General Commands Reference Guide D

Usage:

(B) command only available for ICD
(E) command only available for ICE
(F) command only available for FIRE

Version 16-Apr-2019

History

21-Feb-19	Description of the Data.STANDBY command group (only for PowerPC MPC5xxx and TriCore).
18-Sep-18	Updated Data.SUM. Added description of ByteSWAP option.
10-Jan-18	Updated Data.LOAD.CrashDump.
12-Jun-17	Updated Data.CLEARVM.

Data

Data — Memory access

See also

■ Data.AllocList	■ Data.Assemble	■ Data.BDTAB	■ Data.BENCHMARK
■ Data.CHAIN	■ Data.CHAINFind	■ Data.CLEARVM	■ Data.ComPare
■ Data.COPY	■ Data.CSA	■ Data.DRAW	■ Data.DRAWFFT
■ Data.DRAWXY	■ Data.dump	■ Data.EPILOG	■ Data.Find
■ Data.FindCODE	■ Data.GOTO	■ Data.GREP	■ Data.IMAGE
■ Data.In	■ Data.LOAD	■ Data.MSYS	■ Data.Out
■ Data.PATTERN	■ Data.Print	■ Data.PROfile	■ Data.PROGRAM
■ Data.PROLOG	■ Data.REF	■ Data.ReProgram	■ Data.ReRoute
■ Data.SAVE.<format>	■ Data.Set	■ Data.SOFTEPILOG	■ Data.SOFTPROLOG
■ Data.STANDBY	■ Data.STARTUP	■ Data.STRING	■ Data.SUM
■ Data.TABle	■ Data.TAG	■ Data.TAGFunc	■ Data.Test
■ Data.TestList	■ Data.TIMER	■ Data.UNTAGFunc	■ Data.UPDATE
■ Data.USRACCESS	■ Data.View	■ Data.WRITESTRING	■ SYStem.LOG.List
❏ ADDRESS.OFFSET()	❏ ADDRESS.SEGMENT()	❏ ADDRESS.STRACCESS()	❏ ADDRESS.WIDTH()
❏ Data.Byte()	❏ Data.Float()	❏ Data.Long()	❏ Data.Quad()
❏ Data.STRing()	❏ Data.STRingN()	❏ Data.Word()	

▲ 'ADDRESS Functions' in 'General Function Reference'
▲ 'Data Functions' in 'General Function Reference'

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
```

Memory Access by TRACE32-ICE, TRACE32-FIRE

The TRACE32 In-Circuit Emulators feature dual-port emulation memory which can be accessed by both the emulation CPU, as well as the system control CPU. The emulation CPU always reads data from memory which is assigned to it via the **MAP.Intern** or **MAP.Extern** command (external or internal memory). However, the dual-port function allows access to emulation memory only. Memory access to external memory or to I/O areas must always be handled via the emulation CPU.

Internal emulation memory works in parallel to external emulation memory. After write-to execution both memory areas are updated. Wait states results on the signals generated by the target system.

Access Procedures

Access procedures are selected by entering a memory class in the address field. For example, if the command

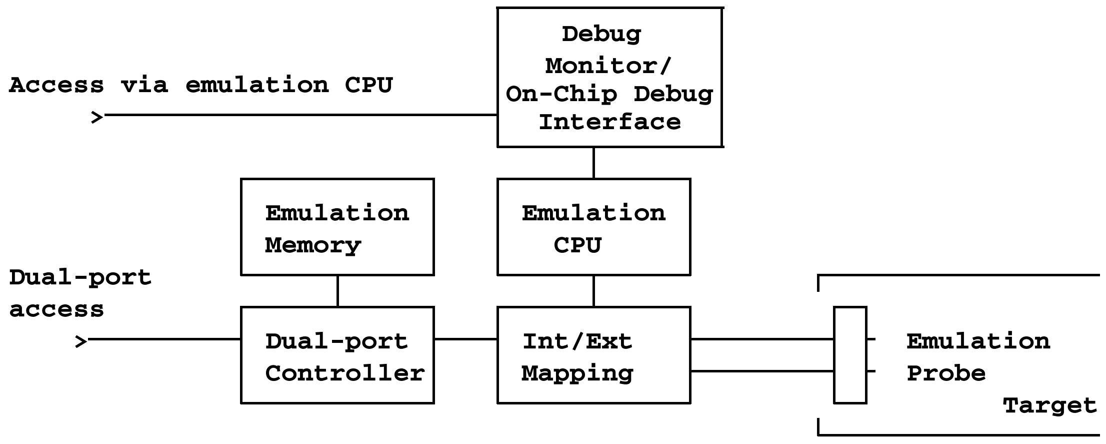
```
Data.dump UD:0x1000
```

is used to represent memory in the USERDATA address space, as of address 0x1000.

The command

```
Data.dump EUD:0x1000
```

will be used to represent the same address space, except that it can be accessed only directly via emulation memory, thus ensuring that memory contents are also visible during real-time emulation.



Memory Classes

The available memory access classes depend on the target processor. Some classes are available at all probes:

FCode	Description
P	Program Space
D	Data Space
C	Access by the CPU
E	Direct access to emulation memory
EP	Direct access to program memory
ED	Direct access to data memory
A	Absolute (Physical addressing)

Other classes are described within the **Processor Architecture Manuals**, **ICE Target Guides**, and **FIRE Target Guides**.

Keywords for <width>

TRACE32 provides 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)

Functions

The following table lists frequently-used **Data.*()** functions. For a complete list, see **"Data Functions"** in General Function Reference, page 126 (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:

T32_allocHeader
guard area head
allocated memory block
guard area tail

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
```

What does T32_malloc(size) do?

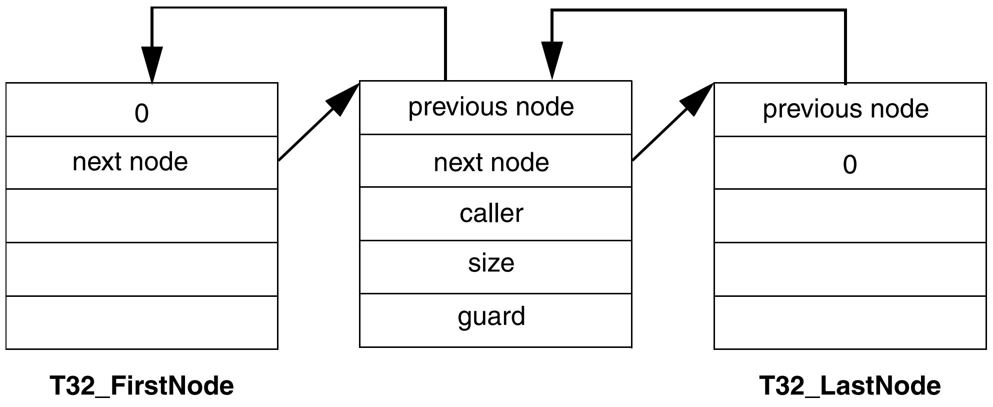
1. A memory block is allocated. This memory block has the following size:
   size of the requested memory block + sizeof(T32_allocHeader) + T32_GUARD_SIZE

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.

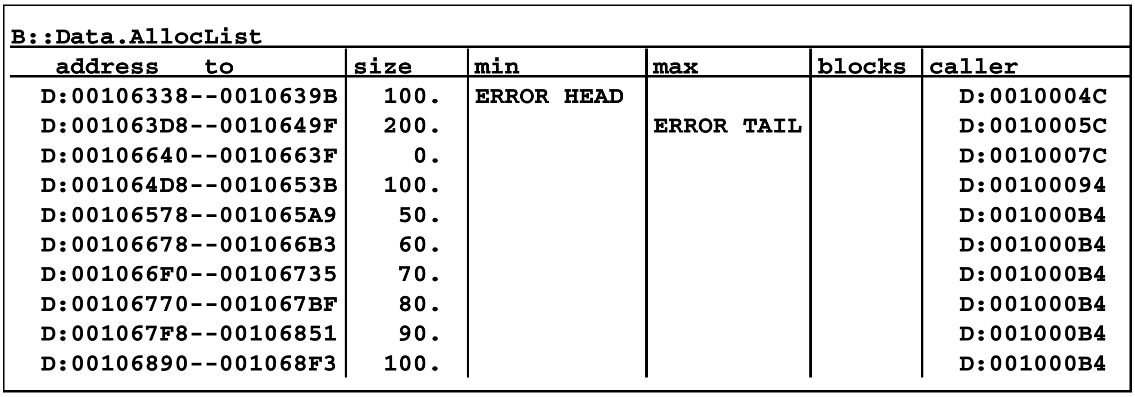
```
Data.AllocList List_M2          ; List_M2 is the start of the linked
                                ; list for the command Data.AllocList
```

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. This simulator can be installed from the TRACE32 software DVD.



```
B::Data.AllocList
   address    to       |size    |min          |max        |blocks |caller
  D:00106338--0010639B |  100.  |ERROR HEAD   |           |       |    D:0010004C
  D:001063D8--0010649F |  200.  |             |ERROR TAIL |       |    D:0010005C
  D:00106640--0010663F |    0.  |             |           |       |    D:0010007C
  D:001064D8--0010653B |  100.  |             |           |       |    D:00100094
  D:00106578--001065A9 |   50.  |             |           |       |    D:001000B4
  D:00106678--001066B3 |   60.  |             |           |       |    D:001000B4
  D:001066F0--00106735 |   70.  |             |           |       |    D:001000B4
  D:00106770--001067BF |   80.  |             |           |       |    D:001000B4
  D:001067F8--00106851 |   90.  |             |           |       |    D:001000B4
  D:00106890--001068F3 |  100.  |             |           |       |    D:001000B4
```

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 ❏ Data.AL.ERRORS()

▲ 'Release Information' in '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 **Data.List** window. The commands **Data.PROGRAM** and **Data.ReProgram** can be used to enter multiple instructions or small programs.

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

Data.A 0 move.b d0,d1        ; insert and assemble am move- command at
Data.A , move.b d3,d4        ; address 0
                             ; next command to next address
```

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 ■ Data.dump ■ Data.PROGRAM

▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Data and Program Memory' in 'Training FIRE Basics'
▲ 'Data and Program Memory' in 'Training ICE Basics'

Data.BDTAB Display buffer descriptor table

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 ■ 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:

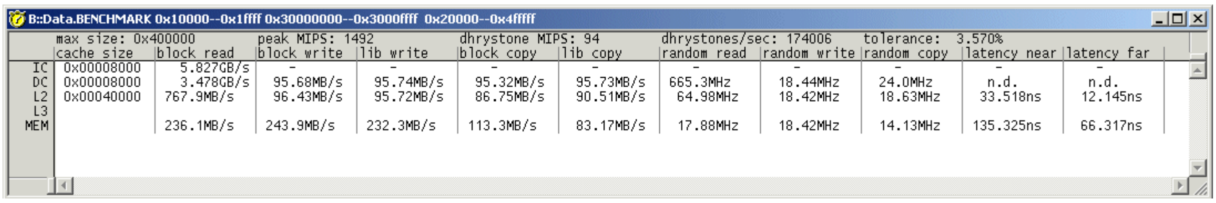
- Load a benchmark program that performs various memory read, memory write and memory copy operations to the target.
- Enable all caches.
- The command **Data.BENCHMARK** start the benchmark program and measures the bandwidth of all caches and memories with the help of the **RunTime** counters.

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.

Example for the PowerQuicc III:

```
; select the CPU
SYStem.CPU 85xx

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
```

See also

■ Data

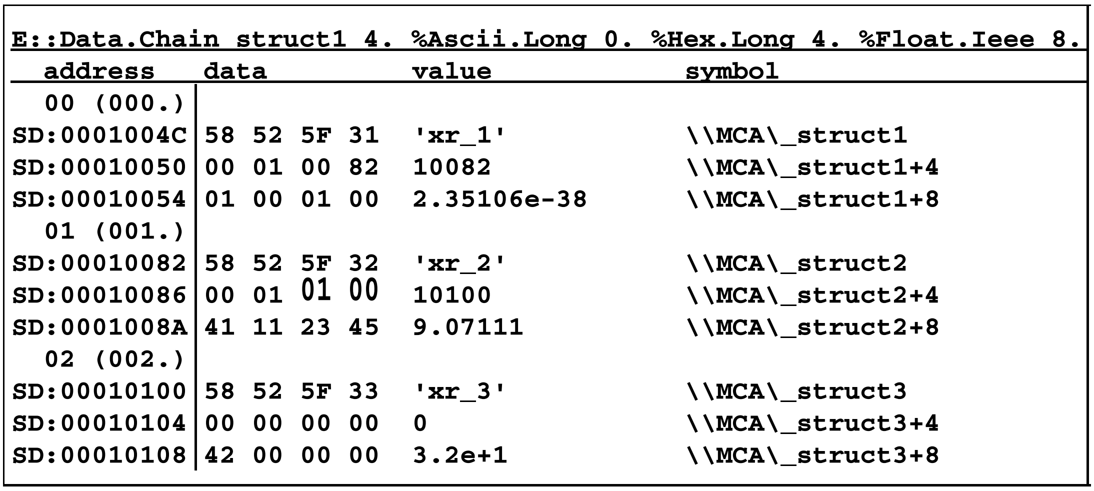
Format:	**Data.CHAIN** *<base> <link_offset> <elements>* [**/***<option>* …]
<elements>:	[[**%***<format>*] [*<address>* | *<range>*] …]
<format>:	**Decimal.** *<width>* **DecimalU.** *<width>* **Hex.** *<width>* **Ascii.** *<width>* **Binary.** *<width>* **Float. Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>* **Var** **DUMP** *<width>*
<width>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<option>:	**Mark** *<break>* **Flag** *<flag>* **Track**
<flag>:	**ReadFlag** | **WriteFlag** | **NoRead** | **NoWrite**
<break>:	**Program** | **Hll** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

This command 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.

DecimalU	Decimal unsigned.
Var	Display in HLL format.
Byte, **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	See **"Keywords for <width>"**, page 10.
Mark	By using the **Mark** option individual bytes can be marked, depending on their corresponding flags.

```
; A linked list is displayed 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. The element consists of a
; floating-point numbers and a pointers.

Data.CHAIN xlist 0x0c %Float.Ieee 0x0--0x7 %Hex.Long 0x8--0x0f
```

```
; Display a linked list, the first element is the symbol ast. The
; pointer to the next element is found at offset 6. from the base
; address. The element consists of a pointer, a counter, 2 pointers
; and a byte

Data.CHAIN ast 4. %Hex.Long 0. %Decimal.Word 4. %Hex.Long 6.
%Hex.Long 10. %Hex.Byte 15.
```

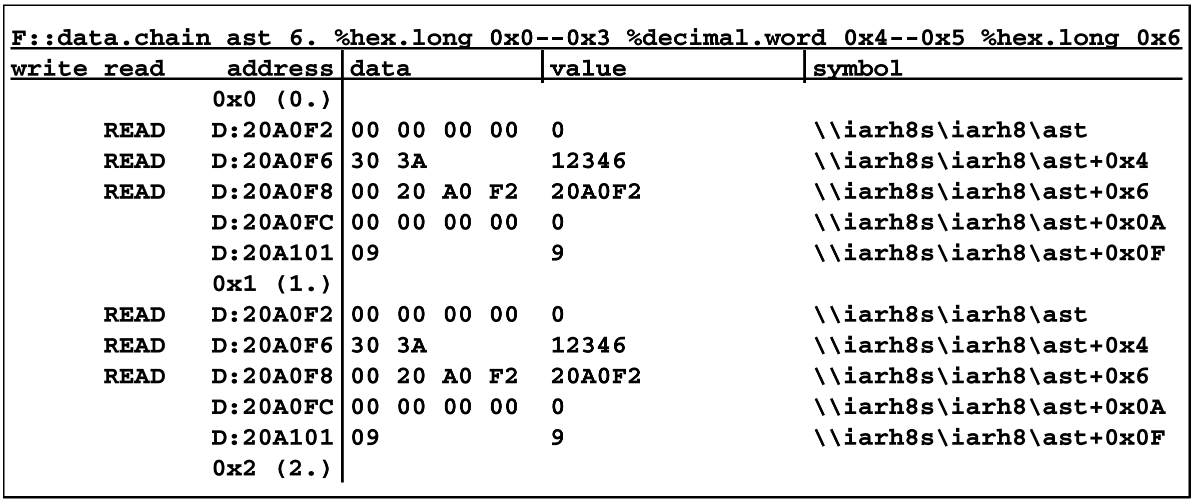


```
E::Data.Chain struct1 4. %Ascii.Long 0. %Hex.Long 4. %Float.Ieee 8.
  address    data          value            symbol
  00 (000.)
SD:0001004C 58 52 5F 31   'xr_1'           \\MCA\_struct1
SD:00010050 00 01 00 82   10082            \\MCA\_struct1+4
SD:00010054 01 00 01 00   2.35106e-38      \\MCA\_struct1+8
  01 (001.)
SD:00010082 58 52 5F 32   'xr_2'           \\MCA\_struct2
SD:00010086 00 01 01 00   10100            \\MCA\_struct2+4
SD:0001008A 41 11 23 45   9.07111          \\MCA\_struct2+8
  02 (002.)
SD:00010100 58 52 5F 33   'xr_3'           \\MCA\_struct3
SD:00010104 00 00 00 00   0                \\MCA\_struct3+4
SD:00010108 42 00 00 00   3.2e+1           \\MCA\_struct3+8
```

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.



```
F::data.chain ast 6. %hex.long 0x0--0x3 %decimal.word 0x4--0x5 %hex.long 0x6
write read      address|data         |value            |symbol
                0x0 (0.)|
       READ   D:20A0F2|00 00 00 00  0                 \\iarh8s\iarh8\ast
       READ   D:20A0F6|30 3A        12346             \\iarh8s\iarh8\ast+0x4
       READ   D:20A0F8|00 20 A0 F2  20A0F2            \\iarh8s\iarh8\ast+0x6
              D:20A0FC|00 00 00 00  0                 \\iarh8s\iarh8\ast+0x0A
              D:20A101|09           9                 \\iarh8s\iarh8\ast+0x0F
                0x1 (1.)|
       READ   D:20A0F2|00 00 00 00  0                 \\iarh8s\iarh8\ast
       READ   D:20A0F6|30 3A        12346             \\iarh8s\iarh8\ast+0x4
       READ   D:20A0F8|00 20 A0 F2  20A0F2            \\iarh8s\iarh8\ast+0x6
              D:20A0FC|00 00 00 00  0                 \\iarh8s\iarh8\ast+0x0A
              D:20A101|09           9                 \\iarh8s\iarh8\ast+0x0F
                0x2 (2.)|
```

The permanent update of long lists slows down the user interface. To balance this effect, the following radio button are provided to select the update mode:

Radio Buttons	
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	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 ■ Data.BDTAB
- ■ Data.dump ■ Data.TABle ■ Data.View

▲ 'Program and Data Memory' in 'ICE User's Guide'

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

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

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.

Data.CHAINFind xlist 0x0c /Address !0x10000--0x1ffff
```

See also

- Data.CHAIN
- Data

Data.CLEARVM

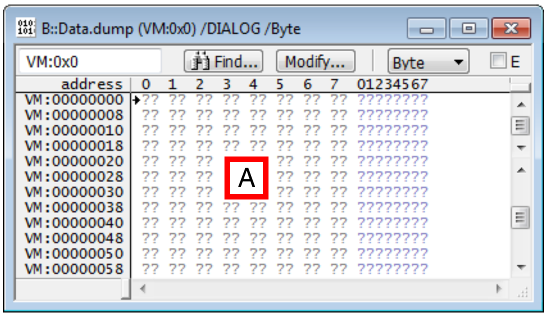
Format:	**Data.CLEARVM** [*<address>* | *<addressrange>*]

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

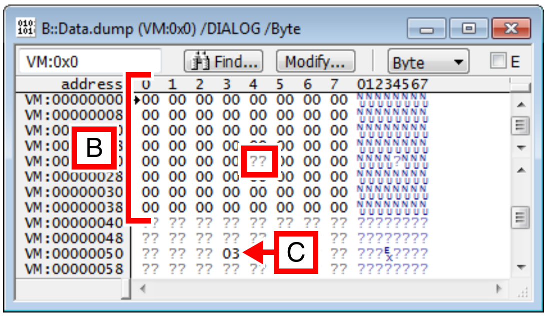
<address>	Clears one byte at the specified address.
<addressrange>	Clears the specified range.

Example: This script is for demo purposes only. To try this script, simply copy it to a `test.cmm` file, and then step through it in TRACE32 (See “**How to...**”).

```
;let's open the Data.dump window for demo purposes
Data.dump (VM:0x0) /DIALOG /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 1 byte at the specified address, see [B]
Data.CLEARVM VM:0x24

;in the uninitialized range, write 1 byte e.g. at VM:0x53, see [C]
Data.Set VM:0x53 %LE %Byte 0x3
```



```
     address   0  1  2  3  4  5  6  7  01234567
VM:00000000   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000008   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000010   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000018   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000020   ?? ?? ??       ?? ?? ??  ????????
VM:00000028   ?? ?? ??       ?? ?? ??  ????????
VM:00000030   ?? ?? ??       ?? ?? ??  ????????
VM:00000038   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000040   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000048   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000050   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000058   ?? ?? ?? ?? ?? ?? ?? ??  ????????
```



```
     address   0  1  2  3  4  5  6  7  01234567
VM:00000000   00 00 00 00 00 00 00 00
VM:00000008   00 00 00 00 00 00 00 00
VM:00          00 00 00 00 00 00 00 00
VM:00          00 00 00 00    00 00 00
VM:00          00 00 00 00 ?? 00 00 00
VM:00000028   00 00 00 00    00 00 00
VM:00000030   00 00 00 00 00 00 00 00
VM:00000038   00 00 00 00 00 00 00 00
VM:00000040   ?? ?? ?? ?? ?? ?? ?? ??  ????????
VM:00000048   ?? ?? ?? ?? ??       ??  ????????
VM:00000050   ?? ?? ?? 03          ??  ???
VM:00000058   ?? ?? ?? ?? ??       ??  ????????
```

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

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

C The *<address>* VM:0x53 was initialized with 0x3.

