General Commands Reference Guide D

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ....

General Commands and Functions ....

General Commands Reference Guide D 1

Data 6

Data Memory of TRACE32-ICE, TRACE32-FIRE		6
Access Procedures		6
Memory Classes		7
Data Memory of TRACE32-ICD		8
Access Procedures		8
Memory Classes		8
Functions		9
Data.AllocList	Static memory allocation analysis	10
Data.Assemble	Inline assembler	14
Data.BDTAB	Display buffer descriptor table	14
Data.BENCHMARK	Determine cache/memory bandwidth	15
Data.CHAIN	Display linked list	19
Data.CHAINFind	Search in linked list	22
Data.ComPare	Compare memory	23
Data.COPY	Copy memory	25
Data.DRAW	Graphical memory display of y-arrays	26
Data.DRAWFFT	Graphical memory display for fast fourier transform	31
Data.DRAWXY	Graphical display of xy-graphs	35
Data.dump	Memory dump	39
Data.EPILOG	Sequence after stop	44
Data.EPILOG.CONDition	Define EPILOG condition	44
Data.EPILOG.OFF	Switch EPILOG off	45
Data.EPILOG.ON	Switch EPILOG on	45
Data.EPILOG.RESet	Reset EPILOG	45
Data.EPILOG.SEQuence	Define EPILOG sequence	46
Data.EPILOG.state	EPILOG state display	47
Data.EPILOG.TARGET	Define EPILOG target call	47
Data.Find	Search in memory	48
Data.GOTO	Track to address	51
Data.GREP	Search for string	52
Data.IMAGE	Display image data	53

Data.In Read port 56
Data.List Display source listing 57
Data.ListAsm Display disassembler 62
Data.ListHll Display source 63
Data.ListJava Display JAVA byte code 64
Data.ListMix Disassembler and source 65
Data.LOAD Load file 66
Program Loading 71
Format Specific Data.LOAD commands and options 73
Data.LOAD.AIF Load ARM image file 73
Data.LOAD.AOUT Load a.out file 74
Data.LOAD.ASAP2 Load ASAP2 file 74
Data.LOAD.AsciiHex Load hex file 75
Data.LOAD.AsciiOct Load octal file 75
Data.LOAD.AVocet Load AVOCET file 76
Data.LOAD.Binary Load binary file 76
Data.LOAD.BounD Load BOUND file 77
Data.LOAD.CDB Load SDCC CDB file format 78
Data.LOAD.COFF Load COFF file 79
Data.LOAD.COMFOR Load COMFOR (TEKTRONIX) file 80
Data.LOAD.CORE Load linux core dump file 80
Data.LOAD.COSMIC Load COSMIC file 81
Data.LOAD.DBX Load a.out file 82
Data.LOAD.Elf Load ELF file 83
Data.LOAD.eXe Load EXE file 87
Data.LOAD.HiCross Load HICROSS file 88
Data.LOAD.HiTech Load HITECH file 88
Data.LOAD.HP Load HP-64000 file 89
Data.LOAD.ICoff Load ICOFF file 90
Data.LOAD.Ieee Load IEEE-695 file 91
Data.LOAD.IntelHex Load INTEL-HEX file 92
Data.LOAD.Jedec Load JEDEC file 93
Data.LOAD.MachO Load 'Mach-O' file 94
Data.LOAD.MAP Load MAP file 96
Data.LOAD.MCDS Load MCDS file 96
Data.LOAD.MCoff Load MCOFF file 96
Data.LOAD.Omf Load OMF file 97
Special Requirements for Intel PL/M-86 and PL/M-51 100
Data.LOAD.REAL Load R.E.A.L. file 100
Data.LOAD.ROF Load OS-9 file 101
Limitations 101
Data.LOAD.SDS Load SDSI file 102
Data.LOAD.Srecord Load S-Record file 103

Data.LOAD.sYm Load symbol file 104
Data.LOAD.SysRof Load RENESAS SYSROF file 105
Data.LOAD.TEK Load TEKTRONIX file 105
Data.LOAD.TekHex Load TEKTRONIX HEX file 105
Data.LOAD.Ubrof Load UBROF file 106
Data.LOAD.VersaDos Load VERSADOS file 107
Data.LOAD.XCoff Load XCOFF file 107
Data.LOG Log data accesses done by TRACE32 108
Data.MSYS M-SYSTEMS FLASHDISK support 108
Data.Out Write port 109
Data.PATTERN Fill memory with pattern 110
Data.Print Display multiple areas 111
Data.PROfile Graphic variable display 113
Data.PROGRAM Assembler window 115
Data.PROLOG Sequence before start 116
Data.PROLOG.CONDition Define PROLOG condition 116
Data.PROLOG.OFF Switch PROLOG off 117
Data.PROLOG.ON Switch PROLOG on 117
Data.PROLOG.RESet Reset PROLOG 117
Data.PROLOG.SEQuence Define PROLOG sequence 118
Data.PROLOG.state PROLOG state display 119
Data.PROLOG.TARGET Define PROLOG target call 119
Data.REF Display current values 120
Data.ReProgram Assembler 120
Data.ReRoute Reroute function call 121
Data.SAVE.<format> Save data in file with specified forma 122
Data.SAVE.ASAP2 Save octal file 122
Data.SAVE.AsciiHex Save hex file 123
Data.SAVE.AsciiOct Save octal file 124
Data.SAVE.Binary Save binary file 125
Data.SAVE.IntelHex Save INTEL-HEX file 125
Data.SAVE.Omf Save OMF file 126
Data.SAVE.Srecord Save S-Record file 126
Data.Set Modify memory 127
Data.STARTUP Startup data sequence 128
Data.STARTUP.CONDition Define startup condition 128
Data.STARTUP.OFF Switch startup sequence off 128
Data.STARTUP.ON Switch startup data sequence on 129
Data.STARTUP.RESet Reset startup data sequence 129
Data.STARTUP.SEQuence Define startup data sequence 130
Data.STARTUP.state Startup data state display 130
Data.STRING ASCII display 131
Data.SUM Memory checksum 131

Data.TABle Display arrays 134
Data.TAG Tag code for analysis 135
Data.TAGFunc Tag code for analysis 136
Data.Test Memory test 138
Data.TestList Test for memory type 140
Data.TIMER Periodically data sequence 140
Data.TIMER.CONDition Define timer condition 141
Data.TIMER.OFF Switch timer off 141
Data.TIMER.ON Switch timer on 142
Data.TIMER.RESet Reset timer 142
Data.TIMER.SEQuence Define timer sequence 143
Data.TIMER.state Timer state display 144
Data.TIMER.TARGET Define timer target call 144
Data.TIMER.Time Define period for timer 145
Data.UNTAGFunc Remove code tags 145
Data.UPDATE Target memory cache update 145
Data.USRACCESS Access to special target memory 146
Data.View Display memory 147

DTM (Data Trace Module) **149**

Usage:

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

Version October 24, 2011

07/19/11	Revision of **Data.DRAW**, **Data.DRAWFFT**, and **Data.DRAWXY** plus examples and screenshots.
08/10/09	Description for Data.LOAD options CHECKONLY, CHECKLOAD, ZIPLOAD added.

Data Memory of 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

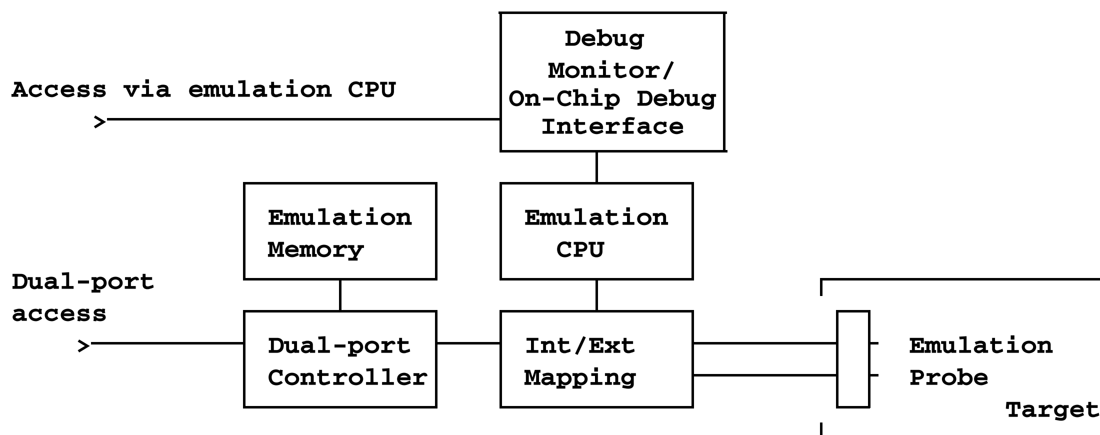
```
Data.dump UD:0x1000
```

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 realtime 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 **ICE Target Manual** and **FIRE Target Manual**.

