value>* *<memory_access>* **&** *<mask>* **!=** *<value>*
<memory_access>:	**Data.Byte(***<address>***)** | **Data.Word(***<address>***)** | **Data.Long(***<address>***)**

Defines a condition on which the command sequence defined with **Data.TIMER.SEQuence** will be executed periodically.

<memory_access>	Supported **Data.*()** functions are: • **Data.Byte()** and its short form **D.B()** • **Data.Long()** and its short form **D.L()** • **Data.Word()** and its short form **D.W()**

Examples:

```
; Data.TIMER is only active if most significant bit of
; 32-bit word at address 0x3fa30 is set.
Data.TIMER.CONDition (Data.Long(D:0x3fa30)&0x80000000)!=0
```

```
; Data.TIMER is only active if most significant bit of
; 16-bit word at address 0x3fa30 is set to value 0x3344.
Data.TIMER.CONDition Data.Word(D:0x3fa30)==0x3344
```

```
; Data.TIMER is only active if most significant bit of
; byte at address 0x3fa30 has most significant bits set to b'10.
Data.TIMER.CONDition Data.Byte(D:0x3fa30)==0y10xxXXXX
```

See also

- Data.TIMER
- Data.TIMER.state

Format: **Data.TIMER.CORE** *<core_number>*

Selects the core for which you want to define one or more data timer sequences.

Prerequisite: You have successfully configured an SMP system with the **CORE.ASSIGN** command.

Example: This script shows how to define a data timer sequence that is executed on core 3 of a multicore chip.

```
;Select the core for which you want to define a sequence
Data.TIMER.CORE 3.

;Define the sequence for core 3
Data.TIMER.CONDition <your_code>
Data.TIMER.SEQuence  <your_code>
```

For information on how to configure two different sequences, see **Data.TIMER.SELect**.

See also

- Data.TIMER
- Data.TIMER.state

Format: **Data.TIMER.ERRORSTOP [ON | OFF]**

Default: ON.

If this command is set to OFF, the **Data.TIMER** sequence is not stopped on memory access errors.

See also

- Data.TIMER
- Data.TIMER.state

Data.TIMER.OFF Switch timer off

Format: **Data.TIMER.OFF**

Switches the **Data.TIMER** feature off.

See also

- Data.TIMER
- Data.TIMER.state

Data.TIMER.ON Switch timer on

Format: **Data.TIMER.ON**

Switches the **Data.TIMER** feature on.

See also

- Data.TIMER
- Data.TIMER.state

Format: **Data.TIMER.RESet**

Switches the **Data.TIMER** feature off and clears all settings.

See also

- Data.TIMER
- Data.TIMER.state

Data.TIMER.SELect Increment the index number to the next sequence

Format: **Data.TIMER.SELect** *<serial_number>*

Increments the index number for each new sequence. This is useful, for example, if you need two separate sequences with each sequence having its own **Data.TIMER.CONDition**.

TRACE32 automatically assigns the index number 1. to the first **Data.TIMER.SEQuence**. If you require a second, separate sequence, then increment the *<index_number>* to 2. Otherwise the second sequence will overwrite the first one. You can define a maximum of 10 sequences.

Example 1: Two sequences with the *same* **Data.TIMER.CONDition** may have the *same* index number. The backslash **** is used as a line continuation character. No white space permitted after the backslash.

```
;Set the index number to 1.
Data.TIMER.SELect 1.

;Sequences shall be executed only if this condition is true:
Data.TIMER.CONDition (Data.Word(D:0x4faf34)&0xff00)==0x4000

;Define the two sequences:
Data.TIMER.SEQuence SET 0x4faf54 %Word 0xC0C0 \
                    SET 0x4faf64 %Word 0xD0D0
```

Example 2: Two sequences with *different* **Data.TIMER.CONDition** settings require two *different* index numbers.

```
;1st sequence - TRACE32 automatically sets the index number to 1.
Data.TIMER.SELect 1.

;If this condition is true, ...
Data.TIMER.CONDition (Data.Word(D:0x4faf38)&0xff00)==0x2000

;... then the 1st sequence will be executed
Data.TIMER.SEQuence SET 0x4faf58 %Word 0xE0E0

;Increment the index number to define the 2nd sequence
Data.TIMER.SELect 2.

;If this condition is true, ...
Data.TIMER.CONDition (Data.Word(D:0x4faf38)&0xff00)==0x3000

;... then the 2nd sequence will be executed
Data.TIMER.SEQuence SET 0x4faf58 %Word 0xF0F0
```

See also

- Data.TIMER
- Data.TIMER.state

Data.TIMER.SEQuence — Define timer sequence

Format:	**Data.TIMER.SEQuence** <*command*> …
<*command*>:	**SET** <*address*> **%**<*format*> <*data*> **SETI** <*address*> **%**<*format*> <*data*> <*increment*> **SETS** <*address*> **GETS** <*address*>

Defines a sequence of **Data.Set** commands that are periodically executed by the TRACE32 software when the program execution is stopped. The period is defined by **Data.TIMER.Time**.