See also

■ Data

▲ ’Release Information’ in ’Release History’

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

…
```

Additional example for TRACE32-ICE/TRACE32-FIRE

```
MAP.Extern 0x0--0x3fff              ; map to target
Data.ComPare p:0x0--0x3fff ep:0     ; compare emulation memory with target
                                    ; memory
```

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.
ADDRESS.OFFSET()	Extracts the hex-address from the address object returned by TRACK.ADDRESS().

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

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

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

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

See also

■ Data.COPY ■ Data

▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in '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**

Data areas are copied. The address ranges may overlap. By copying to the same address programs from target may be copied to emulation memory.

Byte, **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	See **"Keywords for <width>"**, page 10.
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>*.

To illustrate **Data.COPY**, the following examples are provided:

- Example 1 shows various copy operations within the same memory and between target and emulation memory.
- Example 2 shows how to use **Data.COPY** together with **/DIFF.**
- Example 3 shows how to use **Data.COPY** together with **/LongSWAP.**

Example 1

```
;-------------------------------------------------------------------
; copy within memory

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

Additional examples for TRACE32-ICE and TRACE32-FIRE:

```
;-------------------------------------------------------------------
; copy from target to emulation memory
```

```
MAP.Extern 0x0--0x0ffff             ; map to target
Data.COPY 0x0--0x0ffff              ; copy to same address
MAP.Intern 0x0--0x0ffff             ; map to emulation memory

;------------------------------------------------------------------
; copy from target to emulation memory via dual port access

MAP.Extern 0x0--0x0ffff             ; map to target
Data.COPY 0x0--0x0ffff e:           ; copy to same address
                                    ; (dual-port access)
MAP.Intern 0x0--0x0ffff             ; map to emulation memory

;------------------------------------------------------------------
; copy from emulation memory to target

Data.COPY e:0x0--0x0ffff c:         ; read emulation memory and
                                    ; write with cpu
```

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.
ADDRESS.OFFSET()	Extracts the hex-address from the address object returned by **TRACK.ADDRESS()**.

```
;Your code

;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 "Error found at address " ADDRESS.OFFSET(TRACK.ADDRESS())

;Your code
```

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 ■ Data

▲ 'Data Access' in 'EPROM/FLASH Simulator'
▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'

Data.CSA

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

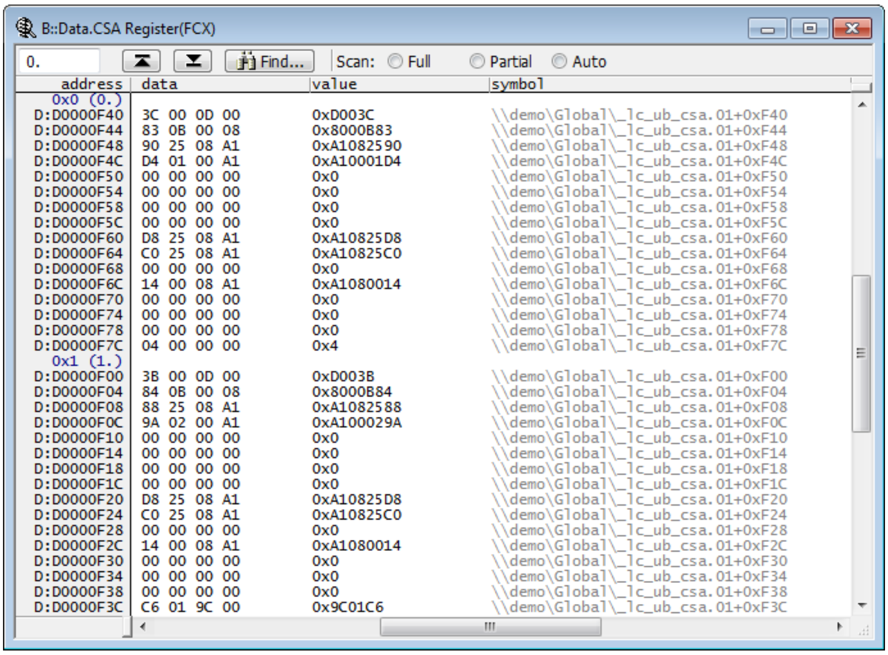
Displays a linked list of CSA entries. **Data.CSA** is a specialized variant of the **Data.CHAIN** command.

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)
```



```
    address  data          value       symbol
   0x0 (0.)
 D:D0000F40  3C 00 0D 00   0xD003C     \\demo\Global\_lc_ub_csa.01+0xF40
 D:D0000F44  83 0B 00 08   0x8000B83   \\demo\Global\_lc_ub_csa.01+0xF44
 D:D0000F48  90 25 08 A1   0xA1082590  \\demo\Global\_lc_ub_csa.01+0xF48
 D:D0000F4C  D4 01 00 A1   0xA10001D4  \\demo\Global\_lc_ub_csa.01+0xF4C
 D:D0000F50  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF50
 D:D0000F54  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF54
 D:D0000F58  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF58
 D:D0000F5C  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF5C
 D:D0000F60  D8 25 08 A1   0xA10825D8  \\demo\Global\_lc_ub_csa.01+0xF60
 D:D0000F64  C0 25 08 A1   0xA10825C0  \\demo\Global\_lc_ub_csa.01+0xF64
 D:D0000F68  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF68
 D:D0000F6C  14 00 08 A1   0xA1080014  \\demo\Global\_lc_ub_csa.01+0xF6C
 D:D0000F70  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF70
 D:D0000F74  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF74
 D:D0000F78  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF78
 D:D0000F7C  04 00 00 00   0x4         \\demo\Global\_lc_ub_csa.01+0xF7C
   0x1 (1.)
 D:D0000F00  3B 00 0D 00   0xD003B     \\demo\Global\_lc_ub_csa.01+0xF00
 D:D0000F04  84 0B 00 08   0x8000B84   \\demo\Global\_lc_ub_csa.01+0xF04
 D:D0000F08  88 25 08 A1   0xA1082588  \\demo\Global\_lc_ub_csa.01+0xF08
 D:D0000F0C  9A 02 00 A1   0xA100029A  \\demo\Global\_lc_ub_csa.01+0xF0C
 D:D0000F10  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF10
 D:D0000F14  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF14
 D:D0000F18  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF18
 D:D0000F1C  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF1C
 D:D0000F20  D8 25 08 A1   0xA10825D8  \\demo\Global\_lc_ub_csa.01+0xF20
 D:D0000F24  C0 25 08 A1   0xA10825C0  \\demo\Global\_lc_ub_csa.01+0xF24
 D:D0000F28  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF28
 D:D0000F2C  14 00 08 A1   0xA1080014  \\demo\Global\_lc_ub_csa.01+0xF2C
 D:D0000F30  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF30
 D:D0000F34  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF34
 D:D0000F38  00 00 00 00   0x0         \\demo\Global\_lc_ub_csa.01+0xF38
 D:D0000F3C  C6 01 9C 00   0x9C01C6    \\demo\Global\_lc_ub_csa.01+0xF3C
```

See also

■ Data.CHAIN ■ Data

Format:	**Data.DRAW** [**%**<*format*>] <*range1*> [<*range2*> …] [<*scale*> [<*offset*>]] [**/**<*display_option*>] [**/**<*array_options*>] [**/**<*options*>…]
<*format*>:	**Decimal.**[<*width*>] (signed integer, y-axis labeled with decimal numbers) **DecimalU.**[<*width*>] (unsigned integer, y-axis labeled with decimal numbers) **Hex.**[<*width*>] (unsigned integer, y-axis labeled with hex numbers) **Float.**<*float_rep*> (floating point)
<*width*>:	**DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte**
<*float_rep*>:	**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeQuad** | **IeeeXt10** | **IeeeRev** **IeeeS** | **IeeeDblS** | **IeeeDblT** **MFFP** | **Pdp11** | **Pdp11Dbl** | **RTOSUH** | **RTOSUHD** **Dsp16** | **Dsp16C** | **Dsp16Fix** | **Dsp32Fix** **M56** | **M560** | **M561** | **LACCUM** **Fract8** | **Fract16** | **Fract24** | **Fract32** | **Fract48** | **Fract64** **UFract8** | **UFract16** | **UFract24** | **UFract32** | **UFract48** | **UFract64** | **Fract40G** **MICRO** | **MICRO64** | **MILLI** | **MILLI64** | **NANO64** | **PICO64**
<*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**
<*display_ option*>:	**Vector** (default) | **Points** | **Steps** | **Impulses**
<*array_ options*>:	**Alternate** <*dimension_of_array*>: ***1…6*** **Element** <*elements_to_draw_in_window*>: **1…n**
<*options*>:	**LOG**: display y-axis in logarithmic scale **Track**: Highlight position selected in **Data.dump** window

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.

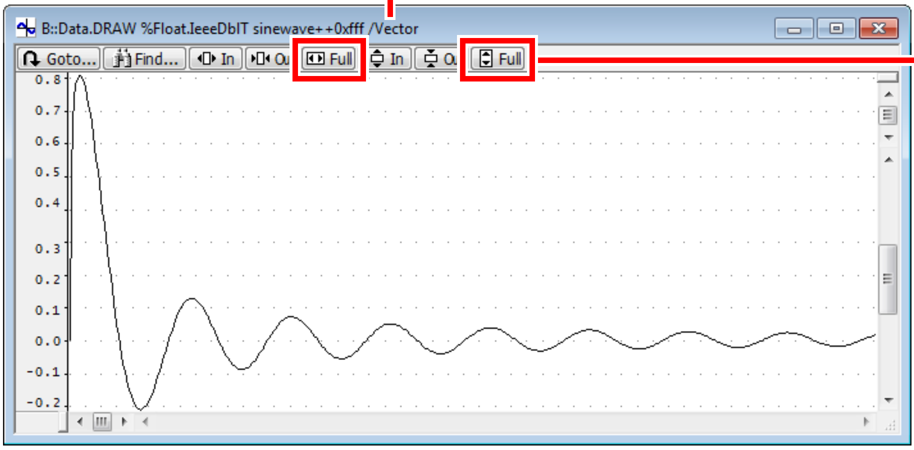
The command is useful, for example, for sampling signal quality in the mobile communications area or for sampling fuel injection in the automotive area. The **Var.DRAW** command has the same functionality, but takes the *<format>* information from the debug symbols.

Example for ARM Simulator

```
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>            <display_option>
Data.DRAW %Float.IeeeDblT Var.RANGE(sinewave) /Vector

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

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

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

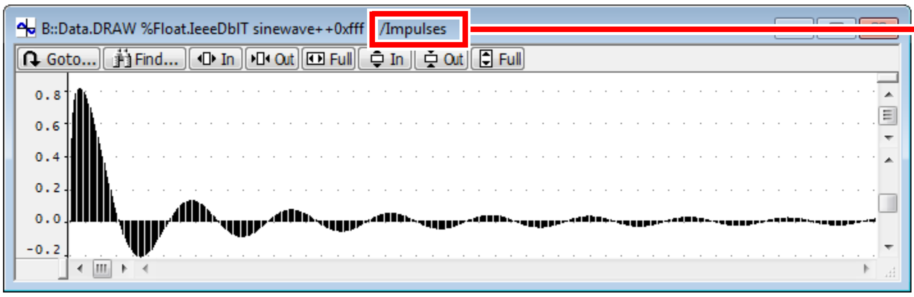
Result 1:

Here, the **/Vector** option is used.



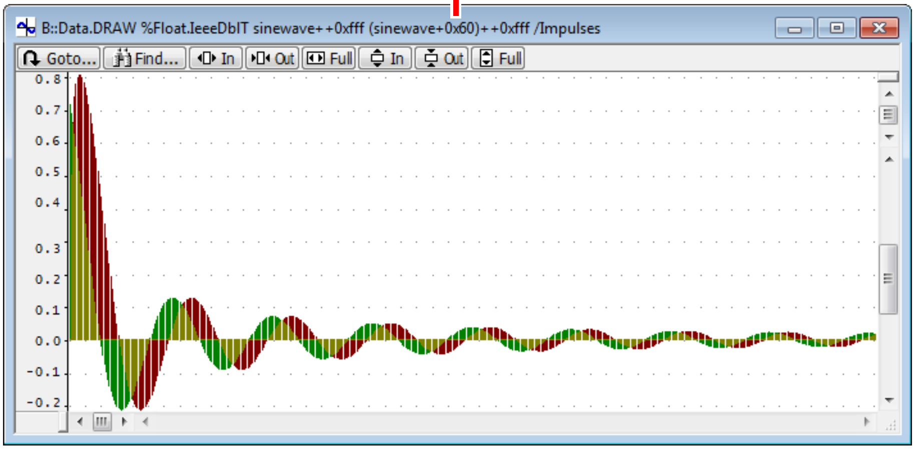
Click the two **Full** buttons.

Result 2:


B::Data.DRAW %Float.IeeeDblT sinewave++0xfff /Impulses

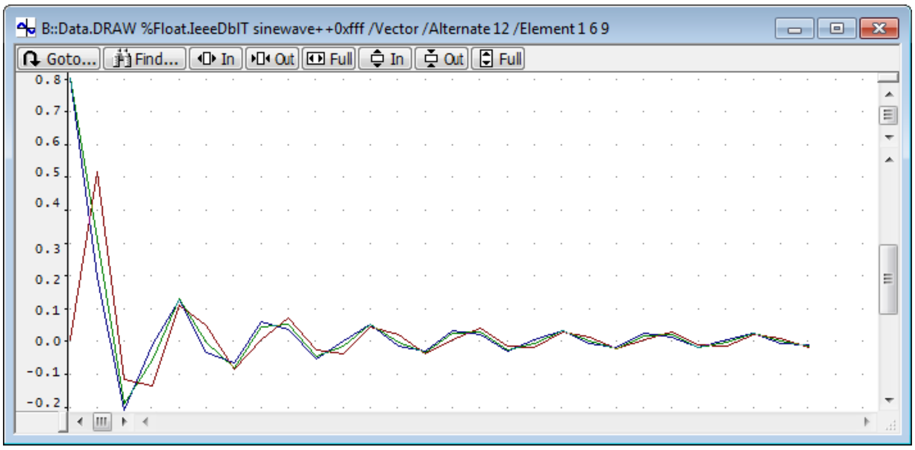
Here, the **/Impulses** option is used.

Result 3:


B::Data.DRAW %Float.IeeeDblT sinewave++0xfff (sinewave+0x60)++0xfff /Impulses

An offset is used for the second *<range>*.

Result 4:


B::Data.DRAW %Float.IeeeDblT sinewave++0xfff /Vector /Alternate 12 /Element 1 6 9

See also

■ Data.DRAWFFT	■ Data.DRAWXY	■ Data	■ Data.IMAGE
■ <trace>.DRAW	■ Var.DRAW		

▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'

[Example]

Format:	**Data.DRAWFFT** [**%**<*format*>] <*range*> <*scale*> <*fftsize*> [**/**<*window_option*>] [**/**<*input_option*>] [**/**<*display_option*>]
<*format*>:	**Decimal.**[<*width*>] (signed integer, y-axis labeled with decimal numbers) **DecimalU.**[<*width*>] (unsigned integer, y-axis labeled with decimal numbers) **Hex.**[<*width*>] (unsigned integer, y-axis labeled with hex numbers) **Float.**<*float_rep*> (floating point)
<*width*>:	**DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte**
<*float_rep*>:	**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeQuad** | **IeeeXt10** | **IeeeRev** **IeeeS** | **IeeeDblS** | **IeeeDblT** **MFFP** | **Pdp11** | **Pdp11Dbl** | **RTOSUH** | **RTOSUHD** **Dsp16** | **Dsp16C** | **Dsp16Fix** | **Dsp32Fix** **M56** | **M560** | **M561** | **LACCUM** **Fract8** | **Fract16** | **Fract24** | **Fract32** | **Fract48** | **Fract64** **UFract8** | **UFract16** | **UFract24** | **UFract32** | **UFract48** | **UFract64** | **Fract40G** **MICRO** | **MICRO64** | **MILLI** | **MILLI64** | **NANO64** | **PICO64**
<*scale*>:	Scale factor (normalization) for the x-axis as floating-point value. The spectrum will range from 0 to <*scale*>/2. E.g. **44100.0**
<*fftsize*>:	Number of points, must be power of 2, e.g. **128.** | **256.** | **512.**
<*window_ option*>:	**BLACKMAN** **HAMMING** **HANN**
<*input_ option*>:	**Real** (array consists of real numbers only) **COMPLEX** (array consists of complex number pairs)
<*display_ option*>:	**Vector** (default) | **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.

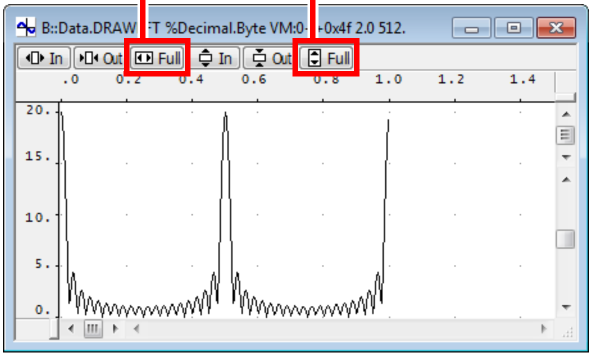
Example for Data.DRAWFFT

```
;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.
```

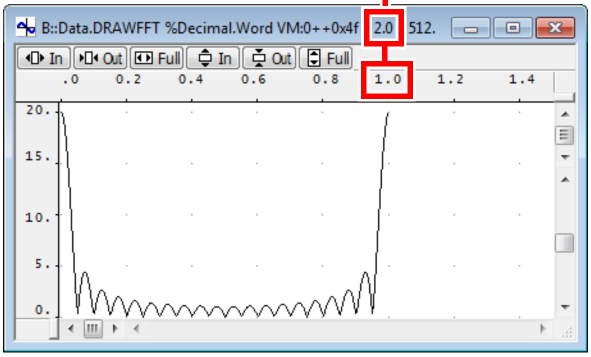
Result:

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

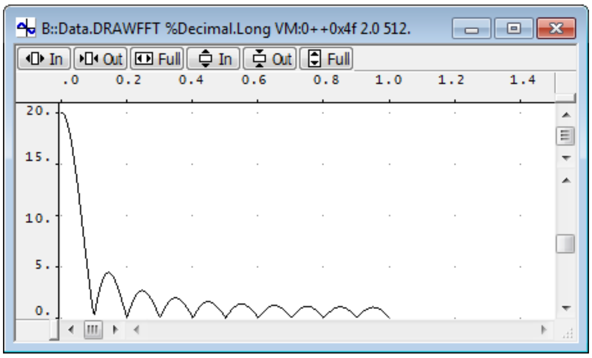
<scale> = 2.0
2.0 divided by 2 = 1.0


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

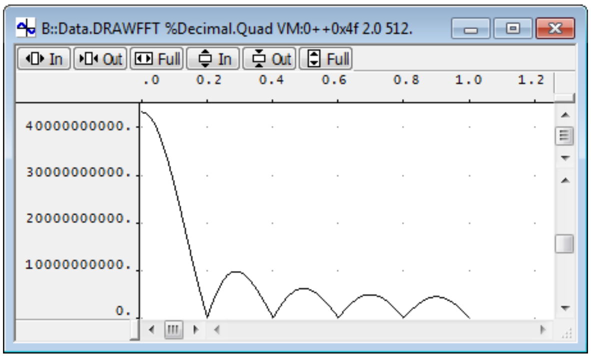
<format>: **Decimal.Byte**


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

<format>: **Decimal.Word**



<format>: **Decimal.Long**



<format>: **Decimal.Quad**

See also

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

▲ 'Release Information' in 'Release History'

[Example]

Format:	**Data.DRAWXY** [**%***<format>*] *<range_y> <range_x>* [*<scale>* [*<offset>*]] [**/***<options>*]
<format>:	**Decimal.**[*<width>*] (signed integer, y-axis labeled with decimal numbers) **DecimalU.**[*<width>*] (unsigned integer, y-axis labeled with decimal numbers) **Hex.**[*<width>*] (unsigned integer, y-axis labeled with hex numbers) **Float.***<float_rep>* (floating point)
<width>:	**DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte**
<float_rep>:	**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeQuad** | **IeeeXt10** | **IeeeRev** **IeeeS** | **IeeeDblS** | **IeeeDblT** **MFFP** | **Pdp11** | **Pdp11Dbl** | **RTOSUH** | **RTOSUHD** **Dsp16** | **Dsp16C** | **Dsp16Fix** | **Dsp32Fix** **M56** | **M560** | **M561** | **LACCUM** **Fract8** | **Fract16** | **Fract24** | **Fract32** | **Fract48** | **Fract64** **UFract8** | **UFract16** | **UFract24** | **UFract32** | **UFract48** | **UFract64** | **Fract40G** **MICRO** | **MICRO64** | **MILLI** | **MILLI64** | **NANO64** | **PICO64**
<options>:	**LOG:** display in logarithmic scale **Track:** Highlight position selected in **Data.dump** window **YX:** swap *<range_y> and <range_x>*
<display_ options>:	**Vector** (default) | **Points** | **Steps** | **Impulses**
<array_ options>:	**Alternate** *<dimension_of_array>*: **1** … **6** **Element** *<elements_to_draw_in_window>*: **1** … ***n***

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

Example

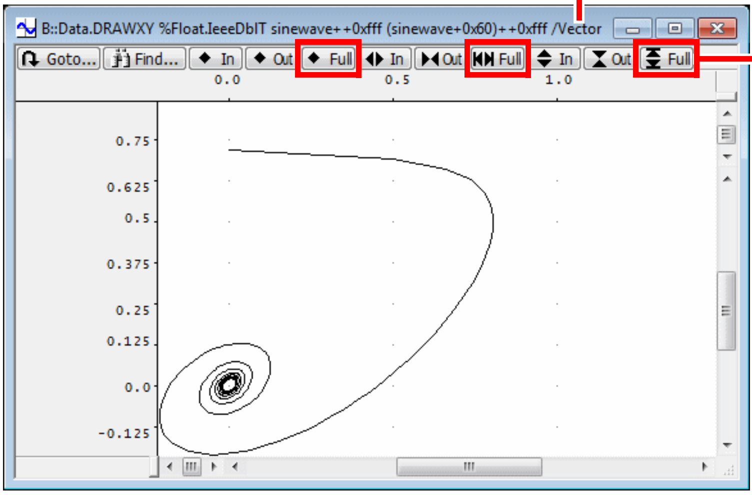
In this example, **Data.DRAWXY** is executed in the TRACE32 Instruction Set Simulator for ARM.

```
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

; See result 3.
;           <format>          <range>   <array_option>
Data.DRAWXY %Float.IeeeDblT 0++0xfff /Alternate 2 /Element 2 1
```

Result 1:

The **/Vector** option is used.