Data Memory of TRACE32-ICD

TRACE32-ICD (On-chip debug interfaces as well as ROM monitors) use the memory on the target system. An EPROM-simulator can be used to load and debug programs in the ROM area.

Access Procedures

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

```
Data.dump D:0x1000
```

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

Some on-chip debug interfaces (e.g. 68HC12, C166CBC) allow memory read and write while the CPU is executing the program. The command

```
Data.dump ED:0x1000
```

will be used to represent the same address space, except that it will be accessed while the CPU is running. For more information refer to the **ICD Target Manual**.

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
A	Absolute (Physical addressing)

Other classes are described within the **ICD Target Manual**.

Data.Byte(<*address*>)	
Data.Word(<*address*>)	
Data.Long(<*address*>)	Read memory at specified address. Address must be used with access class.
Data.String(<*address*>)	Read zero-terminated string from memory, the result is a string.
DATA.SUM()	Get the checksum of the last executed **D.SUM** command.

```
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
```

See also

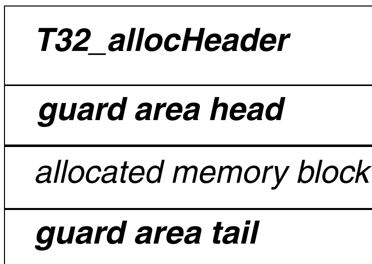
- ❏ ADDRESS.ACCESS()
- ❏ ADDRESS.DATA()
- ❏ ADDRESS.OFFSET()
- ❏ ADDRESS.ONCHIP()
- ❏ ADDRESS.PROGRAM()
- ❏ ADDRESS.SEGMENT()
- ❏ ADDRESS.SPACE()
- ❏ ADDRESS.WIDTH()
- ❏ Data.AL.ERRORS()
- ❏ Data.Byte()
- ❏ Data.Float()
- ❏ Data.Long()
- ❏ Data.Long.BigEndian()
- ❏ Data.Long.LittleEndian()
- ❏ Data.LongLong()
- ❏ Data.LongLong.BigEndian()
- ❏ Data.LongLong.LittleEndian()
- ❏ Data.Quad()
- ❏ Data.Quad.BigEndian()
- ❏ Data.Quad.LittleEndian()
- ❏ Data.Short()
- ❏ Data.Short.BigEndian()
- ❏ Data.Short.LittleEndian()
- ❏ Data.SLong()
- ❏ Data.STRing()
- ❏ Data.SUM()
- ❏ Data.TByte()
- ❏ Data.Word()
- ❏ Data.Word.BigEndian()
- ❏ Data.Word.LittleEndian()
- ❏ Data.WSTRING()

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 analyse this information.

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



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

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

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

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

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

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

In order to maintain the double linked list that is required by the TRACE32 software to analyse 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 2 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 Y.SECRANGE(.text) malloc T32_malloc \t32mem
Data.ReRoute Y.SECRANGE(.text) realloc T32_realloc \t32mem
Data.ReRoute Y.SECRANGE(.text) free T32_free \t32mem
```

What does T32_malloc(size)?

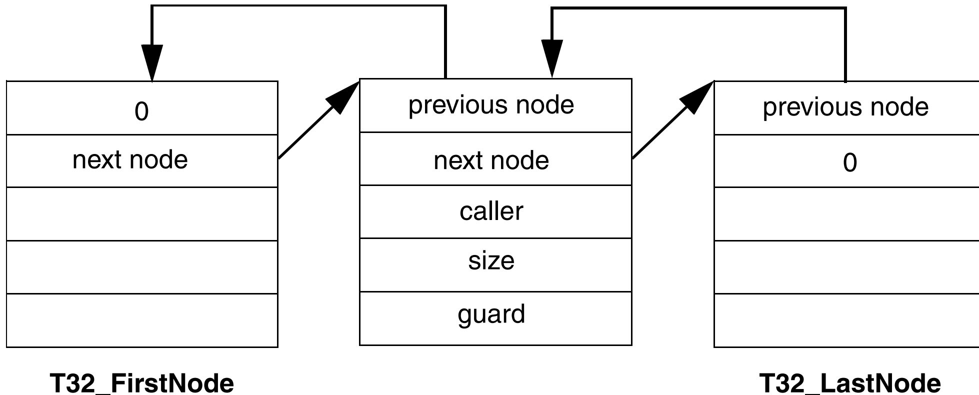
1. A memory block is allocated. This memory block has the following size:
   size of the requested memory block + sizeof(T32_allocHeader) + T32_GUARD_SIZE

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
```

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 on the TRACE32 software CD under ***\dos\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 CD.

```
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.

```
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
```

Data.Assemble

Inline assembler

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

Data.Assemble is used to replace the code at memory <*address*> with the assembler instruction specified by <*mnemonic*>; <*mnemonic*> describes the instruction with respect to the CPU-mnemonic.

The command entry is supported by softkeys. If no address is specified, the entry will be attached to the address following on from the last **Data.Assemble** command. To quickly modify a code line use the command "Modify here" or "Assemble here" from the pull down 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
```

See also

- 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.

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

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

See also

- Data.CHAIN

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 under:
`…\demo\<cpu>\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 Result

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

max size	Maximum block size used during the test
tolerance	Tolerance of the **RunTime** counters. Depending of the implementation the **RunTime** counters the result of the time measurement deviates slightly.
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.
block read	Bandwidth for block read
block write	Bandwidth for block write
lib write	Bandwidth for block write function provided by the compiler library
block copy	Bandwidth for block copy algorithm of the benchmark program
lib copy	Bandwidth for block copy function provided by the compiler library
random read	Frequency of random read operations
random write	Frequency of random write operations
random copy	Frequency of random copy operations
latency near	lateNcy until data is available for near addresses
latency far	laTency until data is available for far addresses

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 comma

; Data.BENCHMARK <program_range> <data_stack_range> <data_test_range>
; <IC_size> <DC_size> <L2_size> <L3_size>

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

Format:	**Data.CHAIN** <*base*> <*link offset*> <*elements*> [**/**<*option*> …]
<*elements*>:	**[[%**<*format*>] [<*address*> | <*range*>] …]
<*format*>:	**Decimal.** <*width*> **DecimalU.** <*width*> **Hex.** <*width*> **Ascii.** <*width*> **Binary.** <*width*> **Float. Ieee | IeeeDbl | IeeeeXt |** <*others*> **Var** **DUMP** <*width*>
<*width*>:	**Byte | Word | Long | Quad | TByte | TWord**
<*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.

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 clicking on a data word, 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 the Data 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.BDTAB
- Data.CHAINFind
- Data.dump
- Data.TABle
- Data.View

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

Data.CHAINFind

Search in linked list

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.

```
; 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

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

The contents of a memory area is compared 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 effects 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

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

Format:	**Data.COPY** *<addressrange>* [*<address>*] [**/***<option>*]
<option>:	**Verify** **Byte | Word | Long**

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