SET — Write <*data*> periodically to <*address*> while the program execution is stopped.

SETI — Write <*data*> to <*address*> the first time after the program execution is stopped after **Data.TIMER.ON**.

Then <*data*> is incremented by <*increment*> periodically while the program execution is stopped.

SETS Write the data that was saved with a previous **GETS** back to *<address>*.

GETS Save the data at *<address>* periodically while the program execution is stopped.

Examples:

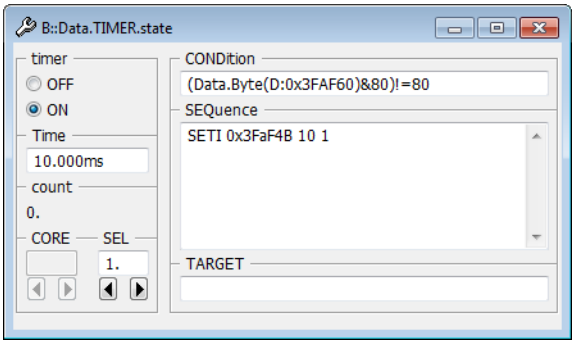
```
Data.TIMER.SEQuence SET  0x3fa50 %Long 0x11223344

Data.TIMER.SEQuence SETI 0x3fa50 %Word 0xa0a0 2

Data.TIMER.SEQuence SETS 0x3fa60

Data.TIMER.SEQuence GETS 0x3fa60
```

See also

- Data.TIMER
- Data.TIMER.state

Data.TIMER.state — Timer state display

Format: **Data.TIMER.state**



See also

- Data.TIMER
- Data.TIMER.CONDition
- Data.TIMER.CORE
- Data.TIMER.ERRORSTOP
- Data.TIMER.OFF
- Data.TIMER.ON
- Data.TIMER.RESet
- Data.TIMER.SELect
- Data.TIMER.SEQuence
- Data.TIMER.TARGET
- Data.TIMER.Time

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.TIMER.TARGET** *<code_range> <data_range>*

Defines a target program that is periodically executed while the program execution is stopped.

<code_range> Defines the address range for the target program.

<data_range> Defines the address range used for the data of the target program.

Example:

```
Data.TIMER.TARGET 0x3fa948--0x3faa07 0x1000--0x11ff
```

See also

- Data.TIMER
- Data.TIMER.state

Data.TIMER.Time Define period for timer

Format: **Data.TIMER.Time** *<time>*

Defines the period for the **Data.TIMER** feature.

Example:

```
Data.TIMER.Time 10.ms
```

See also

- Data.TIMER
- Data.TIMER.state

Format: **Data.UNTAGFunc**

Removes the tags generated by the **Data.TAGFunc** command.

See also

- Data.TAGFunc

Data.UPDATE Target memory cache update

Format: **Data.UPDATE**

Triggers and update of memory buffered by the debugger. Memory is only buffered when the address range is declared by **MAP.UpdateOnce** command.

Format 1:	**Data.USRACCESS** *<code_range> <data_range>* [*<bin_file>* [**/***<option>*]]
Format 2:	**Data.USRACCESS** *<code_address> <data_address> <buffer_size>* [*<bin_file>* [**/***<option>*]]
<option>:	**STACKSIZE** *<size>* | **KEEP**

Targets may include memory that is not in the address space accessible by the debugger. An external access algorithm can be linked to TRACE32 to realize an access to this memory.

After the external access algorithm is linked to TRACE32 by the command **Data.USRACCESS** this memory can be displayed and modified like any other memory by using the access class **USR** and a command from the **Data** command group.

The external access algorithm is unlinked on every execution of **SYStem.Mode** (e.g. SYStem.Mode.Up) and when a error occurs. If no external access algorithm is linked, the access class **USR** is inaccessible.