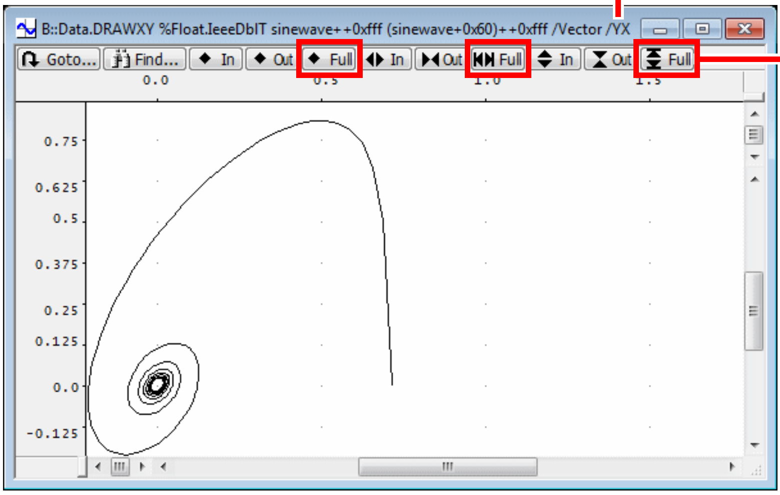
Click the three **Full** buttons.


B::Data.DRAWXY %Float.IeeeDblT sinewave++0xfff (sinewave+0x60)++0xfff /Vector

Result 2:

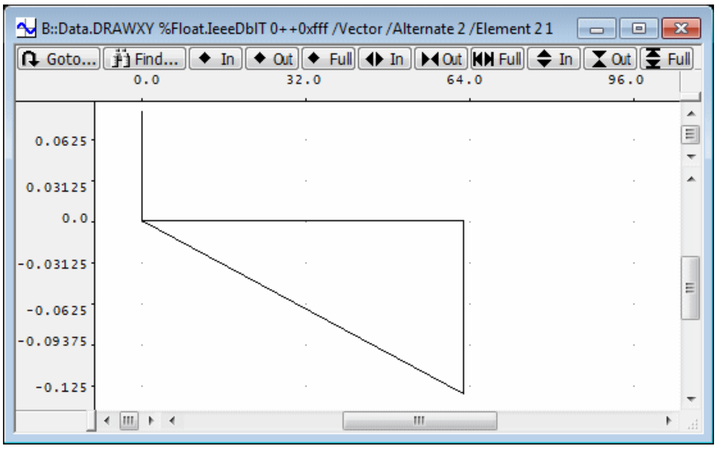
You can switch the x and y axis by using the **/YX** option.

Click the three **Full** buttons.



B::Data.DRAWXY %Float.IeeeDblT sinewave++0xfff (sinewave+0x60)++0xfff /Vector /YX

Result 3:



B::Data.DRAWXY %Float.IeeeDblT 0++0xfff /Vector /Alternate 2 /Element 2 1

See also

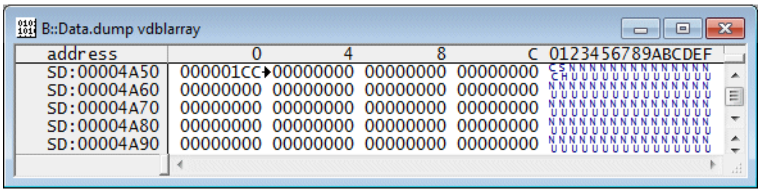
- Data.DRAW
- <trace>.DRAW
- Data.DRAWFFT
- Var.DRAW
- Data
- Data.IMAGE

Format:	**Data.dump** [*<address>* | *<range>*] [**/***<option>* …]
<option>: (**format**)	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** **BE** | **LE** | **Decimal** | **DecimalU** **NoHex** | **Hex** **NoAscii** | **Ascii** **Reverse**
<option>: (**standard**)	**DIALOG** **Track** **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
```

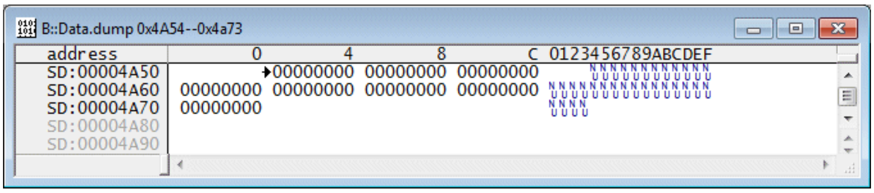


```
B::Data.dump vdblarray
     address              0        4        8        C 0123456789ABCDEF
SD:00004A50   000001CC▸00000000 00000000 00000000
SD:00004A60   00000000 00000000 00000000 00000000
SD:00004A70   00000000 00000000 00000000 00000000
SD:00004A80   00000000 00000000 00000000 00000000
SD:00004A90   00000000 00000000 00000000 00000000
```

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)
```

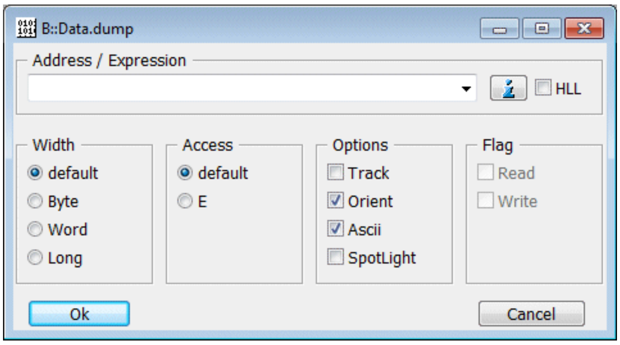
Var.RANGE(<hll_expression>)	Returns the address range occupied by the specified HLL expression



```
B::Data.dump 0x4A54--0x4a73
     address              0        4        8        C 0123456789ABCDEF
SD:00004A50            ▸00000000 00000000 00000000
SD:00004A60   00000000 00000000 00000000 00000000
SD:00004A70   00000000
SD:00004A80
SD:00004A90
```

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

```
Data.dump
```



B::Data.dump

Address / Expression: [ ] ☐ HLL

Width	Access	Options	Flag
● default	● default	☐ Track	☐ Read
○ Byte	○ E	✓ Orient	☐ Write
○ Word		✓ Ascii	
○ Long		☐ SpotLight	

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**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	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 10.

```
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
```

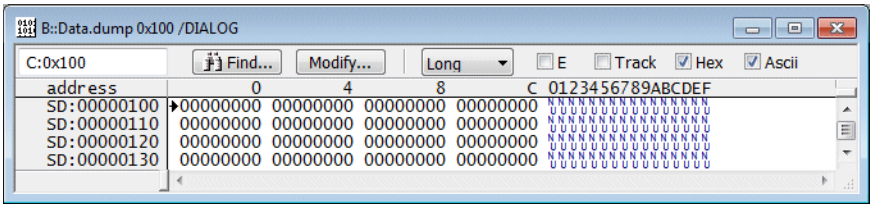
Hex (default) **NoHex**	If a memory dump is displayed TRACE32 PowerView displays an hex representation of the memory contents. This can be suppressed by the option **NoHex**.
Ascii (default) **NoAscii**	If a memory dump is displayed TRACE32 PowerView displays an ASCII representation of the memory contents. 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
```

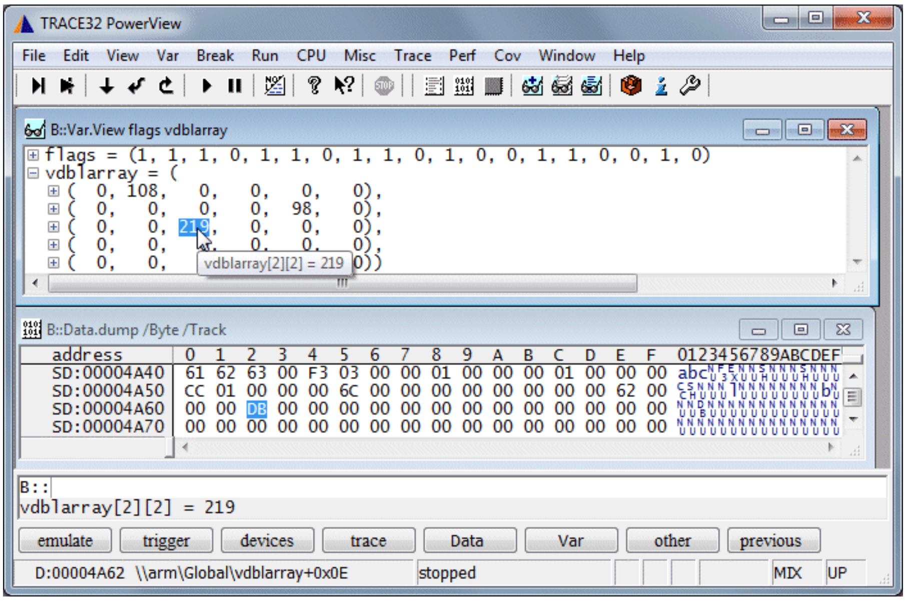


```
address           0        4        8        C 0123456789ABCDEF
SD:00000100 ▸00000000 00000000 00000000 00000000 ␀␀␀␀␀␀␀␀␀␀␀␀␀␀␀␀
SD:00000110  00000000 00000000 00000000 00000000 ␀␀␀␀␀␀␀␀␀␀␀␀␀␀␀␀
SD:00000120  00000000 00000000 00000000 00000000 ␀␀␀␀␀␀␀␀␀␀␀␀␀␀␀␀
SD:00000130  00000000 00000000 00000000 00000000 ␀␀␀␀␀␀␀␀␀␀␀␀␀␀␀␀
```

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.


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov Window Help
B::Var.View flags vdblarray
B::Data.dump /Byte /Track
emulate
trigger
devices
trace
Data
Var
other
previous
stopped
MIX
UP

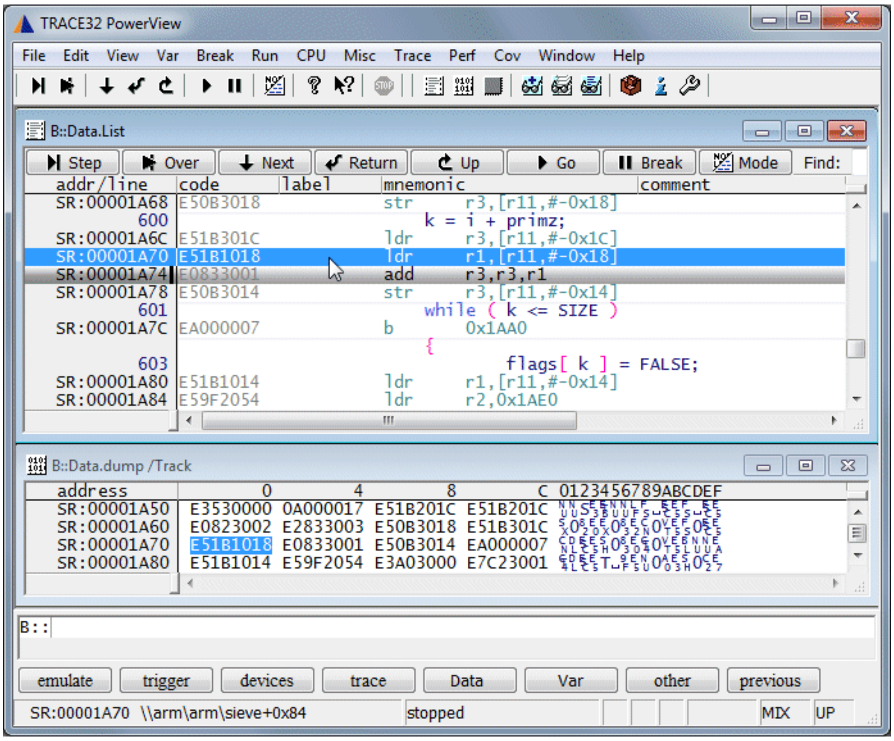
```
⊞ flags = (1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0)
⊟ vdblarray = (
  ⊞ ( 0, 108,   0,   0,   0,   0),
  ⊞ ( 0,   0,   0,   0,  98,   0),
  ⊞ ( 0,   0, 219,   0,   0,   0),
  ⊞ ( 0,   0,   0,   0,   0,   0),
  ⊞ ( 0,   0,   vdblarray[2][2] = 219   0))
```

```
address           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
SD:00004A40      61 62 63 00 F3 03 00 00 01 00 00 00 01 00 00 00  abc
SD:00004A50      CC 01 00 00 00 6C 00 00 00 00 00 00 00 62 00 00  l b
SD:00004A60      00 00 DB 00 00 00 00 00 00 00 00 00 00 00 00 00
SD:00004A70      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
B::
vdblarray[2][2] = 219
```

```
D:00004A62  \\arm\Global\vdblarray+0x0E
```

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.



```
addr/line      code      label    mnemonic                   comment
SR:00001A68    E50B3018           str       r3,[r11,#-0x18]
        600                                 k = i + primz;
SR:00001A6C    E51B301C           ldr       r3,[r11,#-0x1C]
SR:00001A70    E51B1018           ldr       r1,[r11,#-0x18]
SR:00001A74    E0833001           add       r3,r3,r1
SR:00001A78    E50B3014           str       r3,[r11,#-0x14]
        601                                 while ( k <= SIZE )
SR:00001A7C    EA000007           b         0x1AA0
                                            {
        603                                       flags[ k ] = FALSE;
SR:00001A80    E51B1014           ldr       r1,[r11,#-0x14]
SR:00001A84    E59F2054           ldr       r2,0x1AE0
```

```
address         0        4        8        C        0123456789ABCDEF
SR:00001A50   E3530000 0A000017 E51B201C E51B201C
SR:00001A60   E0823002 E2833003 E50B3018 E51B301C
SR:00001A70   E51B1018 E0833001 E50B3014 EA000007
SR:00001A80   E51B1014 E59F2054 E3A03000 E7C23001
```

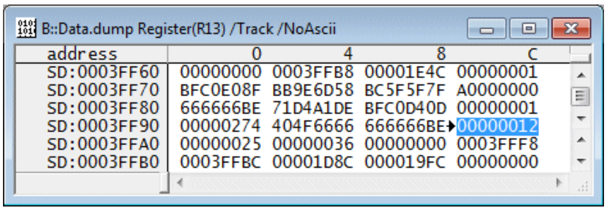
```
B::
```

```
SR:00001A70  \\arm\arm\sieve+0x84
```

Example for register tracking:

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

Register(<register_name>)	Returns register contents



B::Data.dump Register(R13) /Track /NoAscii

```
     address             0        4        8        C
 SD:0003FF60     00000000 0003FFB8 00001E4C 00000001
 SD:0003FF70     BFC0E08F BB9E6D58 BC5F5F7F A0000000
 SD:0003FF80     666666BE 71D4A1DE BFC0D40D 00000001
 SD:0003FF90     00000274 404F6666 666666BE>00000012
 SD:0003FFA0     00000025 00000036 00000000 0003FFF8
 SD:0003FFB0     0003FFBC 00001D8C 000019FC 00000000
```

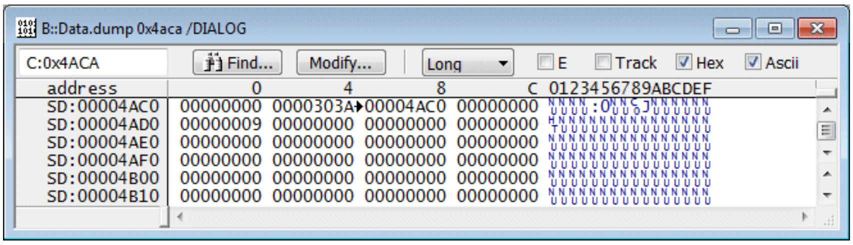
CORE *<number>*	Display memory dump from the perspective of the specified core (SMP debugging only)

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)
```

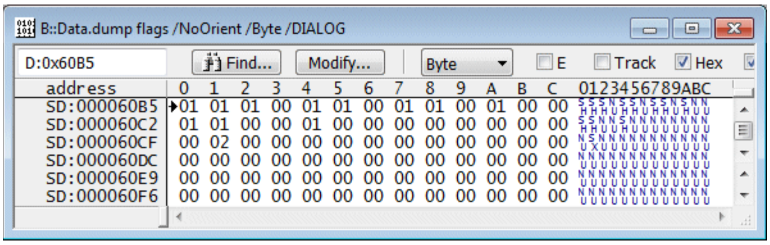
Orient (default)	The start address of the memory dump window matches to bounds of power of two. It is assumed that this is easier to read. The address specified in the **Data.dump** command is marked with a small arrow.
NoOrient	The dump starts exactly at the address specified in the **Data.dump** command.

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



```
address       0        4        8        C  0123456789ABCDEF
SD:00004AC0  00000000 0000303A▸00004AC0 00000000
SD:00004AD0  00000009 00000000 00000000 00000000
SD:00004AE0  00000000 00000000 00000000 00000000
SD:00004AF0  00000000 00000000 00000000 00000000
SD:00004B00  00000000 00000000 00000000 00000000
SD:00004B10  00000000 00000000 00000000 00000000
```

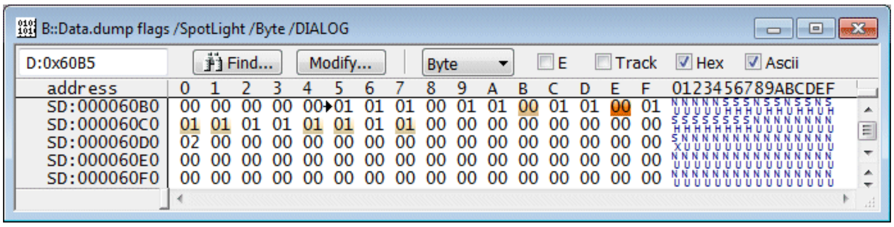
```
Data.dump flags /NoOrient /Byte /DIALOG
```



```
address       0  1  2  3  4  5  6  7  8  9  A  B  C  0123456789ABC
SD:000060B5 ▸01 01 01 00 01 01 00 01 01 00 01 00 00
SD:000060C2  01 01 00 00 01 00 00 00 00 00 00 00 00
SD:000060CF  00 02 00 00 00 00 00 00 00 00 00 00 00
SD:000060DC  00 00 00 00 00 00 00 00 00 00 00 00 00
SD:000060E9  00 00 00 00 00 00 00 00 00 00 00 00 00
SD:000060F6  00 00 00 00 00 00 00 00 00 00 00 00 00
```

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.