```
;-------------------------------------------------------------------
; 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
```

See also

■ Data.ComPare

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

Format 1:	**Data.DRAW** [*%format*] *<range>* [*<range>* …] [*<float>* [*<float>*]] [/*<options>*…]
Format 2:	**Data.DRAW** [*%format*] *<range>* *<range>* [*<float>* [*<float>*]] [/*<options>*]
<format>:	**Decimal.** [*<width>*] **DecimalU.** [*<width>*] **Hex.** [*<width>*] **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeMFFP** | …]
<width>:	**DEFault** | **Byte** | **Word** | **Long** | **Quad** | **TByte** | **TWord**
<options> 1:	**Track** | *<drawoptions>* | *<arrayoptions>* | **LOG**
<options> 2:	**Track** | *<drawoptions>* | **XY** | **YX** | **LOG**
<draw_ options>:	**Points** | **Vector** | **Steps** | **Impulses**
<array_ options>:	**Alternate** *<dimension_of_array>* | **Element** *<elements_to draw_in_window>*
	1..6
	1..n

(1): Visualizes the contents of up to six arrays as a graph. Each array value is the y-value in a coordinate system. The corresponding x-value results from the position of the y-value within the array.
Example: If the value of `myArray[5]` is `10`, then the x-value is `5` and the y-value is `10`.

(2): Visualizes the contents of one array as a graph. Here the x- and y-values are in separate arrays and are correlated to form the graph.
Example: If the value of `myFirstArray[7]` is `10` and the value of `mySecondArray[7]` is `5`, then the x-value is `5` and the y-value is `10`.

The **Data.DRAW** command can be 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 can be used to display HLL arrays graphically.

For a description of the parameters including some examples, see below.

<format>

- **Decimal, DecimalU** (Decimal Unsigned), **Hex**, and **Float** format the display of the y-axis.
- <width> DEFault = Byte
- **Float.** The following floating-point formats are available:
  - 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

<range>

range and range…: Memory range where an array is located. You can specify up to six independent arrays. For each additional array, an additional graph is displayed in the **Data.DRAW** window.

range and range: Values for the x and y-axis. See also **XY** and **YX** below.

tbd.

<float>

The first float specifies the units per pixel on the y-axis.
The second float specifies the starting point on the y-axis.

tbd.

<options>

Track: Clicking a value in the **Data.dump** window highlights the corresponding position in the diagram.

XY: The first <range> represents the values on the x-axis, the second <range> represents the corresponding values on the y-axis.

YX: Switches x and y-axis.

LOG: Displays the data values in a logarithmic format.

<draw_options>

Available options are **Points**, **Vector**, **Steps**, **Impulses**. Default = **Vector**. These options specify if and how a point in the diagram is connected to its neighbours.

tbd.

<array_options>

Alternate: Specifies the array structure to be visualized.

Element: Specifies the elements to be drawn.

Example 1:	Example 2:	Example 3:
`/Alternate 3` draws one graph each for **a**, **b**, and **c**:	`/Alternate 3 /Element 2` draws one graph for only **b**:	`/Alternate 3 /Element 2 3` draws one graph each for **b** and **c**:
a_0	a_0	a_0
b_0	b_0	b_0
c_0	c_0	c_0
a_1	a_1	a_1
b_1	b_1	b_1
c_1	c_1	c_1
...	...	...

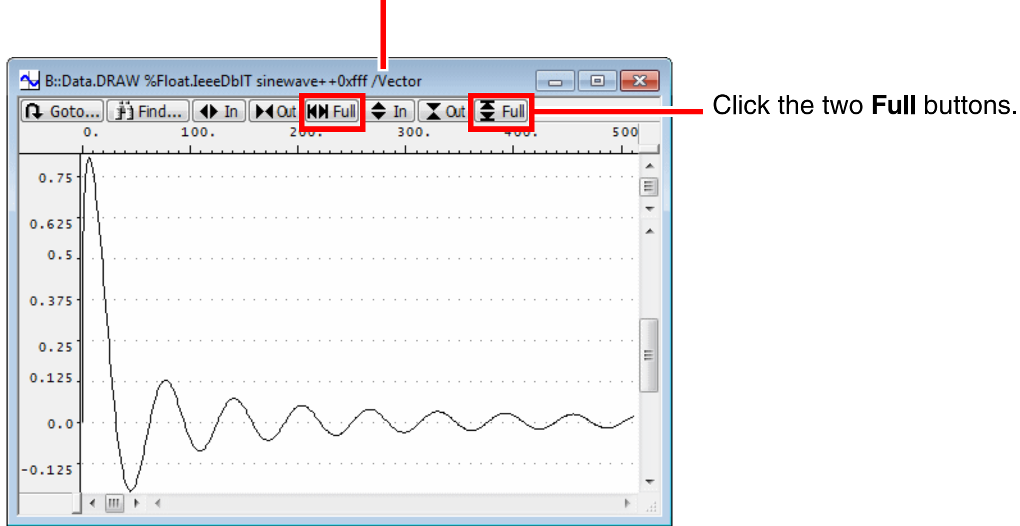
Example for Format 1

```
SYStem.Up

; Load file from current working directory.
Data.LOAD.Elf armle.axf /SourcePATH /LowerPATH

Register.Set PC main ; Set the program counter to the label main.
Go ; Same effect as a mouse-click on the Go button.
WAIT 2.s
Break ; Same effect as a mouse-click on the Break button.

; See result 1.
;         <format>            <range>     <drawoption>
Data.DRAW %Float.IeeeDblT sinewave++0xfff /Vector


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

```
; See result 3.
;          <format>          <range>              <range>       <drawoption>
Data.DRAW %Float.IeeeDblT sinewave++0xfff (sinewave+0x60)++0xfff
                                                                /Impulses

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

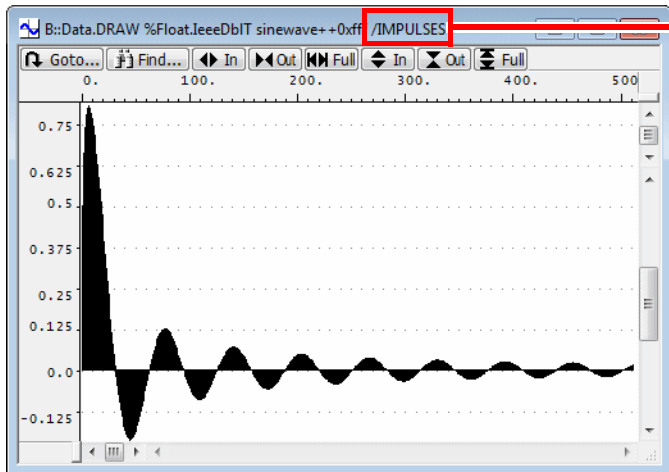
Result 1:

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



Click the two **Full** buttons.

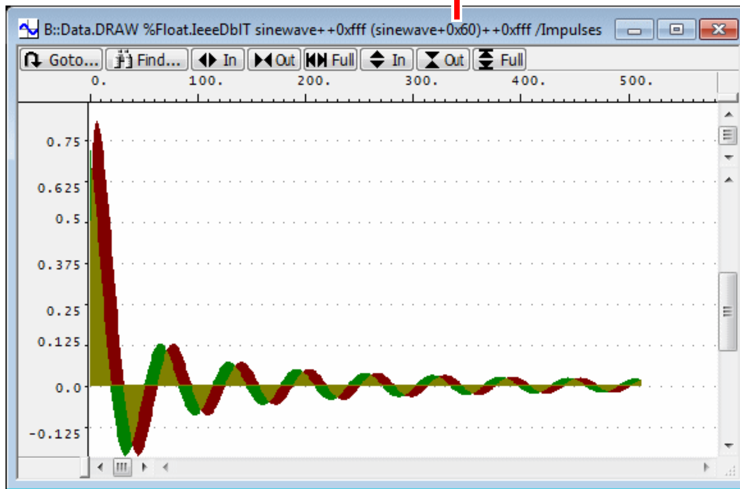
Result 2:



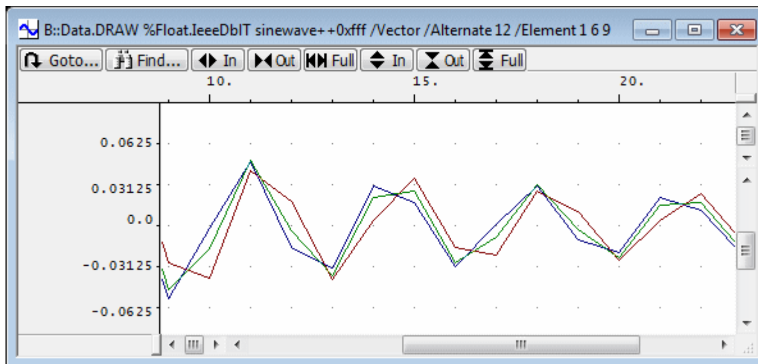
Here, the **/IMPULSES** option is used.

Result 3:

Here, an offset is used for the second <range>.



Result 4:



See also

■ Data.IMAGE

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

Format:	**Data.DRAWFFT** [**%**<*format*>] <*range*> <*scale*> <*fftsize*> [**/**<*window_option*>] [**/Real** | **/COMPLEX**]
<format>:	**Decimal.**[<*width*>] **DecimalU.**[<*width*>] **Hex.**[<*width*>] **Float.**[**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeMFFP** | …]
<width>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **TWord**
<window _option>:	**BLACKMAN** **HAMMING** **HANN**

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.

For a description of the parameters including an example illustrated with screenshots, see below.

<format>

- **Decimal, DecimalU** (Decimal Unsigned), **Hex**, and **Float** format the display of the y-axis.
- <width> DEFault = Byte
- **Float.** The following floating-point formats are available:
  - 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>

The scale is a scale factor (normalization) for the x-axis: The spectrum will range from 0 to <scale> divided by 2; see figure below.

- Enter the scaling factor for the x-axis as a float value.
- If you specify the sampling rate of your data as <scale>, then the x axis labels the frequency.
  Example: For a CD, the <scale> is 44100.0

<fftsize>

Enter the number of points as an integer value. The number of points must be 2 to the power of x, where x being an integer.
Example: 2 to the power of 9 = 512. Entering for example 514 as <fftsize> results in an error.

<window _option>

BLACKMAN: Folds input data with Blackman window before FFT computation

HAMMING: Folds input data with Hamming window before FFT computation

HANN: Fold input data with Hann window before FFT computation

REAL

Input data consists of real-valued numbers.

COMPLEX

Input data consists of complex-valued numbers.

Example for Data.DRAWFFT

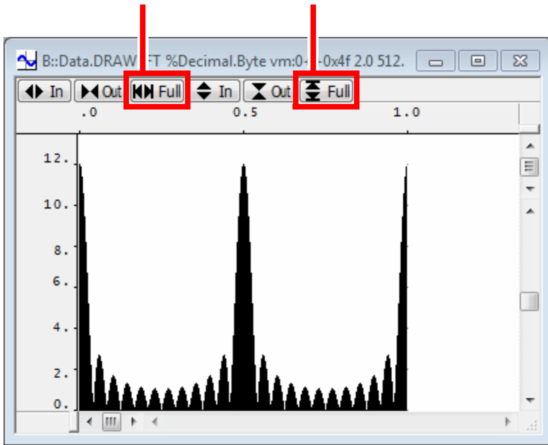
```
; Set a test pattern to the virtual memory of the TRACE32 application.
Data.Set VM:0--0x2f %Byte 1 0 0 0