STACKSIZE	*<data_range>* includes 256 bytes for the stack. If your access algorithm requires a smaller/larger stack the default stack size can be changed by the option **STACKSIZE** *<size>*.
KEEP	TRACE32 loads *<bin_file>* to the target RAM before the memory is accessed and restores the saved *<code_range>* and *<data_range>* when the memory access is done. The option **KEEP** advises TRACE32 not to restore the saved *<code_range>* and *<data_range>*. This is useful for tests and for performance improvements.

Example:

```
;               <code_range>       <data_range>       <bin_file>
Data.USRACCESS 0x10000000++0x3ff 0x10000400++0xbff usraccess.bin

Data.dump USR:0x9000

Data.Set USR:0x9005 %Long 0xaa74
```

Further examples: Scripts that demonstrate the usage of the command **Data.USRACCESS** can be found in ~~/demo/*<architecture>*/etc/usraccess, e.g. ~~/demo/arm/etc/usraccess

See also

- Data.dump

▲ 'Release Information' in 'Legacy Release History'

Format: **Data.VECTOR** [**%***<format>*] [*<address>* | *<range>*] [***/***<option> …]

<format>:
Decimal.[*<width>*[**.***<endianness>*]]
DecimalU.[*<width>*[**.***<endianness>*]]
Hex.[*<width>*[**.***<endianness>*]]
HexS.[*<width>*[**.***<endianness>*]]
OCTal.[*<width>*[**.***<endianness>*]]
Float[**.***<float_rep>*[**.***<endianness>*]]
Byte [**.***<endianness>*]
Word [**.***<endianness>*]
Long [**.***<endianness>*]
Quad [**.***<endianness>*]
TByte [**.***<endianness>*]
PByte [**.***<endianness>*]
HByte [**.***<endianness>*]
SByte [**.***<endianness>*]

<width>: **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

<endianness>: **DEFault** | **LE** | **BE**

<bitorder>: **DEFault** | **BitSwap**

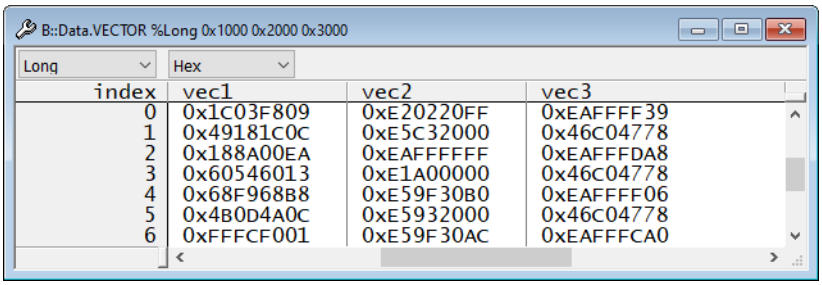
<option>: **CORE** *<core_number>*

Displays memory contents from up to 10 addresses/address ranges as vectors side by side. If a single address is selected then this address defines the windows' initial position. Scrolling makes all memory contents visible. When selecting an address range only the defined data range is shown.

Decimal, DecimalU,...	Refer to **"Keywords for <format>"**, page 10
Byte, **Word**, …	See **"Keywords for <width>"**, page 11.
BE, **LE**	Define byte-order display direction: Big Endian or Little Endian.
BitSwap	Display data with reverse bit-order in each byte. If **BitSwap** is used with **BE** or **LE**, the byte order will not changed, otherwise **BitSwap** will also reverse the byte-order.
CORE *<number>*	Displays memory from the perspective of the specified core (SMP debugging only).

Example:

```
Data.VECTOR %Long 0x1000 0x2000 0x3000
```



index	vec1	vec2	vec3
0	0x1C03F809	0xE20220FF	0xEAFFFF39
1	0x49181C0C	0xE5C32000	0x46C04778
2	0x188A00EA	0xEAFFFFFF	0xEAFFFDA8
3	0x60546013	0xE1A00000	0x46C04778
4	0x68F968B8	0xE59F30B0	0xEAFFFF06
5	0x4B0D4A0C	0xE5932000	0x46C04778
6	0xFFFCF001	0xE59F30AC	0xEAFFFCA0

Format: **Data.View** [**%***<format>*] [*<address>* | *<range>*] [**/***<option>* …]