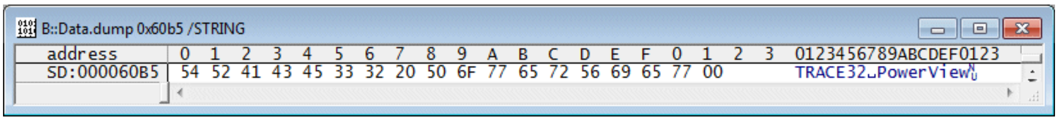
```
Data.dump flags /SpotLight /Byte /DIALOG
```



```
B::Data.dump flags /SpotLight /Byte /DIALOG
D:0x60B5        Find...  Modify...  Byte   E  Track  Hex  Ascii
address       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
SD:000060B0  00 00 00 00 00>01 01 01 00 01 01 00 01 01 00 01
SD:000060C0  01 01 01 01 01 01 01 01 00 00 00 00 00 00 00 00
SD:000060D0  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
SD:000060E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
SD:000060F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

STRING	Display memory dump of zero-terminated string.

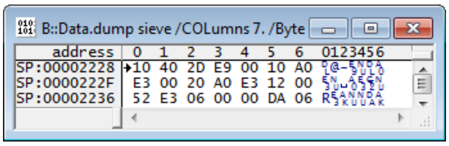
```
Data.dump 0x60b5 /STRING
```



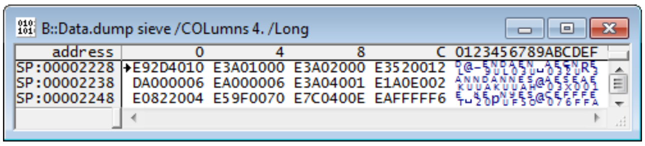
```
B::Data.dump 0x60b5 /STRING
address       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0  1  2  3  0123456789ABCDEF0123
SD:000060B5  54 52 41 43 45 33 32 20 50 6F 77 65 72 56 69 65 77 00        TRACE32 PowerView
```

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
```



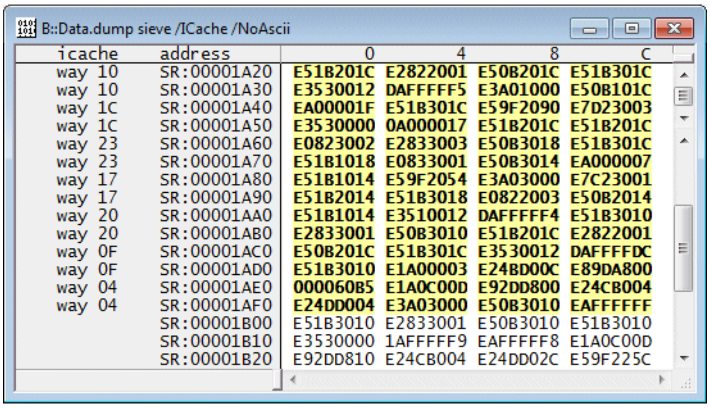
```
B::Data.dump sieve /COLumns 7. /Byte
    address   0  1  2  3  4  5  6   0123456
SP:00002228  10 40 2D E9 00 10 A0
SP:0000222F  E3 00 20 A0 E3 12 00
SP:00002236  52 E3 06 00 00 DA 06   R
```



```
B::Data.dump sieve /COLumns 4. /Long
    address         0        4        8        C  0123456789ABCDEF
SP:00002228  E92D4010 E3A01000 E3A02000 E3520012
SP:00002238  DA000006 EA000006 E3A04001 E1A0E002
SP:00002248  E0822004 E59F0070 E7C0400E EAFFFFF6
```

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

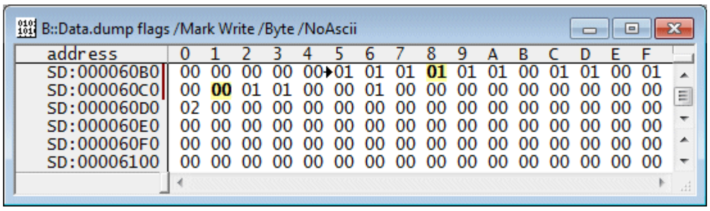
```
Data.dump sieve /ICache /NoAscii
```



```
B::Data.dump sieve /ICache /NoAscii
   icache       address          0        4        8        C
   way 10   SR:00001A20   E51B201C E2822001 E50B201C E51B301C
   way 10   SR:00001A30   E3530012 DAFFFFF5 E3A01000 E50B101C
   way 1C   SR:00001A40   EA00001F E51B301C E59F2090 E7D23003
   way 1C   SR:00001A50   E3530000 0A000017 E51B201C E51B201C
   way 23   SR:00001A60   E0823002 E2833003 E50B3018 E51B301C
   way 23   SR:00001A70   E51B1018 E0833001 E50B3014 EA000007
   way 17   SR:00001A80   E51B1014 E59F2054 E3A03000 E7C23001
   way 17   SR:00001A90   E51B2014 E51B3018 E0822003 E50B2014
   way 20   SR:00001AA0   E51B1014 E3510012 DAFFFFF4 E51B3010
   way 20   SR:00001AB0   E2833001 E50B3010 E51B201C E2822001
   way 0F   SR:00001AC0   E50B201C E51B301C E3530012 DAFFFFDC
   way 0F   SR:00001AD0   E51B3010 E1A00003 E24BD00C E89DA800
   way 04   SR:00001AE0   000060B5 E1A0C00D E92DD800 E24CB004
   way 04   SR:00001AF0   E24DD004 E3A03000 E50B3010 EAFFFFFF
            SR:00001B00   E51B3010 E2833001 E50B3010 E51B3010
            SR:00001B10   E3530000 1AFFFFF9 EAFFFFF8 E1A0C00D
            SR:00001B20   E92DD810 E24CB004 E24DD02C E59F225C
```

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
```



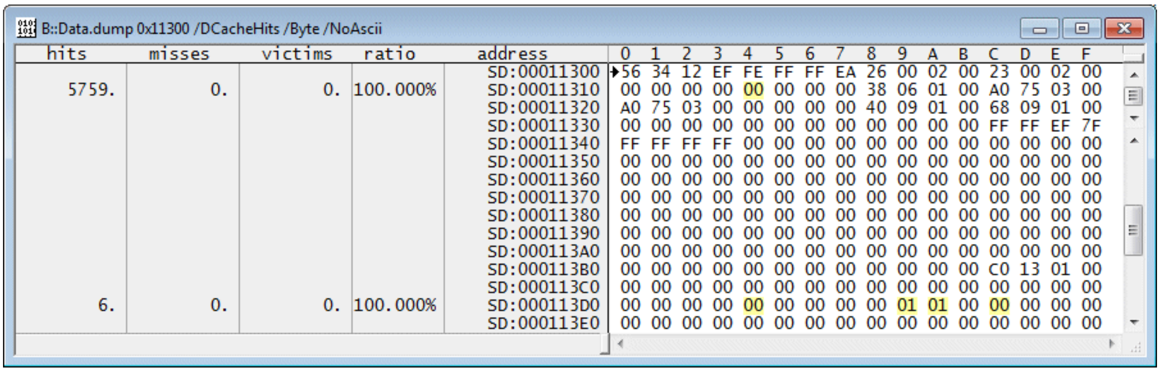
```
    address      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
    SD:000060B0  00 00 00 00 00>01 01 01 01 01 01 00 01 01 00 01
    SD:000060C0  00 00 01 01 00 00 01 00 00 00 00 00 00 00 00 00
    SD:000060D0  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    SD:000060E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    SD:000060F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    SD:00006100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

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

[Back to Top]

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.

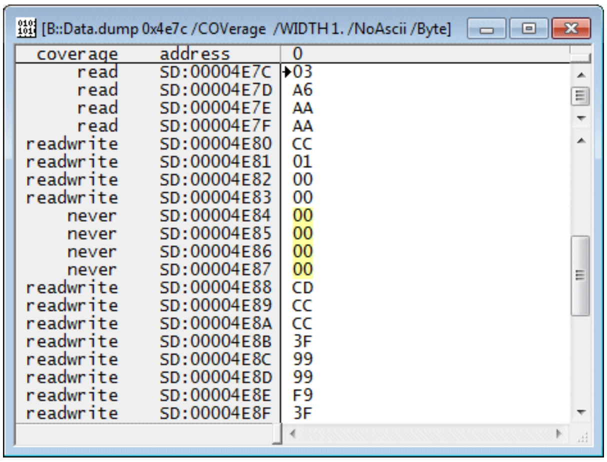


```
   hits      misses      victims   ratio        address     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
                                             SD:00011300 ▸56 34 12 EF FE FF FF EA 26 00 02 00 23 00 02 00
  5759.          0.           0. 100.000%    SD:00011310  00 00 00 00 00 00 00 00 38 06 01 00 A0 75 03 00
                                             SD:00011320  A0 75 03 00 00 00 00 00 40 09 01 00 68 09 01 00
                                             SD:00011330  00 00 00 00 00 00 00 00 00 00 00 00 FF FF EF 7F
                                             SD:00011340  FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00
                                             SD:00011350  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                             SD:00011360  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                             SD:00011370  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                             SD:00011380  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                             SD:00011390  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                             SD:000113A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                             SD:000113B0  00 00 00 00 00 00 00 00 00 00 00 00 C0 13 01 00
                                             SD:000113C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     6.          0.           0. 100.000%    SD:000113D0  00 00 00 00 00 00 00 00 00 01 01 00 00 00 00 00
                                             SD:000113E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The following option are 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
```



```
 coverage     address      0
     read     SD:00004E7C  03
     read     SD:00004E7D  A6
     read     SD:00004E7E  AA
     read     SD:00004E7F  AA
readwrite     SD:00004E80  CC
readwrite     SD:00004E81  01
readwrite     SD:00004E82  00
readwrite     SD:00004E83  00
    never     SD:00004E84  00
    never     SD:00004E85  00
    never     SD:00004E86  00
    never     SD:00004E87  00
readwrite     SD:00004E88  CD
readwrite     SD:00004E89  CC
readwrite     SD:00004E8A  CC
readwrite     SD:00004E8B  3F
readwrite     SD:00004E8C  99
readwrite     SD:00004E8D  99
readwrite     SD:00004E8E  F9
readwrite     SD:00004E8F  3F
```

See also

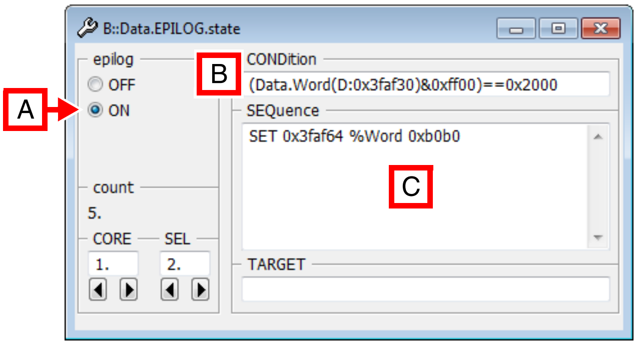
■ Data	■ Data.Assemble	■ Data.CHAIN	■ Data.Find
■ Data.In	■ Data.PATTERN	■ Data.Print	■ Data.STRING
■ Data.TABle	■ Data.Test	■ Data.USRACCESS	■ Data.View
■ List	■ SETUP.DUMP	■ SETUP.FLIST	■ SETUP.TIMEOUT
■ sYmbol.INFO	■ DUMP	■ SETUP.ASCIITEXT	❑ ADDRESS.OFFSET()
❑ ADDRESS.SEGMENT()	❑ ADDRESS.STRACCESS()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

▲ 'Data Access' in 'EPROM/FLASH Simulator'
▲ 'Data Functions' in 'General Function Reference'
▲ 'ADDRESS Functions' in 'General Function Reference'
▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'
▲ 'Data and Program Memory' in 'Training FIRE Basics'
▲ 'Data and Program Memory' in 'Training ICE Basics'

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** input field of the window using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** input field of the window 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.EPILOG.CONDition	■ Data.EPILOG.CORE	■ Data.EPILOG.OFF	■ Data.EPILOG.ON
■ Data.EPILOG.RESet	■ Data.EPILOG.SELect	■ Data.EPILOG.SEQuence	■ Data.EPILOG.state
■ Data.EPILOG.TARGET	■ Data	■ Data.PROLOG	

▲ 'Release Information' in 'Release History'

Data.EPILOG.CONDition

Define condition for data epilog

Format:	**Data.EPILOG.CONDition** *<condition>*

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

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
```

NOTE: **Data.EPILOG.CONDition** supports only these functions:

- **Data.Byte()** and its short form **D.B()**
- **Data.Long()** and its short form **D.L()**
- **Data.Word()** and its short form **D.W()**

See also

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

Data.EPILOG.CORE

Select core for data epilog

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. 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
- 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
- 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 ■ 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 ■ 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
- 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> <inc>* At the first time performed, write *<start>* to *<address>*. *<start>* is incremented by *<inc>* 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:

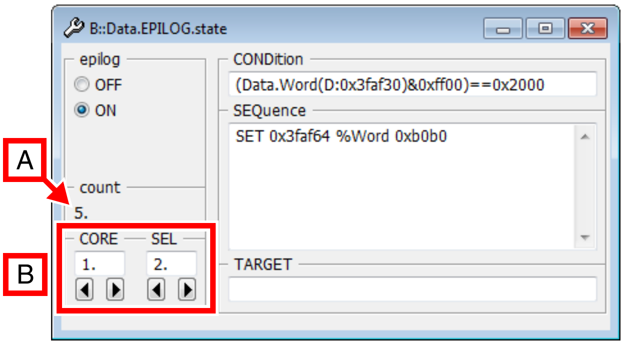
```
;Set peripheral register to 0 when halted, 1 when starting
Data.EPILOG.SEQuence SET 0x3faf50 %Long 0x00000000
Data.PROLOG.SEQuence SET 0x3faf50 %Long 0x00000001

;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

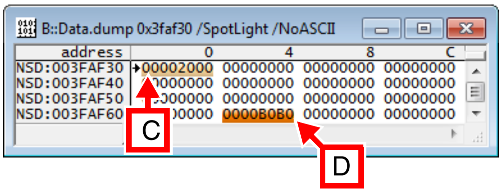
■ Data.EPILOG ■ Data.EPILOG.state

Format: **Data.EPILOG.state**

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



```
B::Data.EPILOG.state
epilog
  OFF
  [✓ ON]
count
  5.
CORE 1.   SEL 2.
CONDition
  (Data.Word(D:0x3faf30)&0xff00)==0x2000
SEQuence
  SET 0x3faf64 %Word 0xb0b0
TARGET
```



```
B::Data.dump 0x3faf30 /SpotLight /NoASCII
     address         0        4        8        C
NSD:003FAF30   00002000 00000000 00000000 00000000
NSD:003FAF40    0000000 00000000 00000000 00000000
NSD:003FAF50     000000 00000000 00000000 00000000
NSD:003FAF60      00000 0000B0B0 00000000 00000000
```

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
- 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 '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.

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

See also

- Data.EPILOG
- Data.EPILOG.state

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

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

Data.Find commands without parameters 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**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	See **"Keywords for <width>"**, page 10.

Examples:

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

; search the next byte x3f
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.
ADDRESS.OFFSET()	Extracts the hex-address from the address object returned by TRACK.ADDRESS().

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

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

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

IF FOUND()
     PRINT "Data found at address " ADDRESS.OFFSET(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 ": " ADDRESS.OFFSET(TRACK.ADDRESS())
     &i=&i+1
)
WHILE FOUND()

CLOSE #1

TYPE result.txt

ENDDO
```

See also

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

▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'

Data.FindCODE

Execute command on specified code type

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

The command **Data.FindCODE** processes the source code in the specified *<addr_range>* in order to find the specified instruction type.

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

The command **Data.FindCODE** is mainly used to automatically set the **statistic markers**.

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 *"
```

See also

■ Data

▲ 'Release Information' in 'Release History'

Data.GOTO — Specify reference address for address tracking

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

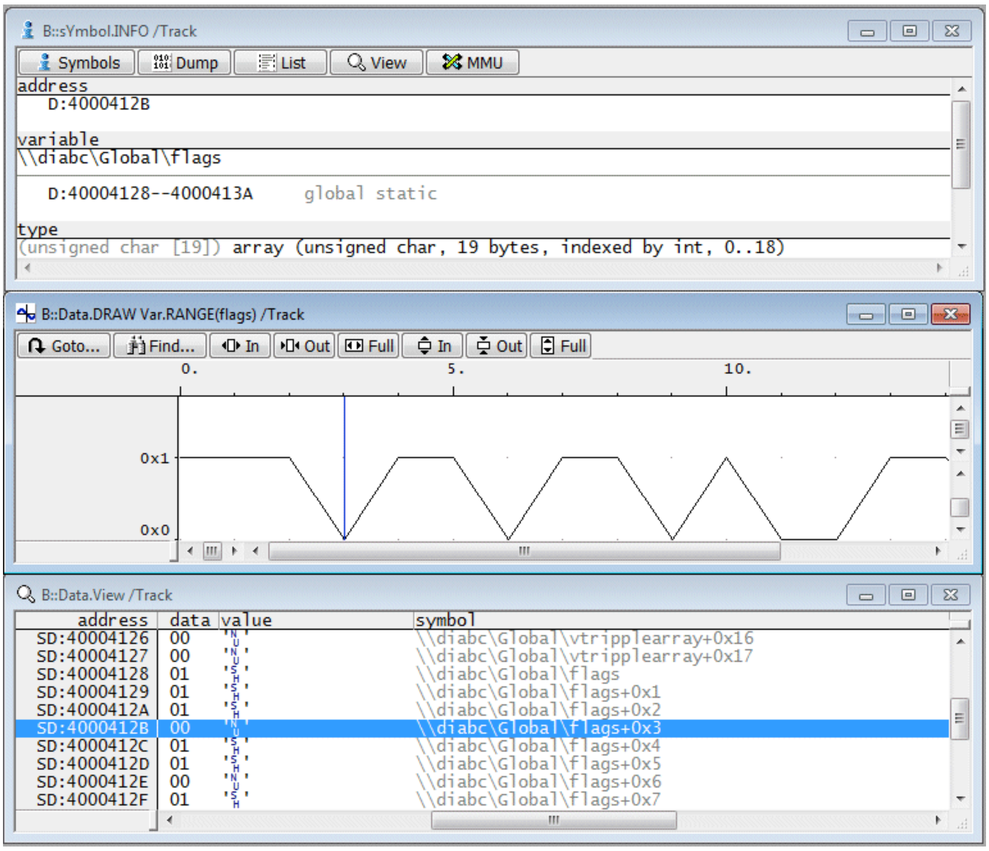
The given address is used for tracking the windows (like **FIND**/**ComPare** commands).

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
```



```
address
   D:4000412B

variable
\\diabc\Global\flags

   D:40004128--4000413A      global static

type
(unsigned char [19]) array (unsigned char, 19 bytes, indexed by int, 0..18)
```

```
     address  data value      symbol
SD:40004126   00   'NU'       \\diabc\Global\vtripplearray+0x16
SD:40004127   00   'NU'       \\diabc\Global\vtripplearray+0x17
SD:40004128   01   'SH'       \\diabc\Global\flags
SD:40004129   01   'SH'       \\diabc\Global\flags+0x1
SD:4000412A   01   'SH'       \\diabc\Global\flags+0x2
SD:4000412B   00   'NU'       \\diabc\Global\flags+0x3
SD:4000412C   01   'SH'       \\diabc\Global\flags+0x4
SD:4000412D   01   'SH'       \\diabc\Global\flags+0x5
SD:4000412E   00   'NU'       \\diabc\Global\flags+0x6
SD:4000412F   01   'SH'       \\diabc\Global\flags+0x7
```

See also

■ Data.GREP	■ Data	■ Data.Find	■ FIND
■ WinFIND	❏ TRACK.ADDRESS()		

▲ 'ADDRESS Functions' in 'General Function Reference'
▲ 'Data Functions' in 'General Function Reference'
▲ 'Program and Data Memory' in 'ICE User's Guide'

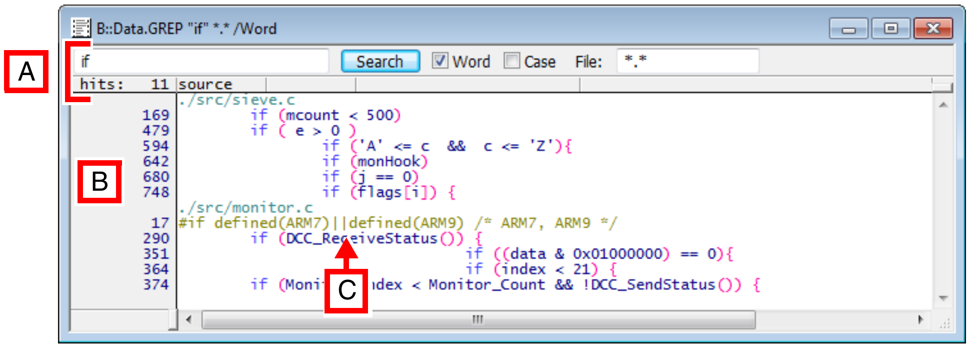
Format:	**Data.GREP** *<string>* [*<file>*] [**/***<option>*]
<option>:	**Word** **Case** *<other_options>*

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 **Data.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
```



```
B::Data.GREP "if" *.* /Word
if                    [Search] [✓ Word] [ Case] File: *.*
hits:   11 source
           ./src/sieve.c
      169          if (mcount < 500)
      479          if ( e > 0 )
      594                  if ('A' <= c  &&  c <= 'Z'){
      642                  if (monHook)
      680                  if (j == 0)
      748                  if (flags[i]) {
           ./src/monitor.c
       17  #if defined(ARM7)||defined(ARM9) /* ARM7, ARM9 */
      290          if (DCC_ReceiveStatus()) {
      351                                  if ((data & 0x01000000) == 0){
      364                                  if (index < 21) {
      374          if (Moni    dex < Monitor_Count && !DCC_SendStatus()) {
```

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 ■ Data.Find ■ APU.GREP
■ FIND ■ WinFIND ■ WinOverlay

▲ 'The Symbol Database' in 'Training HLL Debugging'

[Examples]

Format:	**Data.IMAGE** *<address> <horiz> <vert>* [*<scale>* | **/***<format>* | **/***<option>*]
<format>:	**MONO** | **CGA** | **GrayScale8** | **JPEG** **RGB111** | **RGB555** | **RGB555LE** | **RGB565** | **RGB565LE** | **RGB888** | **RGB888LE** | **RGBX888** | **RGBX888LE** | **RGBCUSTOM** **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.
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 red color, bit 1 is green, bit 2 is blue. 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:

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

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

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

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

See also

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

▲ 'Release Information' in 'Release History'

Data.In

Format:	**Data.In** *<address>* [*<repetitions>*] [**/***<options>*]
<options>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** **BE** | **LE** **Repeat** | **INCrement** | **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**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	See **"Keywords for <width>"**, page 10.
Repeat	Repeats the input endlessly, e.g. for external measurements.
INCrement	Address is incremented by *<width>* with each repeated access.