Data.dump VM:0x0 ; Open the Data.dump window.

; 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.

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

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

The following figures show how the graphs change for the different formats: `Decimal.Byte`, `Word`, `Long`, `Quad`, `TByte`, and `TWord`.

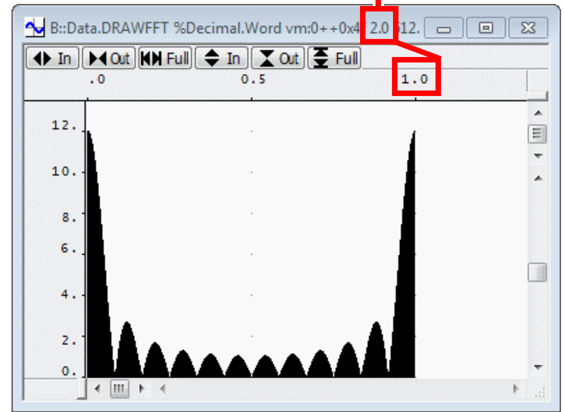
Result:

<scale> = 2.0
2.0 divided by 2 = 1.0

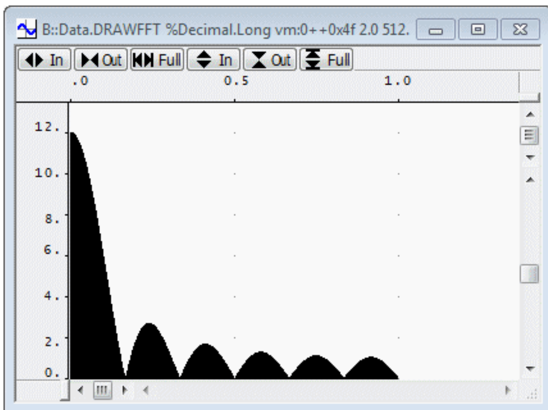
Resize the window, and then click the two **Full** buttons.



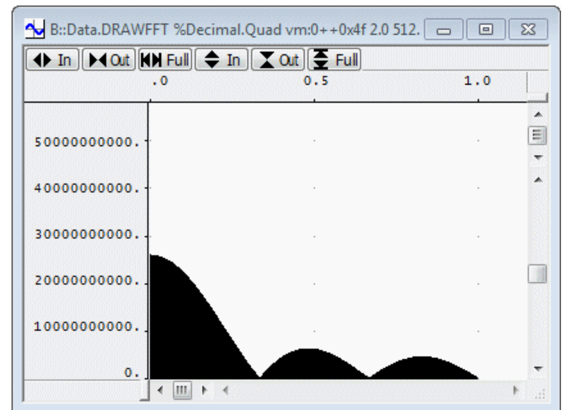
<format>: Decimal.Byte



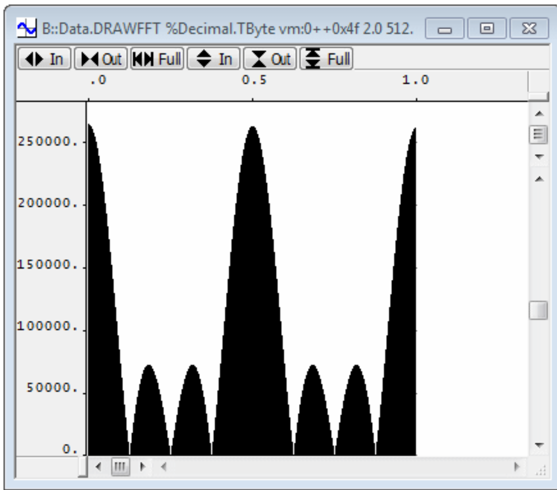
<format>: Decimal.Word



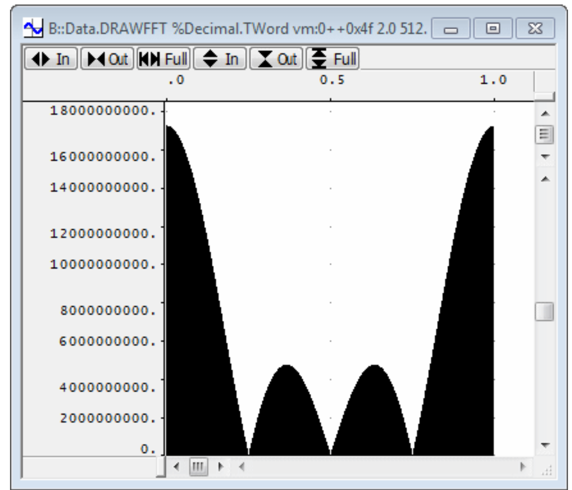
<format>: Decimal.Long



<format>: Decimal.Quad



<format>: Decimal.TByte



<format>: Decimal.TWord

Format:	**Data.DRAWXY** [*%format*] *<range> <range>* [*<float>* [*<float>*]] [/*<options>*]
<format>:	**Decimal.**[*<width>*] **DecimalU.**[*<width>*] **Hex.**[*<width>*] **Float.**[**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeMFFP** | …]
<width>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **TWord**
<options>:	**Track** | *<drawoptions>* | *<arrayoptions>* | **XY** | **YX** | **LOG**
<draw_ options>:	**Points** | **Vector** | **Steps** | **Impulses**
<array_ options>:	**Alternate** *<dimension_of_array>* | **Element** *<elements_to_draw_in_window>*

Draws a graph whose points are available as x and y coordinates.

<format>

- **Decimal, DecimalU** (Decimal Unsigned), **Hex**, and **Float** format the display of the y-axis.
- <width> DEFault = Byte
- **Float.** The following floating-point formats are available:
  - 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

<range>

<range> and <range> specify the values for the x and y-axis. See also **XY** and **YX** below.

<float>

The first float specifies the units per pixel on the y-axis.
The second float specifies the starting point on the y-axis.

tbd.

<options>

TRACK: Clicking a value in a **Data.DUMP** window highlights the corresponding position in the diagram.

XY: The first <range> represents the values on the x-axis, the second <range> represents the corresponding values on the y-axis.

YX: Switches x and y-axis.

LOG: Displays the data values in a logarithmic format.

<draw_options>

Available options are **Points**, **Vector**, **Steps**, **Impulses**. Default = **Vector**. These options specify if and how a point in the diagram is connected to its neighbours.

tbd.

<array_options>

Alternate: Specifies the array structure to be visualized.

Element: Specifies the elements to be drawn.