<format>: **Decimal** [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
DecimalU [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Hex [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
HexS [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
OCTal [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Ascii [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Binary [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Float[**.***<float_rep>*[**.***<endianness>*]]
sYmbol [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Var
DUMP [**.***<width>* [**.***<endianness>* [**.***<bitorder>*]]]
Byte [**.***<endianness>* [**.***<bitorder>*]]
Word [**.***<endianness>* [**.***<bitorder>*]]
Long [**.***<endianness>* [**.***<bitorder>*]]
Quad [**.***<endianness>* [**.***<bitorder>*]]
TByte [**.***<endianness>* [**.***<bitorder>*]]
PByte [**.***<endianness>* [**.***<bitorder>*]]
HByte [**.***<endianness>* [**.***<bitorder>*]]
SByte [**.***<endianness>* [**.***<bitorder>*]]

<width>: **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**

<endianness>: **DEFault** | **LE** | **BE**

<bitorder>: **DEFault** | **BitSwap**

<option>: **CORE** *<core_number>*
COVerage
CTS
Track
FLAG *<flag>*
CFlag *<cflag>*
Mark *<break>*

<flag>: **Read** | **Write** | **NoRead** | **NoWrite**

<cflag>: **OK** | **NoOK** | **NOTEXEC** | **EXEC**

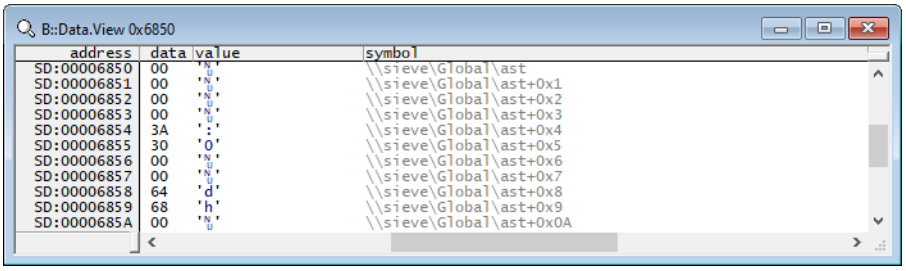
<break>: **Program** | **HII** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

Displays bare memory content as a list.

If a single address is selected then this address defines the windows' initial position. Scrolling makes all memory contents visible.

When selecting an address range only the defined data range is shown. A range definition is useful whenever addresses following the address range are read protected (e.g., in the case of I/O).

Decimal, DecimalU,...	Refer to **"Keywords for <format>"**, page 10
Byte, **Word**, …	See **"Keywords for <width>"**, page 11.
BE, **LE**	Define byte-order display direction: Big Endian or Little Endian.
BitSwap	Display data with reverse bit-order in each byte. If **BitSwap** is used with **BE** or **LE**, the byte order will not changed, otherwise **BitSwap** will also reverse the byte-order.
CORE *<core>*	Display memory from the perspective of the specified core /SMP debugging only).
COVerage	Highlight data memory locations that have never been read/written.
Track	Track the window to the reference address of other windows.
Mark *<break>*	Highlight memory locations for which the specified breakpoint is set.
CTS	Display **CTS** access information when CTS mode is active.



The scale area contains addresses and memory classes. The state line displays all current addresses, both in hexadecimal and symbolic format. By clicking on a data word, or by means of "Set", a **Data.Set** command can be executed on the current address. By holding down the left mouse button the most important memory functions can be executed via softkeys. If the **Mark** option is on, the relevant bytes will be highlighted. For more information see **Data.dump** window.

See also

- Data.CHAIN
- Data.dump
- Data.In
- Data.Print
- Data.TABle
- Data.Byte()
- Data.Float()
- Data.Long()
- Data.Quad()
- Data.STRing()
- Data.STRingN()
- Data.SUM()

Data.WRITESTRING Write string to PRACTICE file

Format: **Data.WRITESTRING** #*<file_number>*

Writes a string from the target memory to a PRACTICE script file (*.cmm).

Example:

```
OPEN #1 testfile /Create
Data.WRITESTRING #1 100--1ff
CLOSE #1
```

See also

■ Data.STRING
■ CLOSE
■ OPEN
❏ ADDRESS.OFFSET()
❏ ADDRESS.SEGMENT()
❏ ADDRESS.STRACCESS()
❏ ADDRESS.WIDTH()
❏ Data.Byte()
❏ Data.Float()
❏ Data.Long()
❏ Data.Quad()
❏ Data.STRing()
❏ Data.STRingN()
❏ Data.Word()
❏ Data.WSTRING()

▲ 'Release Information' in 'Legacy Release History'

DCI Direct Connect Interface (DCI)