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

▲ 'ADDRESS Functions' in 'General Function Reference'
▲ 'Data Functions' in 'General Function Reference'
▲ 'Program and Data Memory' in 'ICE User's Guide'

Data.List

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** *<filename>* [*<address>*] [*<range>*] [**/***<load_option>*] **Data.LOAD.auto** *<filename>* [*<address>*] [*<range>*] [**/***<load_option>*]
<generic_load _option>:	**Verify** | **PVerify** | **NoVerify** | **CHECKLOAD** [*<address_range>*] **ComPare** | **DIFF** | **CHECKONLY** [*<address_range>*] **ZIPLOAD** [*<address_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>* | *<partname>* **StripBeforePART** *<pattern>* **TASK** *<magic>* **NAME** *<name>* **MAP** | **DIAG** **Include** | **NoInclude** **COLumns** | **MACRO** **CYGDRIVE** **SingleLine** | **SingleLineAdjacent** …
<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.<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.<format>**. There are also options which are only usable for a specific file format. These options are only available of **Data.LOAD.<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
<range>	• 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

ALIGN	tbd.
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.
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 try's 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	tbd.
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	tbd.
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. The symbols are not accessible after a load with this option, till a load without this option is made.
MultiLine	tbd.
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).
NOAGENTVERIFY	tbd.
NoBreak (E)	No automatic HLL breakpoint setting. The HLL breakpoint can set also with the **Break.SetHll** command. This allows module, or function selective HLL debugging. Modules without HLL breakpoint are executed in real-time during HLL steps. Only relevant for TRACE32-ICE.
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 try's 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.
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	tbd.
TypesOnly	tbd.
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 about 1/16 of all memory writes 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.

```
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.

```
; 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())
)
…
```

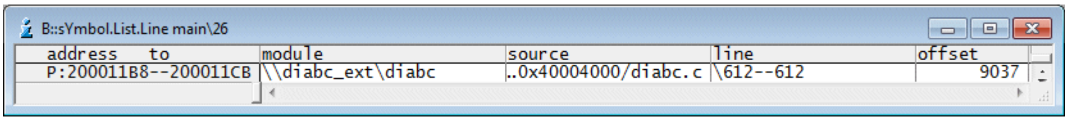
Options to improve the download speed for debug ports with slow download	
DIFFLOAD	Downloads only changed code in a compressed form via a target agent. The target agent is part of the TRACE32 software and is automatically loaded at the end of the data. If this is not practicable it is also possible to 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 an at least 256K byte *<address_range>* for the target agent.

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

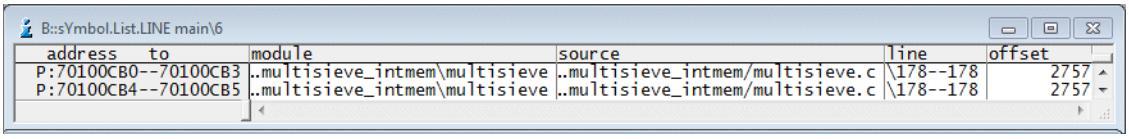
The debug information loaded from *<file>* provides the mapping between HLL source code lines and the blocks of assembler code generated for these lines. Their are mainly three types of mapping:

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



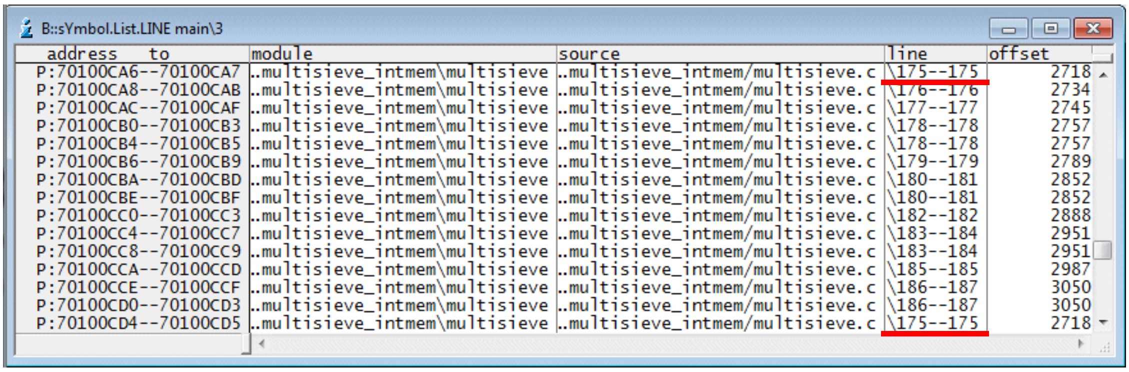
```
B::sYmbol.List.Line main\26
address    to            module            source                 line        offset
P:200011B8--200011CB   \\diabc_ext\diabc  ..0x40004000/diabc.c   \612--612     9037
```

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
```

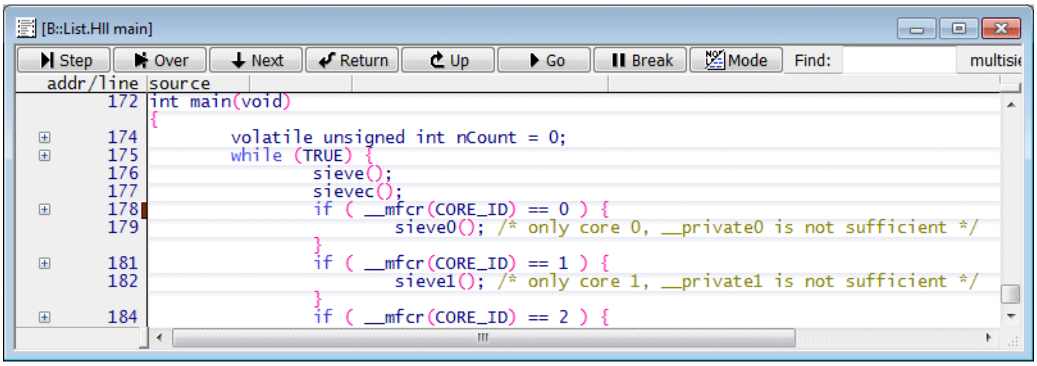
3. Two or more detached blocks of assembler code are generated for an HLL line.



```
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
```

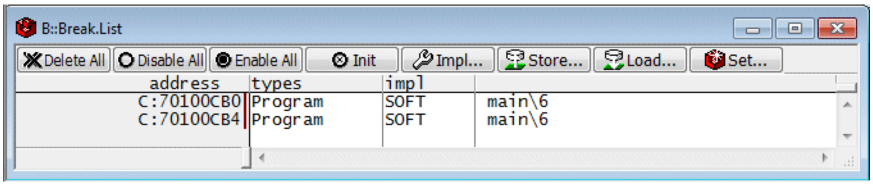
It has the following effects on debugging if more the on block of assembler code is generated for an HLL line:

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



```
addr/line source
      172 int main(void)
          {
⊞     174         volatile unsigned int nCount = 0;
⊞     175         while (TRUE) {
      176                 sieve();
      177                 sievec();
⊞     178                 if ( __mfcr(CORE_ID) == 0 ) {
      179                         sieve0(); /* only core 0, __private0 is not sufficient */
                          }
⊞     181                 if ( __mfcr(CORE_ID) == 1 ) {
      182                         sieve1(); /* only core 1, __private1 is not sufficient */
                          }
⊞     184                 if ( __mfcr(CORE_ID) == 2 ) {
```

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



```
     address  types     impl
  C:70100CB0  Program   SOFT    main\6
  C:70100CB4  Program   SOFT    main\6
```

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.

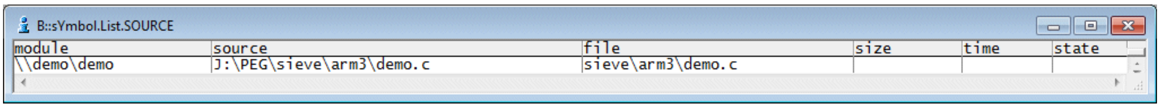
Examples for the options PATH, StripPART, and SOURCEPATH

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

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

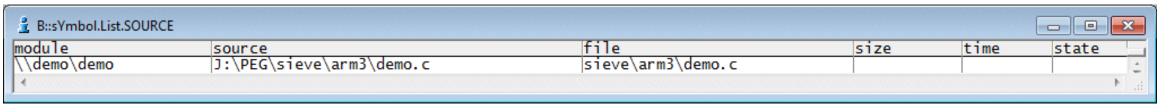
```
Data.LOAD.Elf demo.axf /StripPART 4. /SOURCEPATH ~~/demo/hardware/imx53
```

```
Data.LOAD.Elf demo.axf /StripPART 2.
```



```
module       source                    file                 size   time   state
\\demo\demo  J:\PEG\sieve\arm3\demo.c  sieve\arm3\demo.c
```

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

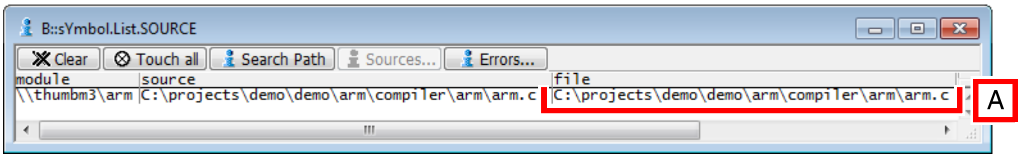


```
module       source                    file                 size   time   state
\\demo\demo  J:\PEG\sieve\arm3\demo.c  sieve\arm3\demo.c
```

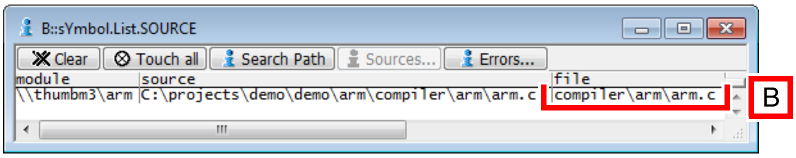
Example for the option StripBeforePART

[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
```



```
module         source                                   file
\\thumbm3\arm  C:\projects\demo\demo\arm\compiler\arm\arm.c  C:\projects\demo\demo\arm\compiler\arm\arm.c
```



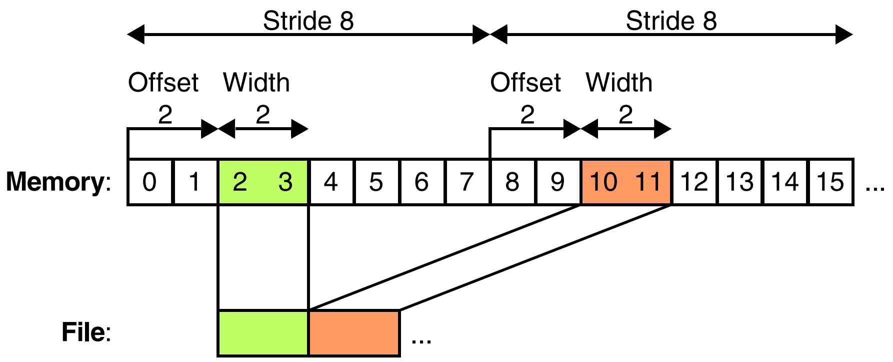
```
module         source                                   file
\\thumbm3\arm  C:\projects\demo\demo\arm\compiler\arm\arm.c  compiler\arm\arm.c
```

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
```

Program Loading (TRACE32-ICE)

Some CPU types have different storage classes (function codes) for program and data access. The CPU never writes to program areas if the program is running correctly. On downloading programs the CPU writes to this memory area, which will force bus errors or bus time-out. To turn around this problems load the program direct to the emulation memory by using the **/DualPort** option. Therefore no memory access by the emulation CPU is made (dual-port access). On microcontrollers which have no possibility to write to program area loading is automatically rerouted to dual-port access.

See also

■ Data.LOAD.AIF	■ Data.LOAD.AOUT	■ Data.LOAD.ASAP2	■ Data.LOAD.Ascii
■ Data.LOAD.AsciiHex	■ Data.LOAD.AsciiOct	■ Data.LOAD.AVocet	■ Data.LOAD.BDX
■ Data.LOAD.Binary	■ Data.LOAD.BounD	■ Data.LOAD.CDB	■ Data.LOAD.COFF
■ Data.LOAD.COMFOR	■ Data.LOAD.CORE	■ Data.LOAD.COSMIC	■ Data.LOAD.CrashDump
■ 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.Jedec
■ 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.SDS
■ Data.LOAD.Srecord	■ Data.LOAD.sYm	■ Data.LOAD.SysRof	■ Data.LOAD.TEK
■ Data.LOAD.TekHex	■ Data.LOAD.Ubrof	■ Data.LOAD.VersaDos	■ Data.LOAD.XCoff
■ Data	■ List	❏ ADDRESS.isDATA()	

▲ 'Data Access' in 'EPROM/FLASH Simulator'
▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'
▲ 'Load the Application Program' in 'Training HLL Debugging'
▲ 'Load the Application Program' in 'Training HLL Debugging'
▲ 'Starting-up the Emulator' in 'Training ICE Basics'

Format Specific Data.LOAD Commands and Options

The following **Data.LOAD.***<format>* 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** *<filename>* [*<class>*] [**/***<option>*]
<option>:	**Puzzled** **AnySym** **PACK** **FASTPACK** **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.
FASTPACK	Same as above. Fastpack is faster and more efficient than **PACK**, but it requires unique type names in the whole application. Fastpack assumes that types of identical name represent the same structure.
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

▲ 'Data Access' in 'EPROM/FLASH Simulator'

Data.LOAD.AOUT — Load a.out file

Format:	**Data.LOAD.AOUT** *<filename>* [*<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 Access' in 'EPROM/FLASH Simulator'

Data.LOAD.ASAP2 — Load ASAP2 file

Format:	**Data.LOAD.ASAP2** *<filename>* [**/***<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 'Release History'

Data.LOAD.Ascii — Load ASCII file

Format:	**Data.LOAD.Ascii** *<filename>* [*<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.AsciiHex — Load hex file

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

Loads a file in a simple Ascii file format.

See also

■ Data.LOAD

▲ 'Data Access' in 'EPROM/FLASH Simulator'

Data.LOAD.AsciiOct — Load octal file

Format:	**Data.LOAD.AsciiOct** *<filename>* [**/***<option>*] **Data.LOAD.AscciOctA** *<filename>* [**/***<option>*] **Data.LOAD.AscciOctP** *<filename>* [**/***<option>*] **Data.LOAD.AscciOctS** *<filename>* [**/***<option>*]
<option>:	**OFFSET** *<offset>*

Loads a file in a simple Ascii file format.

See also

■ Data.LOAD

Data.LOAD.AVocet — Load AVOCET file

Format:	**Data.LOAD.AVocet** *<filename>* [*<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** *<filename> <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

Data.LOAD.Binary Load binary file

[Examples]

Format:	**Data.LOAD.Binary** *<filename> <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>*.
<option>	For a description of the generic options, click **<generic_load_option>**.
<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.

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 'Release History'

Data.LOAD.BounD — Load BOUND file

Format:	**Data.LOAD.BounD** *<filename>* [*<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.CDB — Load SDCC CDB file format

Format:	**Data.LOAD.CDB** *<filename>* [*<class>*] [**/***<option>*]
<option>:	**IntelHexFile** *<binary_file_name>* **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 /NIHF               ; Load symbol information only

; 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** *<filename>* [*<class>*] [**/***<option>*]
<option>:	**FPU** **MCS2** | **ICC** | **MOTC11** | **GHILLS** | **GNUCPP** **INT16** **SHORT8** **ALLLINE** **Puzzled** **PACK** **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.
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.
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 'Release History'

Data.LOAD.COMFOR — Load COMFOR (TEKTRONIX) file

Format:	**Data.LOAD.COMFOR** *<filename>* [**/***<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

Data.LOAD.CORE

Load Linux core dump file

Format:	**Data.LOAD.CORE** *<filename>* [**/***<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 'Release History'

Format:	**Data.LOAD.COSMIC** *<filename>* [*<class>*] [**/***<option>*]
<option>:	**INT16** **SCHAR** | **SPREC** **MODD** | **MODP** | **MODF** (only 68HC16) **IEEE** **MMU** **REV** **ADDBANK** **LOGLOAD** *<generic_load_option>*

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** *<filename>* [**/***<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 ’Release History’

Format:	**Data.LOAD.DBX** *<filename> <code> <data>* [*<sym>*] [**/***<option>*]
<option>:	**CPP** **AnySym** **CFRONT** **GHILLS** **PACK** **FASTPACK** **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.
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.
FASTPACK	Same as above. Fastpack is faster and more efficient than **PACK**, but it requires unique type names in the whole applications. Fastpack assumes that types of identical name represent the same structure.
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** *<filename>* [*<memory_class>* | *<offset>* | *<range>*] [**/***<option>*]
<option>:	**FPU** | **NOFPU** **AnySym** | **ZeroSym** **NOMERGE** | **NOEXTERNALS** **STRIPPREFIX** *<prefix>* **PACK** | **FASTPACK** **CODESEC** **LOGLOAD** | **PHYSLOAD** **GlobTypes** | **NoGlobTypes** | **FastGlobTypes** | **SlowGlobTypes** **GHS** | **GNU** | **IAR** | **METAWARE** | **MetroWerks** | **MRI** | **WRS** | **CFRONT** **STARTTHUMB** **STABS** | **DWARF** | **DWARF2** **NOLINES** | **NOCONST** | **NOMETHODS** **CPP** | **GNUCPP** **IA64**
<option>: (cont.)	**CHILL** | **PASCAL** **ALTBITFIELDS** **ALTRELOC2** **ENUMCONSTS** **ABSLINES** **ABSLIFETIMES** **ForceLines** **FUNClines** **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>*
<option>: (cont.)	**LOCATEAT** *<address>* **RemoveModuleInSection** *<section_name>* **RemoveModuleAfterSection** *<section_name>* **OVERLAY** **NoFILEHDR** | **NoPHDRS** *<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.

Options:

<option>	For a description of the generic options, click **<generic_load_option>**.
ABSLIFETIMES	tbd.
ABSLINES	tbd.
ALTBITFIELDS	This option might solve problems with regards to the display of bitfields.
ALTRELOC2	tbd.
AnySym	Load 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
ChainedAbbrev	tbd.
ChainedLines	tbd.
ChainedStab	tbd.
ChainedStab4	tbd.

CHILL	Set this option if your program was coded in CHILL (CCITT High Level Language).
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.
CPP	Must be set when loading an ELF file with symbol information in STABS format for C++.
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.
FastGlobTypes	tbd.
FASTPACK	Same as **PACK**. Fastpack is faster and more efficient than **PACK**, but it requires unique type names in the whole applications. Fastpack assumes that types of identical name represent the same structure.
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]
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.
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.
NOCONST	Suppresses the load of "const" variables.
NOEXTERNALS	Ignore declaration of external variables in DWARF debug information.
NoFILEHDR	Suppress 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.
NOMERGE	Symbol information of the different formats (DWARF/STABS) are not merged together. The default is to merge the symbol information.
NOMETHODS	tbd.
NoPHDRS	Suppress 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.
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.
RELOCTYPE *<type>*	tbd.

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>*	tbd.
RemoveModuleIn-Section *<secname>*	tbd.
REV	Reverse bit fields. Must be set when the compiler option was set. (only HC11/HC12)
SlowGlobTypes	tbd.
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).
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.

ARM architecture:

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

Intel® x86 architecture:

REAL	tbd.
REAL32	tbd.
SMALLREAL	tbd.
SMALLREAL32	tbd.

M68K and Coldfire architecture:

ALTRELOC	tbd.

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>*	tbd.
DynamicNMF	tbd.

DSP56K architecture:

LARGE	tbd.