Example for Data.DRAWXY

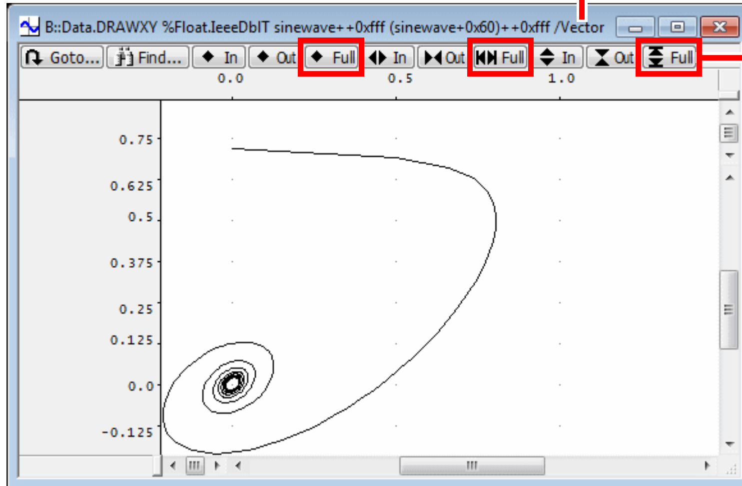
```
SYStem.Up

; Load file from current working directory.
Data.LOAD.Elf armle.axf /SourcePATH /LowerPATH

Register.Set PC main ; Set the program counter to the label main.
Go ; Same effect as a mouse-click on the Go button.
WAIT 2.s
Break ; Same effect as a mouse-click on the Break button.

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

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

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

Result 1:

The **/Vector** option is used.

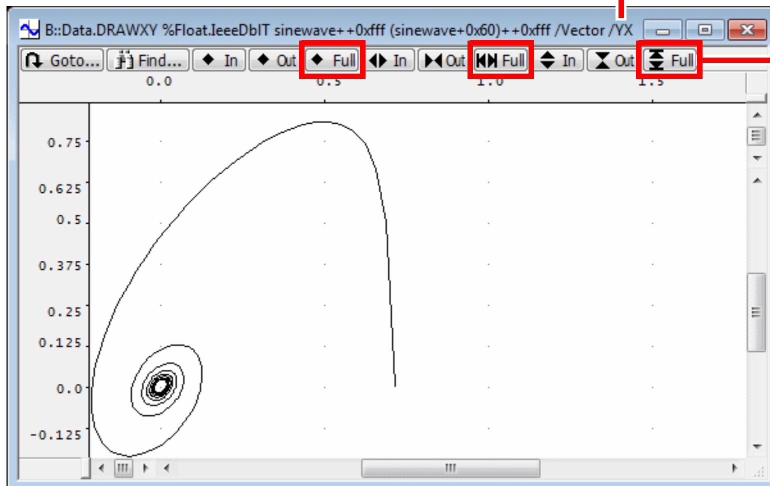
Click the three **Full** buttons.



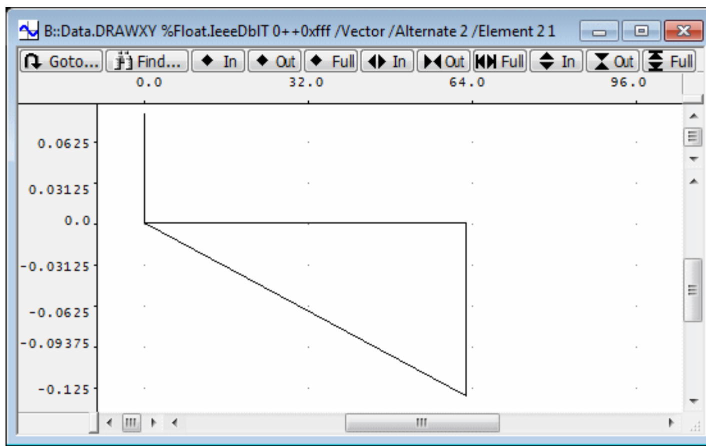
Result 2:

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

Click the three **Full** buttons.



Result 3:



Format: **Data.dump** [<*address*> | <*range*>] [**/**<*format*> | <*option*> …]

<*format*>: **NoHex | NoAscii**
Byte | Word | Long | Quad | TByte | TWord
BE | LE
PC8

<*option*>: **Orient**
NoOrient
WIDTH [<*columns*>]
Mark <*break*>
Flag <*flag*>
Track
CACHE

<*flag*>: **Read | Write | NoRead | NoWrite**

<*break*>: **Program | Hll | Spot | Read | Write | Alpha | Beta | Charly | Delta | Echo**

If a single address is selected this address will define the windows' initial position only. Scrolling makes other memory contents visible. When selecting an address range only the defined data range can be dumped. Range definition is useful whenever the addresses following are read protected (e.g., in the case of I/O) or more than one page will be printed.

The defaults of this command can be defined with the command **SETUP.DUMP**. The command **Var.DUMP** can display the dump as the result of a HLL expression.

```
Single address
                                       ┌ Pointer

E::Data.dump 0x465
   address 0  1  2  3  4  5  6  7  01234567
  D:000460|49 82 5E 5A E8►4A 0C 3F I.^Z.J.?
  D:000468|3F 3F 3F 53 54 41 43 4B ???STACK
  D:000470|54 4F 50 F1 C9 4A 00 08 TOP..J..
  D:000478|E2 C9 4A 82 20 00 E8 4B ..J. ..K
```

```
Address range

E::Data.dump 0x465--0x474
   address 0  1  2  3  4  5  6  7  01234567
  D:000460|              ►4A 0C 3F      J.?
  D:000468|3F 3F 3F 53 54 41 43 4B ???STACK
  D:000470|54 4F 50 F1 C9          TOP..
  D:000478|
```

```
Argument tracking

E::Data.dump Register(a2) /Track /Byte
wrCBAWRSHP     address    0  1  2  3  4  5  6  7  01234567
            SD:00002768|00 00 00 00 00 00 00 00 ........
wr          SD:00002770|00 00 00 00 01 01 01 00 ........
wr          SD:00002778|01 01 00 01 01 00 01 00 ........
wr          SD:00002780|00 01 01 00 00 01 00 00 ........
            SD:00002788|00 00 00 00►00 00 00 00 ........
            SD:00002790|00 00 00 00 00 00 00 00 ........
            SD:00002798|00 00 00 00 00 00 00 00 ........
            SD:000027A0|00 00 00 00 00 00 00 00 ........
            SD:000027A8|00 00 00 00 00 00 00 00 ........
```

WIDTH	Define display width in columns. Default automatically adjusts to selected window setting.
NoHex	Hex display is suppressed.
NoAscii	ASCII display is suppressed.
Byte, Word, Long, Quad, TByte, TWord	Define display width (1, 2, 4, 8, 3, 6 bytes).
BE, LE	Define display direction, big endian or little endian.

PC8	Displays the ASCII part in IBM PC8 character set (partial implementation).
Flag (E,F)	By using the Flag option individual bytes are marked, dependent of their flags.
Mark	By using the Mark option individual bytes are marked, depending on the breakpoint type set to the byte.
Orient	The most left addresses are match to bounds of potencies of two. The starting address is marked with a small arrow.
NoOrient	The dump starts exactly at the given address. The width of the display is matched exactly.
CACHE	Displays cache hit information and marks currently cached code.
Track	Tracks the window to the reference position of other windows. The window tries first to track to the DATA reference, and if this reference is not valid, it tracks to the PROGRAM reference. If this option is combined with an 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.

Example for the option **MARK**

```
F::Data.dump flags /MARK WRITE
  address 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
 D:20A370|1F FF 00 FF FF FF FF FF FF FF 00 07 FF FF►01 01 ................
 D:20A380|01 00 01 01 00 01 01 00 01 00 00 01 01 00 00 01 ................
 D:20A390|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 D:20A3A0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 D:20A3B0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 D:20A3C0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
 D:20A3D0|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

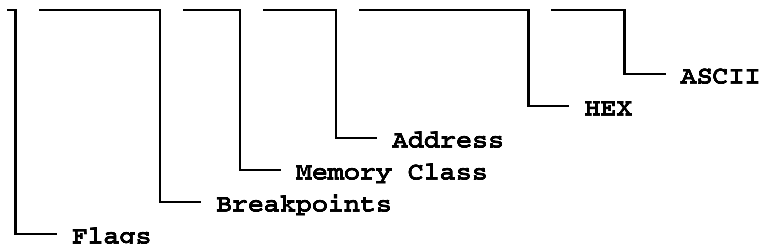
```
Data.dump 0x1234 /Word                  ; memory display by word as of
                                        ; address 1234H,

Data.dump Register(ix)                  ; dump data indexed by register ix

Data.dump Register(ix) /Track           ; same as above, but keep pointer
                                        ; in mid of window

Data.dump Register(ix)++0x0f            ; dump 16 bytes indexed by
                                        ; register ix

Data.dump Register(ix)+Register(a)      ; dump from address ix+a

Data.dump Data.Word(d:Register(ix))     ; dump with pointer indirect
```