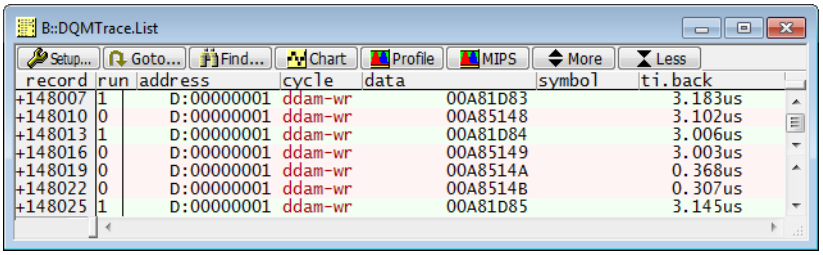
Intel® x86

The Intel® Direct Connect Interface (DCI) allows debugging and tracing of Intel® targets using the USB3 port of the target system. The Intel® DCI trace handler is a hardware module of the implementation on the target system. This module is responsible for forwarding trace data coming from the Intel® Trace Hub to DCI.

The **DCI** command group allows expert control of this hardware module. If the Intel® Trace Hub commands are used, then this configuration is done automatically (see **ITH** commands in .

For more information about the direct connect interface (DCI), see **"Debugging via Intel® DCI User´s Guide"** (dci_intel_user.pdf).

The **DQMTrace** command group allows to display and analyze trace information exported by Data Acquisition Messaging of Nexus PowerArchitecture.




```
DQMTrace.List
```

column layout	
address	Identification tag taken from $DEVENT_{DQTAG}$ register field
cycle	Write access to DDAM (Debug Data Acquisition Message Register)
data	Exported data taken from DDAM register

DTM — DTM trace sources (Data Trace Module)

DTM trace sources can show the contents of simple CoreSight trace sources in different formats. Trace sources are typically either internal signals, busses or instrumentation traces.

DTM.CLOCK — Set core clock frequency for timing measurements

Format:	**DTM.CLOCK** *<frequency>*

Tells the debugger the core clock frequency of the traced Arm core.

DTM.CycleAccurate — Cycle accurate tracing

Format:	**DTM.CycleAccurate** [**ON** | **OFF**]

Enables cycle accurate tracing if ON. Default is OFF. Refer for more information about cycle accurate tracing to **ETM.CycleAccurate**.

DTM.Mode — Define DTM mode

Format:	**DTM.Mode** [*<mode>*]
<mode>]	**Byte** | **NibbleLE** | **WordLE** | **LongLE** | **QuadLE**

Defines DTM mode.

DTM.OFF — Disable DTM

Format:	**DTM.OFF**

Disables DTM functionality.

DTM.ON — Enable DTM

Format:	**DTM.ON**

Enables DTM functionality.

DTM.Register — Display DTM registers

Format:	**DTM.Register** [*<file>* **/***<option>*]
<option>:	**SpotLight** | **DualPort** | **Track** | **AlternatingBackGround** **CORE** *<core_number>*

Display the DTM registers.

<option>	For a description of the options, see **PER.view**.

DTM.RESet — Reset DTM settings

Format:	**DTM.RESet**

Resets the DTM settings to default.

Format: **DTM.TraceID** *<id>*

By default TRACE32 automatically assigns a trace source ID to all cores with a CoreSight ETM, the first ITM, and the first HTM. The command **DTM.TraceID** allows to assign an ID to a trace source overriding the defaults.

DTM.TracePriority Define priority of DTM

Format: **DTM.TracePriority** *<priority>*

The CoreSight Trace Funnel combines 2 to 8 ATB input ports to a single ATB output. An arbiter determines the priority of the ATB input port. Port 0 has the highest priority (0) and port 7 the lowest priority (7) by default.

The command **DTM.TracePriority** allows to change the default priority of an ATB input port.

DTM<trace> - Trace Data Analysis

DTM<trace> Command groups for DTM<trace>

See also

- DTMAnalyzer
- DTMCAnalyzer
- DTMHAnalyzer
- DTMLA
- DTMOnchip
- DTMTrace

Overview DTM<trace>

Using the **DTM<trace>** command groups, you can configure the trace recording as well as analyze and display the recorded DTM trace data. The command groups consist of the name of the trace source, here **DTM**, plus the TRACE32 trace method you have chosen for recording the DTM trace data.

For more information about the TRACE32 convention of combining *<trace_source>* and *<trace_method>* to a *<trace>* command group that is aimed at a specific trace source, see **"Replacing <trace> with Trace Source and Trace Method - Examples"** (general_ref_t.pdf).