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
```

See also

■ Data.LOAD

▲ 'CPU specific Commands' in 'MMDSP Debugger'
▲ 'CPU specific Commands' in 'MMDSP NEXUS Debugger and Trace'
▲ 'Release Information' in 'Release History'

Data.LOAD.ESTFB — Load EST flat binary

Format:	**Data.LOAD.ESTFB** *<filename>* [*<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** *<filename>* [*<access>* | *<address>*] [**/***<option>*]
<option>:	**AnySym** **CPP** **ENUMCONSTS** **RELOC** *<section_name>* **AT** *<address>* **RELOC** *<section_name>* **AFTER** *<section_name>* **FPU** **LOCATEAT** *<address>* **NoMMU** *<generic_load_option>*

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

▲ 'Release Information' in 'Release History'

Data.LOAD.FIASCO — Load FIASCO BB5 file

Format:	**Data.LOAD.FIASCO** *<filename>* [*<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** *<filename>* [*<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** *<filename>* [*<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** *<filename>* [*<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** *<filename>* [*<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** *<filename>* [*<address>* | *<access>*] [**/***<option>*]
<option>:	**FPU** | **NOCLR** | **STandarD** **InLine** | **NoInLine** **INT16** **NoMATCH** **MSECtion** **PACK** | **FASTPACK** **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.
FASTPACK	Same as above. Fastpack is faster and more efficient than **PACK**, but it requires unique type names in the whole applications. Fastpack assumes that types of identical name represent the same structure.
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).
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.
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 'Release History'

Data.LOAD.IntelHex — Load INTEL-HEX file

Format:	**Data.LOAD.IntelHex** *<filename>* [*<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 'Release History'

Data.LOAD.Jedec — Load JEDEC file

Format:	**Data.LOAD.Jedec** *<filename>* [**/***<option>*]
<option>:	*<generic_load_option>*

Loads a file in Jedec format. Not for use by debuggers.

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

See also

■ Data.LOAD

[Examples]

Format:	**Data.LOAD.MachO** *<filename>* [*<class>*] [**/***<option>*]
<option>:	**DebugFile** *<binary_file_name>* **NoDebugFile** **DWARF** | **STABS** **NOCONST** **NOMERGE** **UUID** **IgnoreARCH** **ARCHNumber** *<value>* *<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.
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

Data.LOAD.MAP Load MAP file

Format:	**Data.LOAD.MAP** *<filename>* [*<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

Data.LOAD.MCDS Load MCDS file

Format:	**Data.LOAD.MCDS** *<filename>* [*<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** *<filename>* [*<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

Data.LOAD.OAT — Load OAT file

Format:	**Data.LOAD.OAT** *<filename>* [*<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 'Release History'

Format:	**Data.LOAD.OMF** *<filename>* [*<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.

Special Requirements for Intel PL/M-86 and PL/M-51

The listing file of a module must exist in a directory in the search path. The relation between object file line number and source is taken from this file. The extra tool ~~/demo/i186/compiler/intel/cline.exe can be used to patch the line numbers in the object file after compilation. This tool has the advantage that the list files are not required by the debugger. Similar converters are also available from other third party vendors.

The following example loads an OMF-86 file generated by a PL/M-86 compiler. The source text is read from the list files of the compiler:

```
plm86 module.plm debug
link86 …
loc86 …

E::Data.LOAD.Omf main.abs /LST
```

The next example loads directly the source files. The object files have been fixed by cline. The list files are not required for debugging.

```
plm86 module.plm debug
cline module.obj
delete module.lst
link86 …
loc86 …

E::Data.LOAD.Omf main.abs /PLM
```

If the application consists of files from Intel-C and PL/M compilers and the object files of PL/M are converted, the following command loads the file:

```
E::Data.LOAD.OMF main.abs /IC86 /MIXPLM
```

See also

■ Data.LOAD ■ Data.SAVE.Omf

Data.LOAD.Omf2 Load OMF-251 files

Format:	**Data.LOAD.Omf2** *<filename>* [**/***<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** *<filename> <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

Data.LOAD.PureHex Load hex-byte file

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.

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 'Release History'

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

Format:	**Data.LOAD.REAL** *<filename>* [**/***<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** *<filename>* [*<code>*] [*<data>*] [**/***<option>*]
<option>:	**NoSTB** **NoDBG** **NoMOD** **MAP** **FPU** **CPP** **CFRONT** **TURBOPACK** *<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.
TURBOPACK	Saves memory space by compressing type information during load. Loading speed is also improved. Assumes that types with identical names have the same physical layout (even in different modules).
<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.SDS** *<filename>* [*<address>*] [**/***<option>*]
<option>:	**FPU** **NOCONST** **PACK** **FASTPACK** *<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.
FASTPACK	Same as above. Fastpack is faster and more efficient than **PACK**, but it requires unique type names in the whole applications. Fastpack assumes that types of identical name represent the same structure.
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.Srecord

Format:	**Data.LOAD.S1record** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.LOAD.S2record** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.LOAD.S3record** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.LOAD.S4record** *<filename>* [*<addressrange>*] [**/***<option>*]
<option>:	**FLAT** **OFFSET** *<offset>* **RECORDLEN** *<value>* *<generic_load_option>*

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.

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.Srecord

▲ 'Release Information' in 'Release History'

Format:	**Data.LOAD.sYm** *<filename>* [*<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>**.
<filename>	Specify the global SYM file as *<filename>*. 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** *<filename>* [*<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.Srecord

▲ 'Release Information' in 'Release History'

Data.LOAD.TEK Load TEKTRONIX file

Format:	**Data.LOAD.TEK** *<filename>* [*<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** *<filename>* [*<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** *<filename>* [*<address>*] [**/***<option>*]
<option>:	**ICC3S** | **ICC3L** **XSP** **NoMMU** **LARGE** **EXTPATH** *<.extension>* *<generic_load_option>*

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 'Release History'

Data.LOAD.VersaDos Load VERSADOS file

Format:	**Data.LOAD.VersaDos** *<filename>* [*<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** *<filename>* [*<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** *<dllfile> <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.

See also

- Data

Format:	**Data.Out** *<address>* [**%**[*<accessformat>*.]*<dataformat>*] *<data>* [**/***<option>*]
<access format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<dataformat>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** **BE** | **LE**
<option>:	**Repeat** | **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**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	See **"Keywords for <width>"**, page 10.
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 ■ Data.In ■ Data.Set ■ Data.Test
❏ ADDRESS.OFFSET() ❏ ADDRESS.SEGMENT() ❏ ADDRESS.STRACCESS() ❏ ADDRESS.WIDTH()

▲ 'Data Functions' in 'General Function Reference'
▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'

[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**

Fills the memory with a predefined pattern for the specified address range. The predefined pattern is as follows:

```
(8 byte hexadecimal address) 01 02 04 08 10 20 40 80
```

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**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	Specify memory access size. See **"Keywords for <width>"**, page 10. If no access size is specified, the debugger uses the optimum size for the processor architecture.

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.
ADDRESS.OFFSET()	Extracts the hex-address from the address object returned by TRACK.ADDRESS().

```
…
;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 " ADDRESS.OFFSET(TRACK.ADDRESS())
…
```

See also

■ Data   ■ Data.dump   ■ Data.Set   ■ Data.Test

▲ 'Release Information' in 'Release History'

Format:	**Data.Print** [[**%**<*format*>][<*address*> | <*range*>] …] [**/**<*option*> …]
<*format*>:	**Decimal** [**.**<*width*> [**.**<*endianness*> [**.**<*bitorder*>]]] **DecimalU** [**.**<*width*> [**.**<*endianness*> [**.**<*bitorder*>]]] **Hex** [**.**<*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*>]]]
<*width*>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<*endianness*>:	**DEFault** | **LE** | **BE**
<*bitorder*>:	**DEFault** | **BitSwap**
<*option*>:	**Mark** <*break*> **Track** **CORE** <*core*> **COVerage** **CTS** **FLAG** <*flag*> **CFlag** <*flag*> **CACHE**
<*flag*>:	**Read** | **Write** | **NoRead** | **NoWrite**
<*break*>:	**Program** | **HII** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

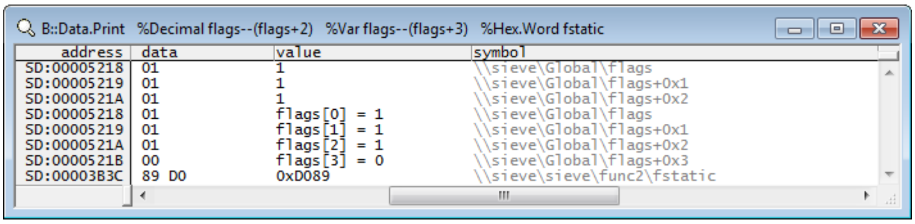
Displays the bare memory content on multiple address ranges as a list. If the single address format is selected, only one word at this address will be displayed. When selecting an address range the defined data range can be dumped.

Byte, **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	See **"Keywords for <width>"**, page 10.
BE, **LE**	Define byte-order display direction: Big Endian or Little Endian.

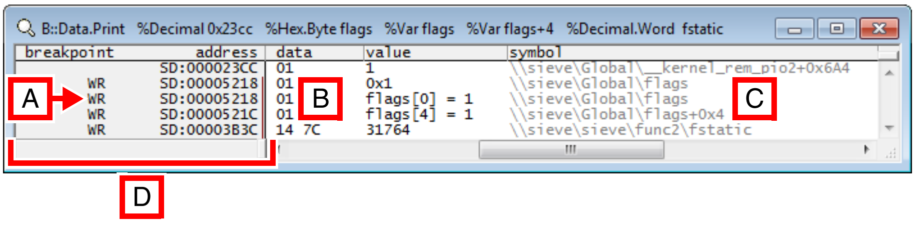
BitSwap	Display data with reverse bit-order in each byte. If **BitSwap** is used with **BE** or **LE**, the byte order will not changed, otherwise **BitSwap** will also reverse the byte-order.
DecimalU	Display the data as unsigned decimal number.
Float	Display data format in a floating point representation.
Mark	By using the **Mark** option individual bytes can be marked, depending on the breakpoint type set to the byte.
sYmbol	Display the data as hexadecimal address value. The column “symbol” in the windows will show the symbol corresponding to the address to which the *data points to* (while without **sYmbol** the column “symbol” show the symbol corresponding to the address *containing* the data).
Var	Display HLL variables at their memory location (similar to **Var.Watch**) The format can be configured with **SETUP.Var**

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 V.VALUE(dataptr)    ; display indexed by HLL pointer
```


B::Data.Print %Decimal flags--(flags+2) %Var flags--(flags+3) %Hex.Word fstatic

```
 address      data      value            symbol
SD:00005218  01        1                \\sieve\Global\flags
SD:00005219  01        1                \\sieve\Global\flags+0x1
SD:0000521A  01        1                \\sieve\Global\flags+0x2
SD:00005218  01        flags[0] = 1     \\sieve\Global\flags
SD:00005219  01        flags[1] = 1     \\sieve\Global\flags+0x1
SD:0000521A  01        flags[2] = 1     \\sieve\Global\flags+0x2
SD:0000521B  00        flags[3] = 0     \\sieve\Global\flags+0x3
SD:00003B3C  89 D0     0xD089           \\sieve\sieve\func2\fstatic
```


B::Data.Print %Decimal 0x23cc %Hex.Byte flags %Var flags %Var flags+4 %Decimal.Word fstatic
A
B
C
D

```
breakpoint      address    data    value          symbol
            SD:000023CC   01      1              \\sieve\Global\__kernel_rem_pio2+0x6A4
    WR      SD:00005218   01      0x1            \\sieve\Global\flags
    WR      SD:00005218   01      flags[0] = 1   \\sieve\Global\flags
    WR      SD:0000521C   01      flags[4] = 1   \\sieve\Global\flags+0x4
    WR      SD:00003B3C   14 7C   31764          \\sieve\sieve\func2\fstatic
```

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	■ Data.dump	■ Data.TABle	■ Data.View
❏ ADDRESS.OFFSET()	❏ ADDRESS.SEGMENT()	❏ ADDRESS.STRACCESS()	❏ ADDRESS.WIDTH()
❏ Data.Byte()	❏ Data.Float()	❏ Data.Long()	❏ Data.Quad()
❏ Data.STRing()	❏ Data.STRingN()	❏ Data.Word()	

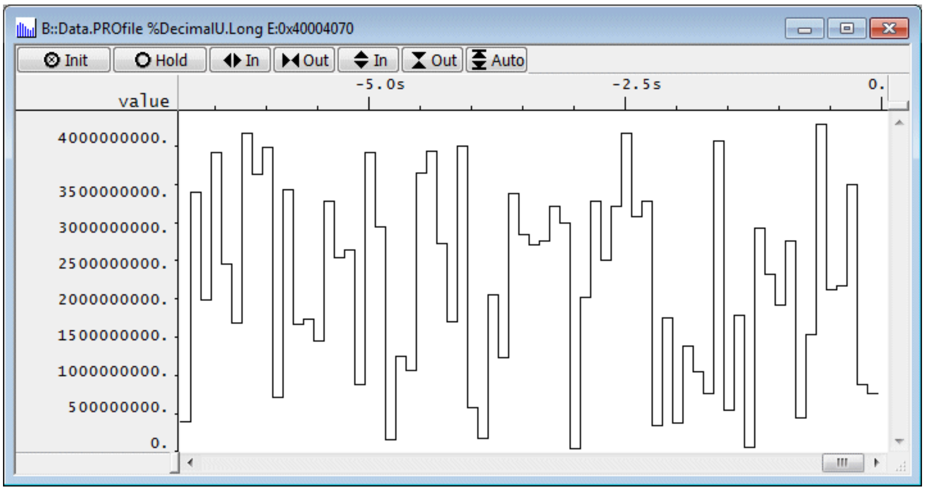
▲ 'ADDRESS Functions' in 'General Function Reference'
▲ 'Data Functions' in 'General Function Reference'
▲ 'Program and Data Memory' in 'ICE User's Guide'

Format:	**Data.PROfile** [%<*format*>][<*address*> …] [<*gate*>] [<*scale*>]
<*format*>:	**Decimal.**<*width*> **DecimalU.**<*width*> **Hex.**<*width*> **Float. Ieee** | **IeeeDbl** | **IeeeeXt** | <*others*> **Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<*width*>:	**DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte**
<*gate*>:	**0.1s** | **1.0s** | **10.0s**
<*scale*>:	**1.** … **32768.**

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.

Example 1:

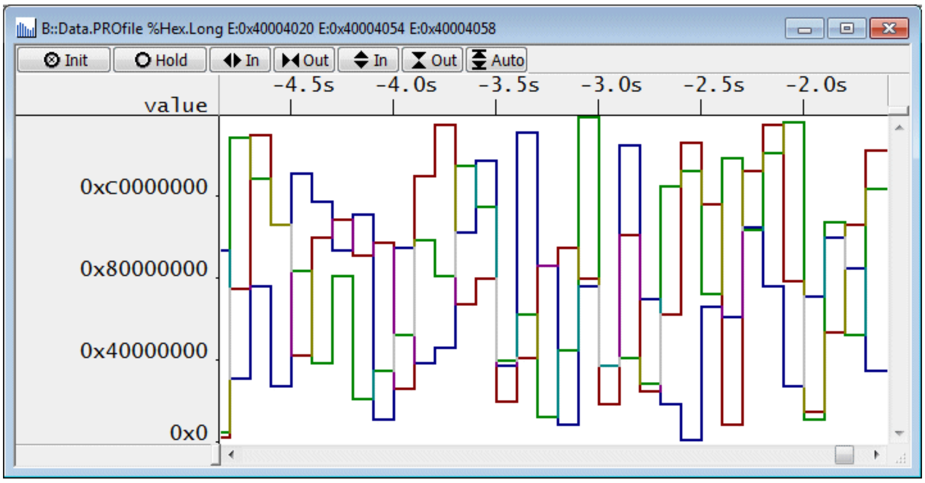
```
Data.PROfile %DecimalU.Long E:0x40004070
```



Buttons	
Init	Restart display
Hold	Stop update/re-start update
AutoZoom	Best vertical scaling

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 %Hex.Long E:0x40004020 E:0x40004054 E:0x40004058
```



See also

■ Data ■ Var.PROfile

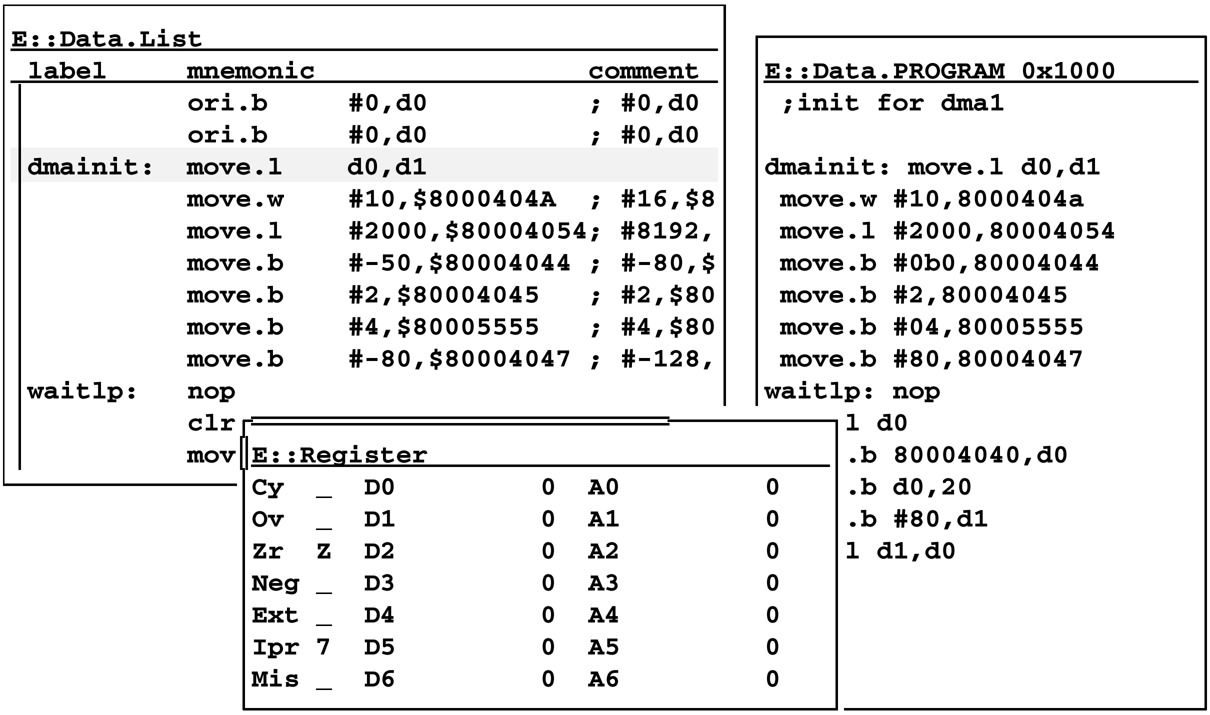
▲ 'Release Information' in 'Release History'

Data.PROGRAM

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.



```
E::Data.List
 label      mnemonic                          comment
            ori.b      #0,d0                  ; #0,d0
            ori.b      #0,d0                  ; #0,d0
 dmainit:   move.l     d0,d1
            move.w     #10,$8000404A          ; #16,$8
            move.l     #2000,$80004054;       #8192,
            move.b     #-50,$80004044         ; #-80,$
            move.b     #2,$80004045           ; #2,$80
            move.b     #4,$80005555           ; #4,$80
            move.b     #-80,$80004047         ; #-128,
 waitlp:    nop
            clr
            mov
```

```
E::Data.PROGRAM 0x1000
 ;init for dma1

dmainit: move.l d0,d1
 move.w #10,8000404a
 move.l #2000,80004054
 move.b #0b0,80004044
 move.b #2,80004045
 move.b #04,80005555
 move.b #80,80004047
waitlp: nop
    l d0
    .b 80004040,d0
    .b d0,20
    .b #80,d1
    l d1,d0
```

```
E::Register
Cy  _  D0          0  A0          0
Ov  _  D1          0  A1          0
Zr  Z  D2          0  A2          0
Neg _  D3          0  A3          0
Ext _  D4          0  A4          0
Ipr 7  D5          0  A5          0
Mis _  D6          0  A6          0
```

See also

■ Data ■ Data.Assemble ■ Data.Set ■ SETUP.EDITOR

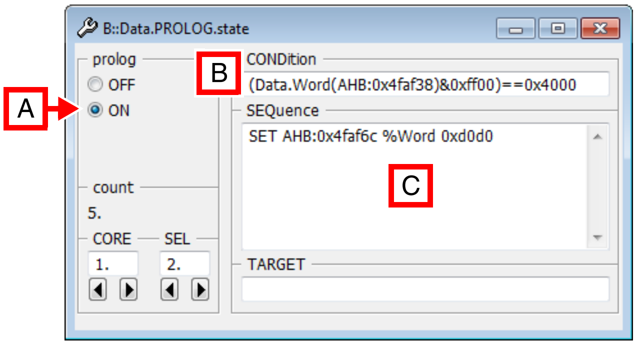
▲ 'Editing Files in the TRACE32 Editors - Syntax Highlighting' in 'PowerView User's Guide'
▲ 'Release Information' in 'Release History'

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:



prolog: ON
count: 5.
CORE: 1.
SEL: 2.
CONDition: (Data.Word(AHB:0x4faf38)&0xff00)==0x4000
SEQuence:
```
SET AHB:0x4faf6c %Word 0xd0d0
```

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** input field of the window using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

C Access sequences can be set up in the **SEQuence** input field of the window 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.PROLOG.CONDition	■ Data.PROLOG.CORE	■ Data.PROLOG.OFF	■ Data.PROLOG.ON
■ Data.PROLOG.RESet	■ Data.PROLOG.SELect	■ Data.PROLOG.SEQuence	■ Data.PROLOG.state
■ Data.PROLOG.TARGET	■ Data	■ Data.EPILOG	

▲ 'Release Information' in '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**.

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
```

NOTE: **Data.PROLOG.CONDition** only supports conditions as documented above. They may contain these memory access functions:

- **Data.Byte()** and its short form **D.B()**
- **Data.Long()** and its short form **D.L()**
- **Data.Word()** and its short form **D.W()**

See also

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

Data.PROLOG.CORE — Select core for data prolog

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 ■ 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 ■ 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
- 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
- Data.PROLOG.state

Data.PROLOG.SELect

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 (D.W(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 (D.W(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 (D.W(D:0x4faf38)&0xff00)==0x3000

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

See also

- Data.PROLOG
- 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> <inc>* At the first time performed, write *<start>* to *<address>*. *<start>* is incremented by *<inc>* 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:

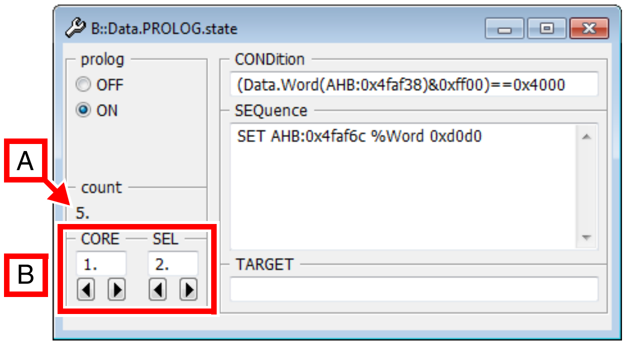
```
;Write 0xa0a0 when starting, increment by 2 for each successive start
Data.PROLOG.SEQuence SETI 0x3faf50 %Word 0xa0a0 2

;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

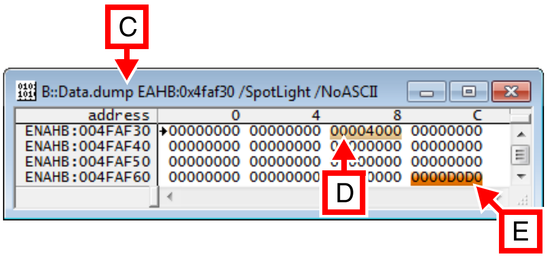
■ Data.PROLOG ■ Data.PROLOG.state

Format:	**Data.PROLOG.state**

Opens the **Data.PROLOG.state** window, where you can configure data prologs.