Additional example for TRACE32-ICE and TRACE32-FIRE:

```
Data.dump /Flag Write                   ; display address starting 0x0
                                        ; mark all write-to addresses.
```

The columns contain addresses and memory classes, as well as Flags and Breakpoints. The status line displays all current addresses, both in hexadecimal and symbolic format.

```
E::Data.dump 0x456
wrCBAWRSHP      address  0   1   2   3   0123
                D:000464 E8►4A  0C  3F  .J.?
     W          D:000468 3F  3F  3F  53  ???S
      R         D:00046C 54  41  43  4B  TACK
                D:000470 54  4F  50  F1  TOP.
```



D:00046C

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

See also

- ■ Data.Assemble
- ■ Data.Find
- ■ Data.ListAsm
- ■ Data.ListJava
- ■ Data.PATTERN
- ■ Data.STRING
- ■ Data.Test
- ■ Data.View
- ■ SETUP.DUMP
- ■ SETUP.TIMEOUT
- ❏ ADDRESS.DATA()
- ❏ ADDRESS.ONCHIP()
- ❏ ADDRESS.SEGMENT()
- ❏ ADDRESS.WIDTH()
- ❏ Data.Byte()
- ❏ Data.Long()
- ❏ Data.Long.LittleEndian()
- ❏ Data.LongLong.BigEndian()
- ❏ Data.Quad()
- ❏ Data.Quad.LittleEndian()
- ❏ Data.Short.BigEndian()
- ❏ Data.SLong()
- ❏ Data.SUM()
- ❏ Data.Word()
- ❏ Data.Word.LittleEndian()
- ■ Data.CHAIN
- ■ Data.List
- ■ Data.ListHll
- ■ Data.ListMix
- ■ Data.Print
- ■ Data.TABle
- ■ Data.USRACCESS
- ■ SETUP.ASCIITEXT
- ■ SETUP.FLIST
- ❏ ADDRESS.ACCESS()
- ❏ ADDRESS.OFFSET()
- ❏ ADDRESS.PROGRAM()
- ❏ ADDRESS.SPACE()
- ❏ Data.AL.ERRORS()
- ❏ Data.Float()
- ❏ Data.Long.BigEndian()
- ❏ Data.LongLong()
- ❏ Data.LongLong.LittleEndian()
- ❏ Data.Quad.BigEndian()
- ❏ Data.Short()
- ❏ Data.Short.LittleEndian()
- ❏ Data.STRing()
- ❏ Data.TByte()
- ❏ Data.Word.BigEndian()
- ❏ Data.WSTRING()

'Data Access' in 'EPROM/FLASH Simulator'

'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.EPILOG — Sequence after stop

The command **Data.EPILOG** allows to define a sequence of **Data.Set** commands, that are automatically executed by the TRACE32 software directly after the program execution was stopped.

The two commands **Data.EPILOG.SEQuence** and **Data.PROLOG.Sequence** offer the option to switch off timers, communication interfaces etc. every time the program execution is stopped (**Data.EPILOG**) and to restart those components every time the program execution is started again (**Data.PROLOG**).

These commands are helpful if the processor doesn't automatically freeze those hardware components when the program execution is stopped.

Data.EPILOG.CONDition — Define EPILOG condition

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.

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

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

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

See also

- Data.EPILOG.state

Data.EPILOG.OFF

Switch EPILOG off

Format:	**Data.EPILOG.OFF**

Switches the **Data.EPILOG** feature off.

See also

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

Data.EPILOG.ON

Switch EPILOG on

Format:	**Data.EPILOG.ON**

Switches the **Data.EPILOG** feature on.

See also

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

Data.EPILOG.RESet

Reset EPILOG

Format:	**Data.EPILOG.RESet**

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

See also

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

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

The command **Data.EPILOG.SEQuence** defines a sequence of **Data.Set** commands, that are automatically executed by the TRACE32 software directly after the program execution is stopped.

SET	Write *<data>* to *<address>* directly after the program execution was stopped.
SETI	Write *<data>* to *<address>* the first time after the program execution is stopped after **Data.PROLOG.ON**. Then *<data>* is incremented by *<increment>* every time the program execution is stopped.
SETS	Write the data that was saved with a previous GETS back to *<address>*. **Data.EPILOG.SETS** can also use data saved with **Data.PROLOG.SEQuence GETS**.
GETS	Save the data at *<address>* directly after the program execution was stopped. Saved data can later be restored by **Data.EPILOG.SEQuence SETS** or **Data.PROLOG.SEQuence SETS**. It is possible to store several data.

```
Data.EPILOG.SEQuence SET 0x3faf50 %Word 0xa0a0 SET 0x3faf52 %Word 0xb0b0

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

Data.EPILOG.SEQuence SETS 0x3faf60

Data.EPILOG.SEQuence GETS 0x3faf60
```

See also

- Data.EPILOG.state

Data.EPILOG.state   EPILOG state display

Format: **Data.EPILOG.state**

```
B::Data.EPILOG
 epilog          CONDition
   OFF
√  ON            SEQuence
                  SET 0x3faf30 %Word 0xa0a0a0

 count           TARGET
17.
```

See also

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

Data.EPILOG.TARGET   Define EPILOG target call

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

The command **Data.EPILOG.TARGET** 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.state

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

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.

```
; 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 effects 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** set-up the search, but doesn´t process it. This can be benefical 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.dump
- Data.GOTO
- Data.GREP
- sYmbol.MATCH

- ADDRESS.OFFSET()
- FOUND()
- TRACK.ADDRESS()

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

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

The given address is used for tracking the windows (like find/compare commands).

See also

- ■ Data.Find
- ❏ ADDRESS.ACCESS()
- ❏ ADDRESS.DATA()
- ❏ ADDRESS.OFFSET()
- ❏ ADDRESS.ONCHIP()
- ❏ ADDRESS.PROGRAM()
- ❏ ADDRESS.SEGMENT()
- ❏ ADDRESS.SPACE()
- ❏ ADDRESS.WIDTH()
- ❏ Data.AL.ERRORS()
- ❏ Data.Byte()
- ❏ Data.Float()
- ❏ Data.Long()
- ❏ Data.Long.BigEndian()
- ❏ Data.Long.LittleEndian()
- ❏ Data.LongLong()
- ❏ Data.LongLong.BigEndian()
- ❏ Data.LongLong.LittleEndian()
- ❏ Data.Quad()
- ❏ Data.Quad.BigEndian()
- ❏ Data.Quad.LittleEndian()
- ❏ Data.Short()
- ❏ Data.Short.BigEndian()
- ❏ Data.Short.LittleEndian()
- ❏ Data.SLong()
- ❏ Data.STRing()
- ❏ Data.SUM()
- ❏ Data.TByte()
- ❏ Data.Word()
- ❏ Data.Word.BigEndian()
- ❏ Data.Word.LittleEndian()
- ❏ Data.WSTRING()

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

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

For the other options see **Data.List**.

The command **Data.GREP** allows to search for a specific string in a (all) source files.

Word	Search for the whole word.
Case	Perform a case sensitive search.

```
Data.GREP “i”                     ; search for “i“in all source files

Data.GREP “i” diabc1.c            ; search for “i“ in diabc1.c

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

```
B::Data.GREP "i" *.* /Word
  hits:    11 source
              diabc1.c
         249          int i:1;
         319          register i, j;
         324          for ( i = 0 ; i < 3 ; i++ )
         325                  v17 += i;
         360          i = j = v17;
         616          register int i, primz, k;
         621          for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
         623          for ( i = 0 ; i <= SIZE ; i++ )
         625                  if ( flags[ i ] )
         627                          primz = i + i + 3;
         628                          k = i + primz;
```

See also

- APU.GREP
- Data.Find

Format:	**Data.IMAGE** *<address>* *<horiz>* *<vert>* [<scale> | **/***<format>* | **/***<options>*]
<format>:	**MONO, CGA, GrayScale8, JPEG** **RGB111, RGB555, RGB555LE, RGB565, RGB565LE,** **RGB888, RGB888LE, RGBX888, RGBX888LE,** **YUV420, YUV420W, YUV420WS** **YUV422, YUV422W, YUV422WS, YUV422P, YUV422PS** **Palette256** *<red>* *<green>* *<blue>* *…* **Palette256X6** *<address>* **Palette256X12** *<address>* **Palette256X24** *<address>*
<options>:	**BottomUp** **FullUpdate** **STRIDE** *<bytesperstride>*

Display graphic bitmap data.

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.
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.
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).

NOTE: Values have to be given with postfix'.' because of decimal number base.