Not any arbitrary combination of *<trace_source>* and *<trace_method>* is possible. For an overview of the available command groups **"Related Trace Command Groups"** (general_ref_t.pdf).

Example:

```
DTMTrace.state            ;optional step: open the window in which the
                          ;trace recording is configured.
DTMTrace.METHOD Analyzer  ;select the trace method Analyzer for
;<configuration>          ;recording trace data.

DTM.state                 ;optional step: open the window in which
                          ;the trace source DTM is configured.
DTM.ON                    ;switch the trace source DTM on.
;<configuration>

;trace data is recorded using the commands Go, WAIT, Break

DTMAnalyzer.List          ;display the DTM trace data recorded with the
                          ;trace method Analyzer as a trace listing.

DTMTrace.List             ;this is the generic replacement for the above
                          ;DTMAnalyzer.List command.
```

Format: **DTMAnalyzer.**<*sub_cmd*>

The **DTMAnalyzer** command group allows to display and analyze the information emitted by the **DTM**.

The DTM information emitted off-chip via the Trace Port Interface Unit (**TPIU**) is recorded by the TRACE32 PowerTrace.

<*sub_cmd*>	For descriptions of the subcommands, please refer to the general <*trace*> command descriptions in **"General Commands Reference Guide T"** (general_ref_t.pdf). **Example**: For a description of **DTMAnalyzer.List** refer to **<trace>.List**

See also

- DTM<trace>

▲ 'Release Information' in 'Legacy Release History'

DTMCAnalyzer Analyze DTM information recorded by CombiProbe

Format: **DTMCAnalyzer.**<*sub_cmd*>

The **DTMCAnalyzer** command group allows to display and analyze the information emitted by the **DTM**.

The DTM information emitted off-chip via the Trace Port Interface Unit (**TPIU**) is recorded by the TRACE32 CombiProbe.

<*sub_cmd*>	For descriptions of the subcommands, please refer to the general <*trace*> command descriptions in **"General Commands Reference Guide T"** (general_ref_t.pdf). **Example**: For a description of **DTMCAnalyzer.List** refer to **<trace>.List**

See also

- DTM<trace>

Format:	**DTMHAnalyzer.**<*sub_cmd*>

The **DTMHAnalyzer** command group allows to display and analyze the information emitted by the **DTM**. Trace data is transferred off-chip using fast protocols as USB or Ethernet and is recorded in the trace memory of the TRACE32 host analyzer. Please refer to the description of the **HAnalyzer** command group for more information.

<*sub_cmd*>	For descriptions of the subcommands, please refer to the general <*trace*> command descriptions in **"General Commands Reference Guide T"** (general_ref_t.pdf). **Example**: For a description of **DTMHAnalyzer.List** refer to **<trace>.List**

See also

- DTM<trace>

DTMLA Analyze DTM information from binary source

[build 135684 - DVD 09/2021]

Format:	**DTMLA.**<*sub_cmd*>

The **DTMLAnalyzer** command group allows to display and analyze the information emitted by the **DTM**. Trace data is collected form Lauterbach's Logic Analyzer or from a binary file.

<*sub_cmd*>	For descriptions of the subcommands, please refer to the general <*trace*> command descriptions in **"General Commands Reference Guide T"** (general_ref_t.pdf). **Example**: For a description of **DTMLAnalyzer.List** refer to **<trace>.List**

See also

- DTM<trace>

Format:	**DTMOnchip.**<*sub_cmd*>

The **DTMOnchip** command group allows to display and analyze the information emitted by the **DTM**.

<*sub_cmd*>	For descriptions of the subcommands, please refer to the general <*trace*> command descriptions in **"General Commands Reference Guide T"** (general_ref_t.pdf). **Example**: For a description of **DTMOnchip.List** refer to **<trace>.List**

See also

- DTM<trace>

DTMTrace Method-independent analysis of DTM trace data

[build 135684 - DVD 09/2021]

Format:	**DTMTrace.**<*sub_cmd*>

The **DTMTrace** command group can be used as a generic replacement for the above **DTM<trace>** command groups.

<*sub_cmd*>	For descriptions of the subcommands, please refer to the general <*trace*> command descriptions in **"General Commands Reference Guide T"** (general_ref_t.pdf). **Example**: For a description of **DTMTrace.List** refer to **<trace>.List**

See also

- DTM<trace>