```
B::Data.PROLOG.state
prolog        CONDition
 ○ OFF        (Data.Word(AHB:0x4faf38)&0xff00)==0x4000
 ● ON         SEQuence
              SET AHB:0x4faf6c %Word 0xd0d0
count
5.
CORE  SEL     TARGET
1.    2.
```



```
B::Data.dump EAHB:0x4faf30 /SpotLight /NoASCII
       address          0        4        8        C
ENAHB:004FAF30 ▸00000000 00000000 00004000 00000000
ENAHB:004FAF40  00000000 00000000 0[illegible]000000 00000000
ENAHB:004FAF50  00000000 00000000 [illegible]00000 00000000
ENAHB:004FAF60  00000000 00000000 [illegible]0000 0000D0D0
```

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 to 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
- 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 '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
- Data.PROLOG.state

Data.REF

Format:	**Data.REF** [**E:**] [**/**<*option*> …]
<*option*>:	**Mark** <*break*> **Flag** <*flag*> **Track**
<*flag*>:	**Read** | **Write** | **NoRead** | **NoWrite**
<*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.

Mark	By using the Mark option individual bytes can be marked, depending on the breakpoint type set to the byte.
Track	Take the assembler instruction at the current track address.

See also

■ Data

Data.ReProgram

Assembler

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. The instructions for **Data.ReProgram** can also be embedded into a PRACTICE file.

See also

■ Data

Format:	**Data.ReRoute** *<range>* | *<module>* *<old_destination>* *<new_destination>* [\*<exclude_range>* | *<exclude_module>*]

Replaces, in *<range>* or *<module>,* all function calls with *<old_destination>* by *<new_destination>. <old_destination>* within *<exclude_range>* or *<exclude_module>* isn't 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 func5 by func7 within the code segment of
; the currently loaded program
sYmbol.List.SECtion
Data.ReRoute sYmbol.SECRANGE(.text) 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
```

See also

■ Data

Data.SAVE.<format> Save data in file with specified format

[Examples]

Format:	**Data.SAVE.***<format> <filename>* [*<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.

<filename>	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**.

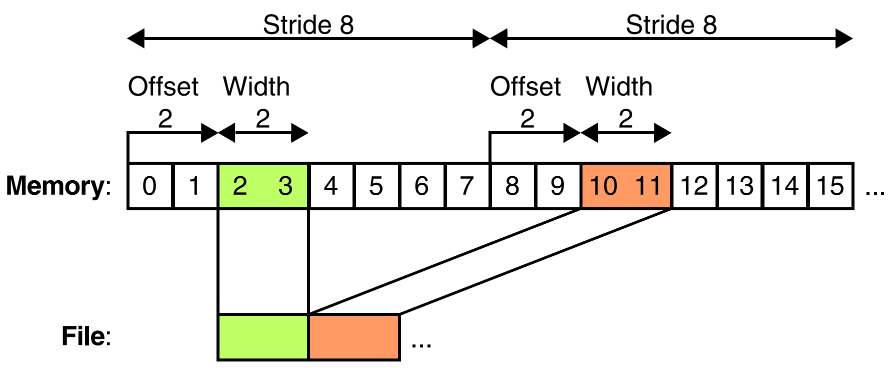
List of Generic Save Options

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.Srecord	■ Data		

▲ 'Release Information' in 'Release History'

Data.SAVE.Ascii — Save ASCII file

Format:	**Data.SAVE.Ascii** *<filename>* [*<addressrange>*] [**/***<option>*]
<option>:	**Hex** | **Decimal** | **DecimalU** | **BINary** | **Float.** [**Ieee** | **IeeeDbl** | **IeeeXt** | *<others>*] | **Append**

Saves an *<addressrange>* as a pure data file in word-oriented ASCII file format.

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** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AsciiHexP** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AsciiHexA** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AsciiHexS** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AsciiHexC** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AsciiHexB** *<filename>* [*<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>

Data.SAVE.AsciiOct — Save octal file

Format:	**Data.SAVE.AsciiOct** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AsciiOctP** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AsciiOctA** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AsciiOctS** *<filename>* [*<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>**.

See also

■ Data.SAVE.<format>

▲ 'Data Access' in 'EPROM/FLASH Simulator'

Data.SAVE.BDX — Save BDX file

Format:	**Data.SAVE.BDX** *<filename> <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** *<filename>* [*<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

▲ 'Program and Data Memory' in 'ICE User's Guide'

Data.SAVE.CCSDAT — Save CCSDAT file

Format:	**Data.SAVE.CCSDAT** *<filename> <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 'Release History'

Data.SAVE.DAB — Save DAB file

Format:	**Data.SAVE.DAB** *<filename> <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 'Release History'

Data.SAVE.Elf — Save ELF file

Format:	**Data.SAVE.Elf** *<filename> <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** *<filename> <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>

Data.SAVE.IntelHex — Save INTEL-HEX file

Format:	**Data.SAVE.IntelHex** *<filename> <addressrange>* [**/***<option>*]
<option>:	**ADDR** *<address_size>* **TYPE2** **TYPE4** **OFFSET** *<offset>* *<generic_save_options>*

Saves a file in an IntelHex format.

ADDR	Defines the *<address_size>* of the INTEL-HEX 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** *<filename> <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

Data.SAVE.PureHex

Save pure HEX file

Format:	**Data.SAVE.PureHEX** *<filename> <addressrange>* [**/***<option>*]
<option>:	*<generic_save_options>*

Saves memory in pure HEX file format.

<option>	For a description of the generic options, see **<generic_save_options>**.

See also

■ Data.SAVE.<format>

▲ 'Release Information' in 'Release History'

Data.SAVE.Srecord   Save S-record file

[Examples]

Format:	**Data.SAVE.S1record** *<filename> <range>*[||*<ranges>* …] [**/***<option>*] **Data.SAVE.S2record** *<filename> <range>*[||*<ranges>* …] [**/***<option>*] **Data.SAVE.S3record** *<filename> <range>*[||*<ranges>* …] [**/***<option>*] **Data.SAVE.S4record** *<filename> <range>*[||*<ranges>* …] [**/***<option>*]
<option>:	**OFFSET** | **RECORDLEN** | **Append** | **SkipErrors** *<generic_save_options>*

Saves memory content to a Srecord (special MOTOROLA file format).

S1record	The address field is interpreted as a 2-byte address. The data field is composed of memory loadable data.
S2record	The address field is interpreted as a 3-byte address. The data field is composed of memory loadable data.
S3record	The address field is interpreted as a 4-byte address. The data field is composed of memory loadable data.

The following options are provided:

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.<format> ■ Data.LOAD.SDS ■ Data.LOAD.Srecord ■ Data.LOAD.SysRof

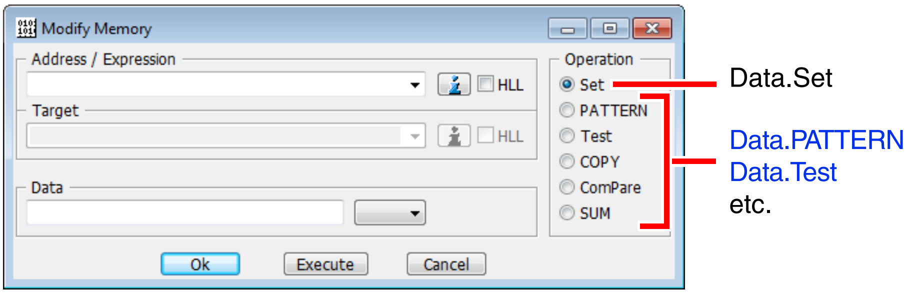
▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'

[Examples]

Format:	**Data.Set** [*<addressrange>*] {[**%**[*<accessformat>*.]*<dataformat>*] *<data>*} [{**/***<option>*}]
<access format>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<dataformat>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte** **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | *<others>*] **BE** | **LE** **BitSwap**
<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**	Define byte-order display direction: Big Endian or Little Endian.
BitSwap	Reverse 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 10.
ComPare	Compare the data against memory, don't write to memory.

CORE *<core_number>*	Perform 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	Verify the data by a following read operation.

Example 1

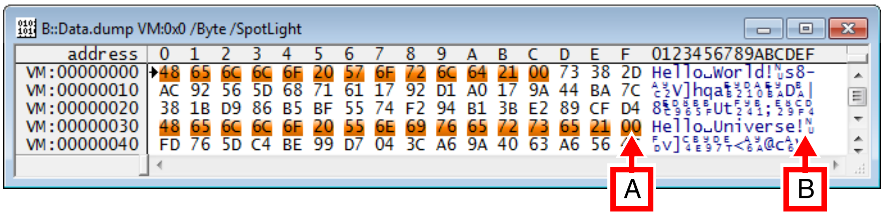
Various **Data.Set** operations with and without the use of PRACTICE functions:

```
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.IeeDbl 1.25         ; set floating point in
                                          ; IEEE-Double format

PRINT D.S(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
```



```
     address  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
VM:00000000 ▸48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00 73 38 2D  Hello World!.s8-
VM:00000010  AC 92 56 5D 68 71 61 17 92 D1 A0 17 9A 44 BA 7C  ..V]hqa......D.|
VM:00000020  38 1B D9 86 B5 BF 55 74 F2 94 B1 3B E2 89 CF D4  8.....Ut...;....
VM:00000030  48 65 6C 6C 6F 20 55 6E 69 76 65 72 73 65 21 00  Hello Universe!.
VM:00000040  FD 76 5D C4 BE 99 D7 04 3C A6 9A 40 63 A6 56     .v]....<..@c..
```

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.
ADDRESS.OFFSET()	Extracts the hex-address from the address object returned by TRACK.ADDRESS().

```
…
Data.Set 0x0++0xffff 123
Data.Set 0x0++0xffff 123 /DIFF

IF FOUND()
    PRINT "Error found at address " ADDRESS.OFFSET(TRACK.ADDRESS())

…
```

See also

■ Data ■ Data.Out ■ Data.PATTERN ■ Data.PROGRAM
■ Data.Test

▲ 'Data Access' in 'EPROM/FLASH Simulator'
▲ 'Appendix A' in 'Tips to Solve NOR FLASH Programming Problems'
▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'
▲ 'Registers' in 'Debugger Basics - Training'
▲ 'Registers' in 'Debugger Basics - SMP Training'
▲ 'Registers' in 'Training FIRE Basics'
▲ 'Registers' in 'Training ICE Basics'

Data.SOFTEPILOG Automated sequence after setting software breakp.

tbd.

See also

■ Data

▲ 'Release Information' in 'Release History'

Data.SOFTPROLOG Automated sequence before setting software breakp.

tbd.

See also

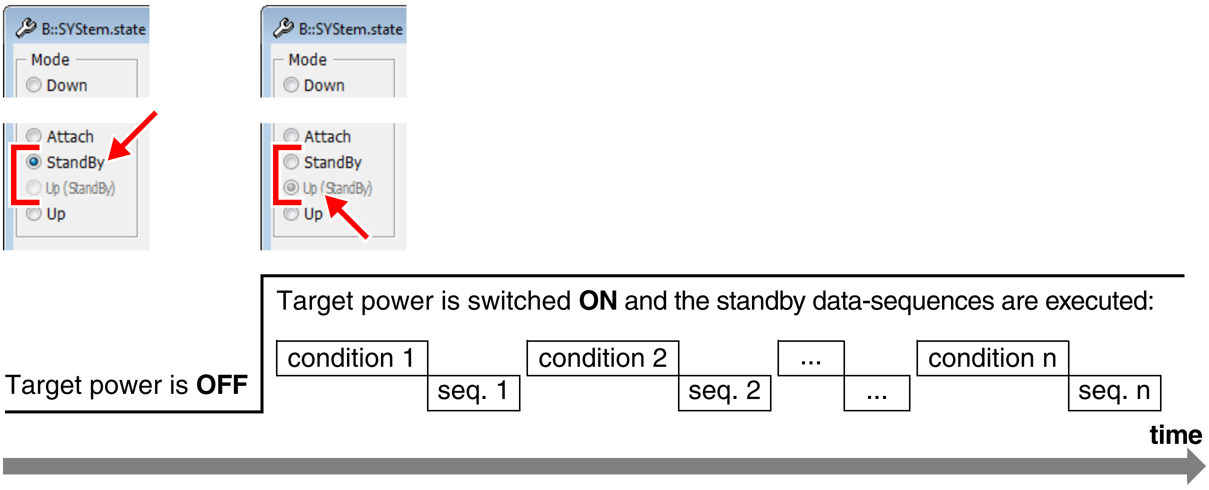
■ Data

▲ 'Release Information' in 'Release History'

Only for PowerPC MPC5xxx, TriCore

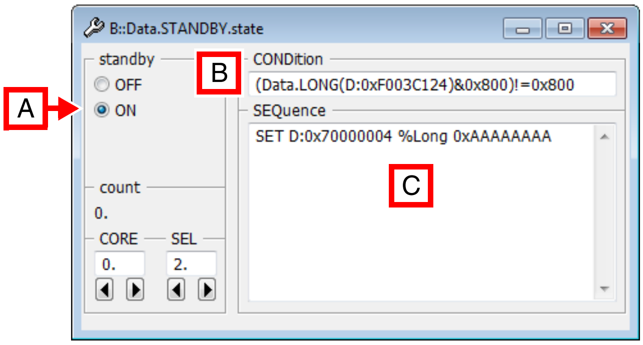
Using the **Data.STANDBY** command group, you can define one or more sequences that perform write operations to registers or memory. For example, you can use the **Data.STANDBY** command group to deactivate a watchdog.

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:



standby: OFF [✓ ON]

count: 0.

CORE: 0. SEL: 2.

CONDition: (Data.LONG(D:0xF003C124)&0x800)!=0x800

SEQuence:

```
SET D:0x70000004 %Long 0xAAAAAAAA
```

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 ■ Data

▲ 'Release Information' in 'Release History'

Data.STANDBY.CONDition

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.STARTUP.RESet**

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

See also

- Data.STANDBY
- Data.STANDBY.state

Data.STANDBY.SELect   Increment index number for next sequence

Only for PowerPC MPC5xxx, TriCore

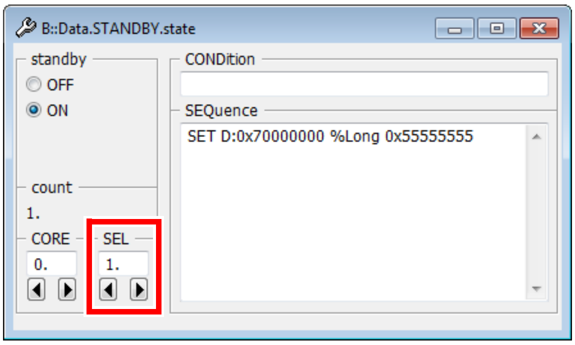
Format: **Data.STANDBY.SELect** *<index_number>*

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
```



B::Data.STANDBY.state

standby: ○ OFF [✓ ON]

CONDition:

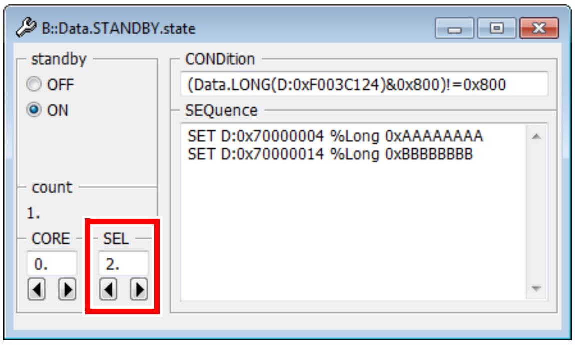
SEQuence:

```
SET D:0x70000000 %Long 0x55555555
```

count: 1.

CORE: 0.

SEL: 1.



B::Data.STANDBY.state

standby: ○ OFF [✓ ON]

CONDition: (Data.LONG(D:0xF003C124)&0x800)!=0x800

SEQuence:

```
SET D:0x70000004 %Long 0xAAAAAAAA
SET D:0x70000014 %Long 0xBBBBBBBB
```

count: 1.

CORE: 0.

SEL: 2.

See also

- Data.STANDBY
- Data.STANDBY.state

Data.STANDBY.SEQuence

Define sequence

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

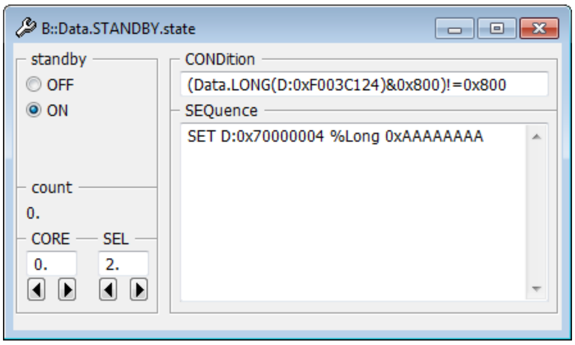
■ Data.STANDBY ■ Data.STANDBY.state

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.


B::Data.STANDBY.state

standby
OFF
[✓ ON]

count
0.

CORE
0.

SEL
2.

CONDition
(Data.LONG(D:0xF003C124)&0x800)!=0x800

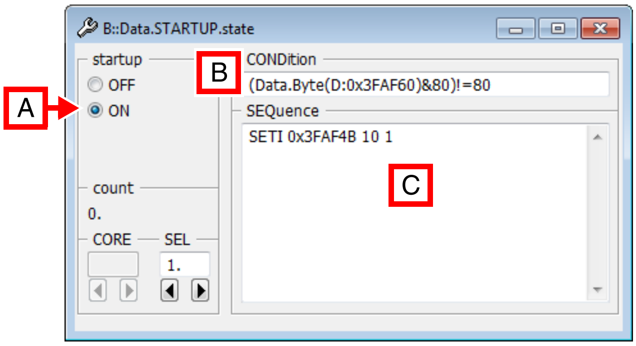
SEQuence
```
SET D:0x70000004 %Long 0xAAAAAAAA
```

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 or emulator is activated with **SYStem.Up**.

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



startup: ON
count: 0.
SEL: 1.
CONDition: (Data.Byte(D:0x3FAF60)&80)!=80
SEQuence: SETI 0x3FAF4B 10 1

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** input field of the window using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

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

See also

- Data.STARTUP.CONDition
- Data.STARTUP.OFF
- Data.STARTUP.ON
- Data.STARTUP.RESet
- Data.STARTUP.SEQuence
- Data.STARTUP.state
- Data.STANDBY
- Data

Data.STARTUP.CONDition — Define startup condition

Format: **Data.STARTUP.CONDition** *<condition>*

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

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 (D.L(D:0x3faf30)&0xffffffff)==0x80000000
```

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

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

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

Data.STARTUP.SEQuence — Define startup data sequence

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>* when emulation is activated.
SETI	Write *<data>* to *<address>*. Then *<data>* is incremented by *<increment>*.
SETS	Write the data that was saved with a previous GETS back to *<address>*.
GETS	Save the data at *<address>*.

Examples:

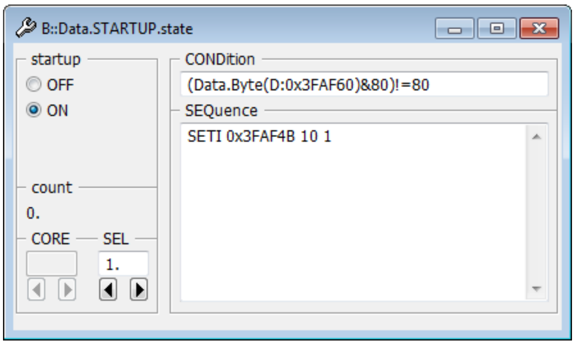
```
Data.STARTUP.SEQuence SET 0x3faf50 %Word 0xa0a0

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

Data.STARTUP.state Startup data state display

Format:	**Data.STARTUP.state**


B::Data.STARTUP.state

startup
OFF [✓ ON]

CONDition
(Data.Byte(D:0x3FAF60)&80)!=80

SEQuence
SETI 0x3FAF4B 10 1

count
0.

CORE SEL
1.

See also

■ Data.STARTUP ■ Data.STARTUP.CONDition ■ Data.STARTUP.OFF ■ Data.STARTUP.ON
■ Data.STARTUP.RESet ■ Data.STARTUP.SEQuence

▲ 'Release Information' in 'Release History'

Data.STRING ASCII display

Format: **Data.STRING** [**%CONTinue**] *<addressrange>* [**/***<option>*]

<option>: **PC8**

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.
PC8	The option **PC8** converts graphic characters from IBM-PC character set to a host independent graphic character set. Linefeed characters are interpreted.

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 /PC8      ; display data in the window
```

See also

■ Data ■ Data.dump ■ Data.WRITESTRING ❏ Data.STRing()
▲ 'Release Information' in '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 ■ Data.Test ❏ Data.SUM()

▲ 'Program and Data Memory' in 'ICE User's Guide'

Format:	**Data.TABle** *<base>*|*<range>* *<size>* *<element>* [*<element>***...**] [**/***<option>* …]
<element>:	[**%***<format>*] [*<offset>*|*<offrange>*]
<format>:	**Decimal.**[*<width>*] **DecimalU.**[*<width>*] **Hex.**[*<width>*] **Ascii.**[*<width>*] **Binary.**[*<width>*] **Float.**[**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeMFFP**] **DUMP.***<width>* **Var**
<width>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<offset>: *<offrange>*:	relative byte offset of the element
<option>:	**Mark** *<break>* **Flag** *<flag>* **Track**
<flag>:	**Read** | **Write** | **NoRead** | **NoWrite**
<break>:	**Program** | **HII** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly** | **Delta** | **Echo**

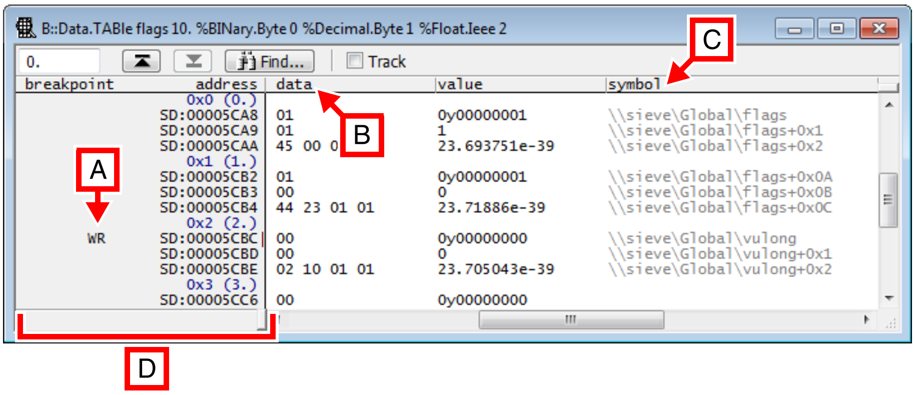
Displays an array without high-level information. If an address is given, it will specify the base address of an array of unlimited size. A range specifies an array of limited size.

<base>, <range>	Base address or address range of data structure
<size>	Size in bytes of a single element of the data structure
<format> **DecimalU** **Decimal** **Hex** **Binary** **Float**	Format of the element: • Unsigned Decimal • Signed Decimal • Hexadecimal value • Binary value • Floating point value
Byte, **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	Width of the element. See **“Keywords for <width>”**, page 10.

Ascii	ASCII value
Var	Display in HLL format.
Mark	By using the Mark option individual bytes can be marked, depending on the breakpoint type set to the byte.

```
; 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.


B::Data.TABle flags 10. %BINary.Byte 0 %Decimal.Byte 1 %Float.Ieee 2
0.
Find...
Track
A
B
C
D

```
breakpoint          address  data          value          symbol
                  0x0 (0.)
               SD:00005CA8   01            0y00000001     \\sieve\Global\flags
               SD:00005CA9   01            1              \\sieve\Global\flags+0x1
               SD:00005CAA   45  00  0     23.693751e-39  \\sieve\Global\flags+0x2
                  0x1 (1.)
               SD:00005CB2   01            0y00000001     \\sieve\Global\flags+0x0A
               SD:00005CB3   00            0              \\sieve\Global\flags+0x0B
               SD:00005CB4   44  23  01  01  23.71886e-39 \\sieve\Global\flags+0x0C
                  0x2 (2.)
   WR          SD:00005CBC   00            0y00000000     \\sieve\Global\vulong
               SD:00005CBD   00            0              \\sieve\Global\vulong+0x1
               SD:00005CBE   02  10  01  01  23.705043e-39 \\sieve\Global\vulong+0x2
                  0x3 (3.)
               SD:00005CC6   00            0y00000000
```

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 ■ Data.CHAIN ■ Data.dump ■ Data.Print
■ Data.View

▲ 'Program and Data Memory' in 'ICE User's Guide'

Data.TAG — Tag code for analysis

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

Data.TAGFunc — Tag code for analysis

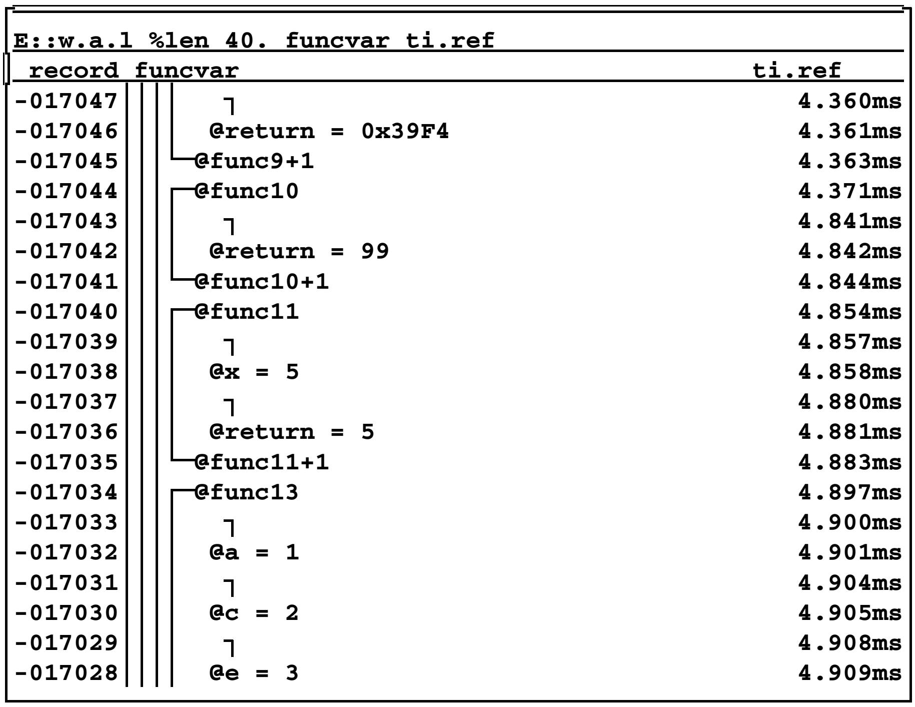
Format:	**Data.TAGFunc** [*<group>* | *<range>*] *<patch>* [*<tags>*] [**/***<options>*]
<tags>:	*<tags_entry>* [*<tags_exit>* [*<tags_parameter>*]]
<options>:	**INTR** **Parameter**

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)
```



```
E::w.a.l %len 40. funcvar ti.ref
 record funcvar                                  ti.ref
-017047|||  ┐                                   4.360ms
-017046|||  @return = 0x39F4                    4.361ms
-017045||└─@func9+1                             4.363ms
-017044||┌─@func10                              4.371ms
-017043|||  ┐                                   4.841ms
-017042|||  @return = 99                        4.842ms
-017041||└─@func10+1                            4.844ms
-017040||┌─@func11                              4.854ms
-017039|||  ┐                                   4.857ms
-017038|||  @x = 5                              4.858ms
-017037|||  ┐                                   4.880ms
-017036|||  @return = 5                         4.881ms
-017035||└─@func11+1                            4.883ms
-017034||┌─@func13                              4.897ms
-017033|||  ┐                                   4.900ms
-017032|||  @a = 1                              4.901ms
-017031|||  ┐                                   4.904ms
-017030|||  @c = 2                              4.905ms
-017029|||  ┐                                   4.908ms
-017028|||  @e = 3                              4.909ms
```

```
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
- Data.UNTAGFunc

[Examples]

Format:	**Data.Test** *<address_range>* [**/***<option>*]
<option>:	**Toggle** **Prime** **RANDOM** | **PRANDOM** **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**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	Specify memory access size. See **"Keywords for <width>"**, page 10. 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.
ADDRESS.OFFSET()	Extracts the hex-address from the address object returned by TRACK.ADDRESS().

```
…

Data.Test 0x0++0xffff /Prime

IF Found()
     PRINT "Error found at address " ADDRESS.OFFSET(TRACK.ADDRESS())

…
```

See also

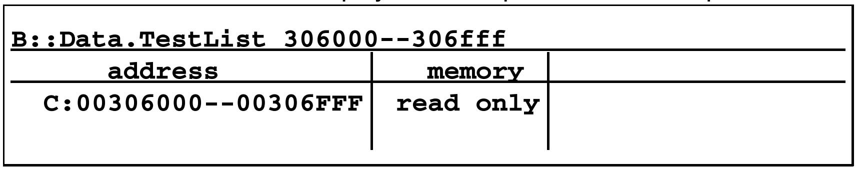
■ Data	■ Data.dump	■ Data.Out	■ Data.PATTERN
■ Data.Set	■ Data.SUM	■ SETUP.TIMEOUT	❏ FOUND()

▲ 'Program and Data Memory' in 'ICE User's Guide'
▲ 'Release Information' in '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>*.



```
B::Data.TestList 306000--306fff
         address           memory
  C:00306000--00306FFF  read only
```

The following results are possible:

ok	RAM
read only	ROM/FLASH
read fail **write fail**	no memory

The command **Data.TestList** may cause a “emulation debug port fail” if the SFR/peripherals are accessed.

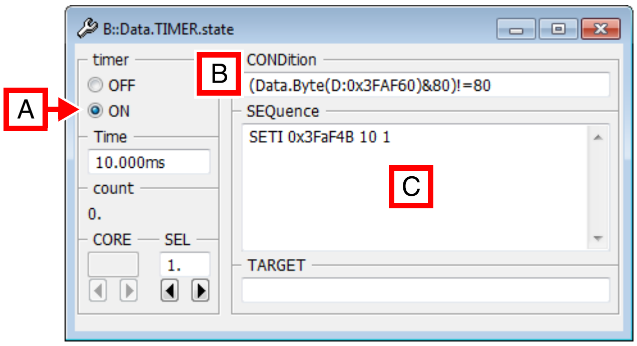
See also

■ Data

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:



timer: OFF [✓ ON]
Time: 10.000ms
count: 0.
SEL: 1.
CONDition: (Data.Byte(D:0x3FAF60)&80)!=80
SEQuence: SETI 0x3FaF4B 10 1

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** input field of the window using the functions **Data.Byte()**, **Data.Long()**, or **Data.Word()**.

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

See also

- Data.TIMER.CONDition
- Data.TIMER.OFF
- Data.TIMER.ON
- Data.TIMER.RESet
- Data.TIMER.SEQuence
- Data.TIMER.state
- Data.TIMER.TARGET
- Data.TIMER.Time
- Data

Data.TIMER.CONDition   Define timer condition

Format: **Data.TIMER.CONDition** *<condition>*

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

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

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

Data.TIMER.RESet Reset timer

Format:	**Data.TIMER.RESet**

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

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:

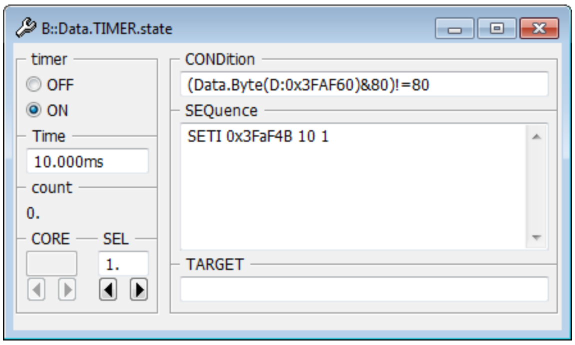
```
Data.TIMER.SEQuence SET  0x3fa50 %Long 0x11223344

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**



Time: 10.000ms
count: 0.
SEL: 1.
CONDition: (Data.Byte(D:0x3FAF60)&80)!=80
SEQuence: SETI 0x3FaF4B 10 1

See also

- ■ Data.TIMER ■ Data.TIMER.CONDition ■ Data.TIMER.OFF ■ Data.TIMER.ON
- ■ Data.TIMER.RESet ■ Data.TIMER.SEQuence ■ Data.TIMER.TARGET ■ Data.TIMER.Time
- ▲ 'Release Information' in 'Release History'

Data.TIMER.TARGET   Define timer target call

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

Format: **Data.TIMER.Time** *<time>*

Defines the period for the **Data.TIMER** feature.

```
Data.TIMER.Time 10.ms
```

See also

- Data.TIMER
- Data.TIMER.state

Data.UNTAGFunc — Remove code tags

Format:	**Data.UNTAGFunc**

Removes the tags generated by the **Data.TAGFunc** command.

See also

- Data
- 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.

See also

- Data

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 ■ Data.dump

▲ 'Release Information' in 'Release History'

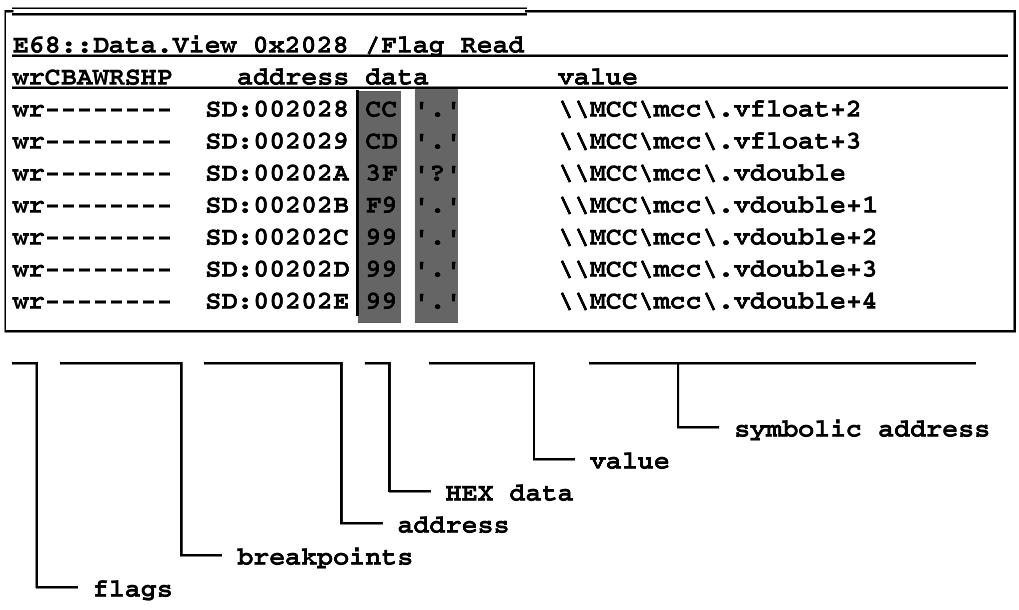
Format:	**Data.View** [**%**<*format*>] [<*address*> | <*range*>] [**/**<*option*> …]
<*format*>:	**Decimal** [**.**<*width*> [**.**<*endianness*> [**.**<*bitorder*>]]] **DecimalU** [**.**<*width*> [**.**<*endianness*> [**.**<*bitorder*>]]] **Hex** [**.**<*width*> [**.**<*endianness*> [**.**<*bitorder*>]]] **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*>]]]
<*width*>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **PByte** | **HByte** | **SByte**
<*endianness*>:	**DEFault** | **LE** | **BE**
<*bitorder*>:	**DEFault** | **BitSwap**
<*option*>:	**Mark** <*break*> **Track** **CORE** <*core*> **COVerage** **CTS** **CFlag** <*flag*> **FLAG** <*flag*> **CACHE**
<*flag*>:	**Read** | **Write** | **NoRead** | **NoWrite**
<*break*>:	**Program** | **HlI** | **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).

DecimalU	Display the data as unsigned decimal number.
sYmbol	Display the data as hexadecimal address value. The column “symbol” in the windows will show the symbol corresponding to the address to which the *data points to* (while without **sYmbol** the column “symbol” show the symbol corresponding to the address *containing* the data).
Var	Display HLL variables at their memory location (similar to **Var.Watch**). The format can be configured with **SETUP.Var**.
Byte, **Word**, **TByte**, **Long**, **PByte**, **HByte**, **SByte**, **Quad**	See **“Keywords for <width>”**, page 10.
Float	Display data format in a floating point representation.
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.
CACHE	Displays cache hit information and marks currently cached code.
Mark	By using the **Mark** option individual bytes can be marked, depending on the breakpoint type set to the byte.



```
E68::Data.View 0x2028 /Flag Read
wrCBAWRSHP    address  data        value
wr--------   SD:002028 CC  '.'     \\MCC\mcc\.vfloat+2
wr--------   SD:002029 CD  '.'     \\MCC\mcc\.vfloat+3
wr--------   SD:00202A 3F  '?'     \\MCC\mcc\.vdouble
wr--------   SD:00202B F9  '.'     \\MCC\mcc\.vdouble+1
wr--------   SD:00202C 99  '.'     \\MCC\mcc\.vdouble+2
wr--------   SD:00202D 99  '.'     \\MCC\mcc\.vdouble+3
wr--------   SD:00202E 99  '.'     \\MCC\mcc\.vdouble+4
```

The scale area contains addresses and memory classes, as well as Flag and Breakpoint information. 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	■ 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.Word()		

▲ ’Data Functions’ in ’General Function Reference’
▲ ’ADDRESS Functions’ in ’General Function Reference’
▲ ’Program and Data Memory’ in ’ICE User’s Guide’
▲ ’Release Information’ in ’Release History’
▲ ’Data and Program Memory’ in ’Training FIRE Basics’
▲ ’Data and Program Memory’ in ’Training ICE Basics’

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).

```
OPEN #1 testfile /Create
Data.WRITESTRING #1 100--1ff
CLOSE #1
```

See also

■ Data	■ Data.STRING	■ CLOSE	■ OPEN
❏ Data.WSTRING()			

▲ ’Release Information’ in ’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.

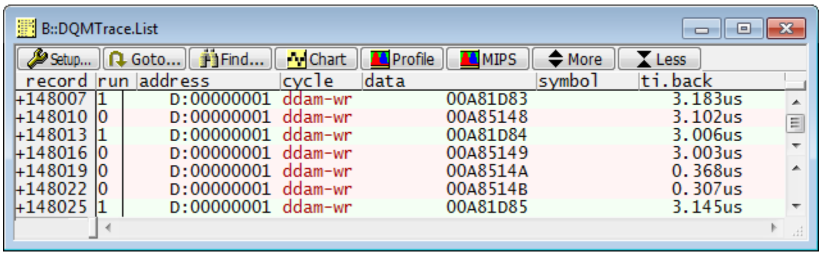
The **DCI** command group allows expert control of this hardware module. If the Intel® Trace Hub commands are used, then this configuration is done automatically (see **ITH** commands in **“Intel® Trace Hub”** (trace_intel_th.pdf).

For more information about the direct connect interface (DCI), see **“Debugging via Intel® DCI User´s Guide”** (dci_intel_user.pdf).

See also

- SYStem

The **DQMTrace** command group allows to display and analyze trace information exported by Data Acquisition Messaging of Nexus PowerArchitecture.



```
 record |run |address          |cycle   |data              |symbol    |ti.back
+148007 |1  |     D:00000001  ddam-wr          00A81D83                  3.183us
+148010 |0  |     D:00000001  ddam-wr          00A85148                  3.102us
+148013 |1  |     D:00000001  ddam-wr          00A81D84                  3.006us
+148016 |0  |     D:00000001  ddam-wr          00A85149                  3.003us
+148019 |0  |     D:00000001  ddam-wr          00A8514A                  0.368us
+148022 |0  |     D:00000001  ddam-wr          00A8514B                  0.307us
+148025 |1  |     D:00000001  ddam-wr          00A81D85                  3.145us
```

```
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 — 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 traced.

DTU

DTU — Debug Trace Unit (DTU)

VSPA only

DTU is the TRACE32 name for the trace cell of the VSPA architecture. For more information refer to **“VSPA Debugger”** (debugger_vspa.pdf).