Zooming is supported by scrolling the mouse wheel or double-clicking the image.

Right-clicking an image allows advanced data operations.

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.
```

See also

- Data.DRAW

  'Release Information' in 'Release History'

Format:	**Data.In** *<address>* *<length>* [**/***<format>*]
<format>:	**NoHex** | **NoAscii** **Byte** | **Word** | **Long** | **Quad** | **TByte** | **TWord** **BE** | **LE** **Repeat**

As opposed to the **Data.dump** command, the address is not increased during reading. Data is only read **once** and is displayed in the message line. The **Repeat** option repeats the input endless, e.g. for external measurements.

```
Data.In io:0x10                ; get one byte in I/O access area
Data.In d:0x10                 ; get one byte from memory-mapped I/O
Data.In io:0x44 3.             ; get 3 bytes from port 44H
```


See also

- ■ Data.Out
- ■ Data.View
- ❏ ADDRESS.ACCESS()
- ❏ ADDRESS.DATA()
- ❏ ADDRESS.OFFSET()
- ❏ ADDRESS.ONCHIP()
- ❏ ADDRESS.PROGRAM()
- ❏ ADDRESS.SEGMENT()
- ❏ ADDRESS.SPACE()
- ❏ ADDRESS.WIDTH()
- ❏ Data.AL.ERRORS()
- ❏ Data.Byte()
- ❏ Data.Float()
- ❏ Data.Long()
- ❏ Data.Long.BigEndian()
- ❏ Data.Long.LittleEndian()
- ❏ Data.LongLong()
- ❏ Data.LongLong.BigEndian()
- ❏ Data.LongLong.LittleEndian()
- ❏ Data.Quad()
- ❏ Data.Quad.BigEndian()
- ❏ Data.Quad.LittleEndian()
- ❏ Data.Short()
- ❏ Data.Short.BigEndian()
- ❏ Data.Short.LittleEndian()
- ❏ Data.SLong()
- ❏ Data.STRing()
- ❏ Data.SUM()
- ❏ Data.TByte()
- ❏ Data.Word()
- ❏ Data.Word.BigEndian()
- ❏ Data.Word.LittleEndian()
- ❏ Data.WSTRING()

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

Format:	**Data.List** [<*address*> | <*range*>] [**/**<*option*>]
<*option*>:	**Mark** <*break*> **Flag** <*flag*>(EF) **COVerage** **CACHE** **Track** **TOrder** | **SOrder** **ISTAT** <*item*>
<*flag*>:	**Read | Write | NoRead | NoWrite**
<*break*>:	**Program | Hll | Spot | Read | Write | Alpha | Beta | Charly | Delta | Echo**
<*item*>:	**DEFault | ALL | CLOCKS | TCLOCKS | SAMPLES | COVerage**

Display format (in assembler, mixed or HLL) is selected dynamically, depending on the current debug mode which could be chosen by the **Mode** command. If no address is specified, the window tracks to the value of the program counter. The window is only scrolled, if the bar moves outside of a predefined subwindow. The display format may be specified with the **SETUP.DIS** command.

Mark	By using the Mark option individual lines can be marked, depending on the value of their breakpoints.
MarkPC	By using the MarkPC option all HLL source lines belonging to the current PC are high-lighted.
Flag	Mark a line with specific flag memory contents, e.g. Read
COVerage	Displays trace based code coverage information (**COVerage**).
CACHE	Displays cache hit information and marks currently cached code.
Track	Tracks the window to the reference position of other windows. The window tries first to track to the PROGRAM reference, and if this reference is not valid, it tracks to the DATA reference.
TOrder	List source lines in target line order, i.e. in the order they appear in the binary program. This is the default for assembly and mixed displays.

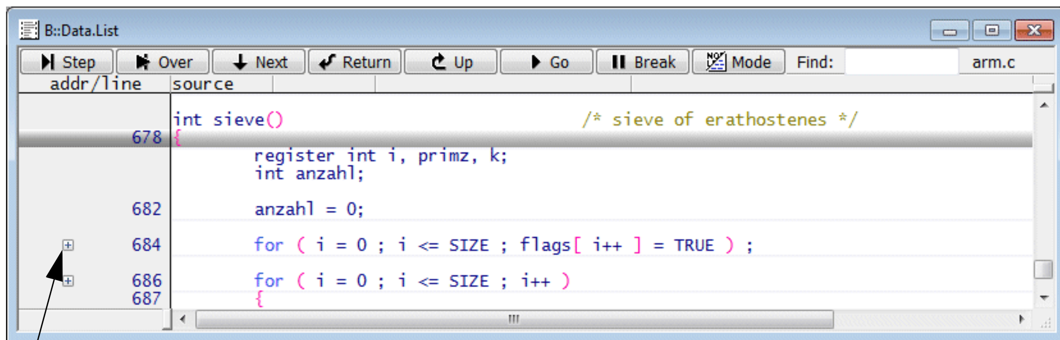
SOrder List source lines in source line order, i.e. in the order of the original program. This is the default for source level displays. If the compiler rearranged the lines during the optimization, the file must be loaded with the **Puzzled** option.

ISTAT Display source listing together with the information provided by the instruction trace data base (**ISTATistic.ListFunc**)

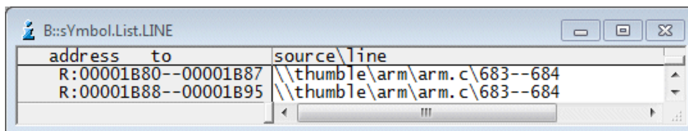
Parameters for the ISTAT option	
DEFault	Display the default information provided by the ISTAT data base.
ALL	Display all information provided by the ISTAT data base.
CLOCK	Display the clock and CPI information provided by the ISTAT data base.
TCLOCK	(only for special purposes)
SAMPLES	tbd.
COVerage	Display the code coverage information provided by the ISTAT data base.

```
Data.List                          ; display source listing around the
                                   ; current PC

Data.List /Mark Program            ; display source listing, bold print
                                   ; all instructions/hll lines on a
                                   ; yellow background if a program
                                   ; breakpoint is set

Data.List /Mark                    ; remove bold printing on yellow
                                   ; background


Data.List /Track                   ; track the window to a reference, e.g.
                                   analyzer

Data.List Register(a0)             ; follow the register A0 of the CPU

print d.l(d:0x200)                 ; prints the memory contents add
                                   D:0x200
                                   ; Do not mix-up the command D.L
                                   ("data.list")
                                   ; with the function D.L ("data.long").
```

If the source listing is display in HLL mode, some code lines may be marked with a drill-down button. This drill-down button indicates, that the compiler generated assembler code at more the one address location for this source code line. This is very common for for/while loops as well as for code compiled with a high optimization level.



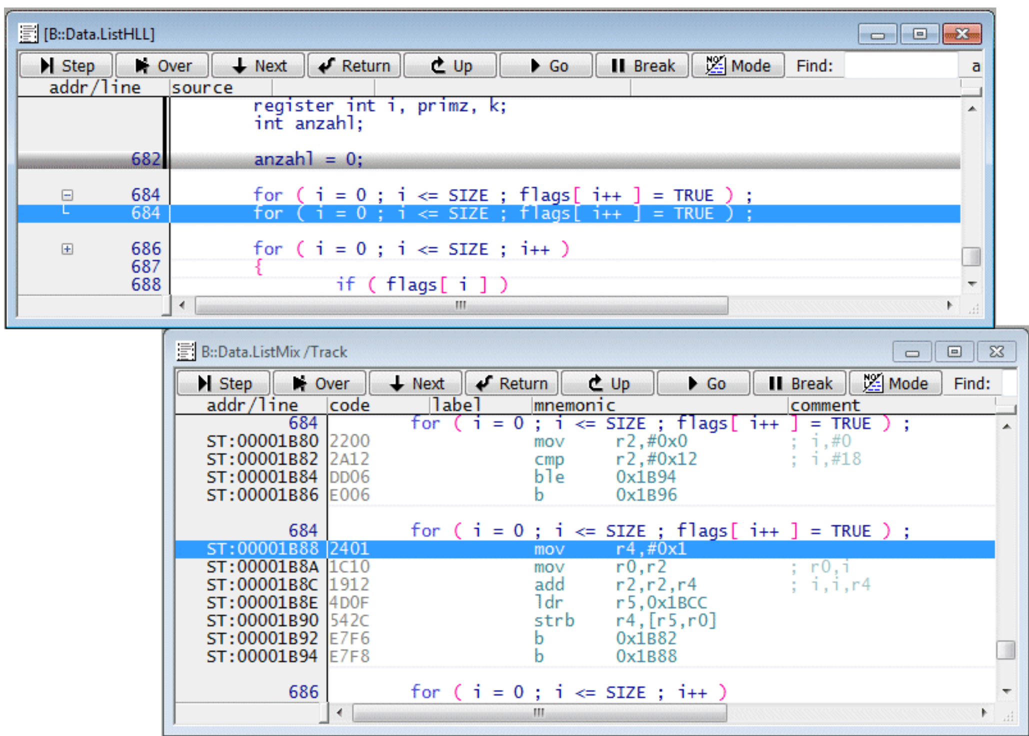
Drill-down button



Assembler code at two address locations is generated for the HLL line number 683--684

If you want to inspect this in detail, the following commands might be helpful:

```
Data.ListHll                          ; display the source code in HLL
                                      ; mode (source order)

Data.ListMix /Track                   ; display the source code in Mixed
                                      ; mode (target line order)
```



If you push the drill-down button to get detailed information, a duplicate of the original HLL line is generated for each assembler code address location.

If you now select one of the duplicates, the cursor in the **Data.ListMix** window points automatically to the corresponding assembler code. This feature is enabled by the **/Track** option.

Additional example for the TRACE32-ICE and TRACE32-FIRE:

```
Data.List e:                ; display the program around the current PC,
                            ; but use dual-port access
```

```
E::Data.List e: /Track
wrCBAWRSHP  addr/line     code            label    mnemonic                    c
 r      H            565                  if ( flags[ i ] )
 r      H  ESP:000017AA 4A1B                       tst.b     (a3)+
 r         ESP:000017AC 671E                       beq       $17CC
                                          {
 r      HP           567                           prime = i + i + 3;
 r      HP ESP:000017AE 2802                       move.l    d2,d4              ;
 r         ESP:000017B0 D882                       add.l     d2,d4              ;
 r         ESP:000017B2 5684                       addq.l    #3,d4              ;
 r      H            568                       k = i + prime;
 r      H  ESP:000017B4 2602                       move.l    d2,d3              ;
 r         ESP:000017B6 D684                       add.l     d4,d3              ;
```

See also

■ Data.dump ■ Data.ListAsm ■ Data.ListHll ■ Data.ListJava
■ Data.ListMix ■ SETUP.DIS ■ SETUP.FLIST ■ SETUP.LISTCLICK
■ SETUP.sYmbol ■ SETUP.TIMEOUT ❏ ADDRESS.ACCESS() ❏ ADDRESS.DATA()
❏ ADDRESS.OFFSET() ❏ ADDRESS.ONCHIP() ❏ ADDRESS.PROGRAM() ❏ ADDRESS.SEGMENT()
❏ ADDRESS.SPACE() ❏ ADDRESS.WIDTH()

'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'

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

<*option*>: **Mark** <*break*>
Flag <*flag*>(EF)
COVerage
CACHE
Track
TOrder | **SOrder**
IgnoreSymbols

Displays the program in disassembled format. The functionality is the same as the **Data.List** command.

Mark <*break*> **Flag** <*flag*>(EF) **COVerage** **CACHE** **Track** **TOrder** **SOrder**	See **Data.List** command.
IgnoreSymbols	Let the disassembler ignore any symbols for deciding at which byte of the machine code the disassembling should start. This command does normally only make sense on architectures with different instruction lengths.

```
E68::Data.ListAsm
   address  codes                label      mnemonic                      comment
 UP:000000 4E56FFF8              main_:     link     a6,#0FFFFFFF8
 UP:000004 48EF00200004                     movem.l  [d5],4(a7)
 UP:00000A 4245                             clr.w    d5
 UP:00000C 6002                             bra      $10
 UP:00000E 5245                             addq.w   #1,d5
 UP:000010 0C45000A                         cmpi.w   #0A,d5
 UP:000014 6CF4                             bge      $0A
 UP:000016 3E85                             move.w   d5,(a7)
 UP:000018 4EB90000002A                     jsr      $2A                  ; $sieb_
 UP:00001E 60EE                             bra      $0E
```

See also

- Data.dump
- Data.List

'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'

Format:	**Data.ListHll** [<*address*>] [**/**<<*option*>]
<*option*>:	**Mark** <*break*> **Flag** <*flag*>(EF) **COVerage** **CACHE** **Track** **TOrder** | **SOrder**

Displays the program in source format. The functionality is the same as the **Data.List** command. If the starting address in not a HLL (High Level Language) line, assembler code is displayed to the next HLL line found in the code segment. Shadowed lines signal that the source information is not on disk or in wrong directory. See command **sYmbol.SPATH**.

```
E68::Data.ListHll
     line  source
        8 |main()
        9 |{
       10 |        int j;
       11 |
       12 |        while ( TRUE )
       13 |        {
       14 |                for ( j = 0 ; j < 10 ; j++ )
       15 |                        sieb( j );
       16 |        }
```

See also

- Data.dump
- Data.List

'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'

Format:	**Data.ListJava** [<*address*>] [**/**<*option*>]
<*option*>:	**Mark** <*break*> **Flag** <*flag*>(EF) **COVerage** **CACHE** **Track** **TOrder** | **SOrder**

Displays the program in JAVA byte code format. The functionality is the same as the **Data.List** command. This command is NOT required when an address range is specified as JAVA byte code area with the JAVA byte code debugger or when the processor implements a special byte code mode (e.g. ARM Jazelle).

See also

- Data.dump
- Data.List

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

<*option*>: **Mark** <*break*>
Flag <*flag*>(EF)
COVerage
CACHE
Track
TOrder | **SOrder**

The code is displayed in HLL (High Level Language) and additionally disassembled from the memory. The functionality is the same as the **Data.List** command. Shadowed lines signal that the source information is not on disk or in wrong the directory. See the command **sYmbol.SPATH**.

```
E68::Data.ListMix
   address codes              label     mnemonic                      comment
         6 char flags[SIZE+1];
         7
         8 main()
 UP:000000 4E56FFF8           main_:    link    a6,#0FFFFFFF8
 UP:000004 48EF00200004                 movem.l [d5],4(a7)
         9 {
        10         int j;
        11
```

See also

- Data.dump
- Data.List

'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'

Format:	**Data.LOAD** *<filename>* [*<address>*] [*<range>*] [**/***<generic option>*] **Data.LOAD.auto** *<filename>* [*<address>*] [*<range>*] [**/***<generic option>*]
<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.
<generic option>:	**Verify** | **PVerify** | **NoVerify** | **CHECKLOAD** [*<address_range>*] **ComPare** | **DIFF** | **CHECKONLY** [*<address_range>*] **ZIPLOAD** [*<address_range>*] **DIFFLOAD** [*<address_range>*] **DualPort**(EF) **Byte** | **Word** | **Long** | **Quad** **BSPLIT** *<width>* *<offset>* **SWAP** **VM** | **PlusVM** **NOCODE** | **NosYmbol** | **NoReg** | **NoBreak** **NoClear** | **More** **PATH** *<dir>* **SOURCEPATH** *<dir>* **StripPATH** | **LowerPATH** **StripPART** *<parts>* | *<partname>* **TASK** *<magic>* **NAME** *<name>* **MAP** | **DIAG** **COLumns** | **MACRO** **FLAT** | **PFLAT** | **LARGE** | **TFLAT** | **LDT** (only 386) **SPlit** (only 80196) **CYGDRIVE**

With **Data.LOAD.auto** (or Data.LOAD), the debugger tries to detect the data format in the file automatically. 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 automatic detection is not possible for all formats. Also in this case use **Data.LOAD.<format>**.

Here is the description for the generic options:

Options which verify that code blocks were written error-free to the target memory	
Verify	Data memory is checked after write. This is done by reading back the data from target memory immediately after a block of memory has been written. This may not be sufficient to detect overwriting caused by memory address errors or wrong linkage. 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. Recommanded 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 comparsion stops after the first difference.
CHECKONLY	Data is compared against the file by calculating checksums. Memory is not changed. The comparsion stops when checksum is wrong. Recommanded 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()** function.

```
…                                         ; reprogram the FLASH
Data.LOAD.COFF arm.abs /DIFF              ; only if the code
IF FOUND()                                ; changed, otherwise
(                                         ; load only symbol and
     FLASH.Program ALL                    ; debug information plus
     Data.LOAD.COFF arm.abs               ; hll source path
     FLASH.Program OFF                    ; information
)
ELSE
     Data.LOAD.COFF arm.abs /NOCODE
…

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

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. Recommanded 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
```

NOCODE Suppress the code download. Only loads symbolic information.

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

NoReg Any startup values for registers (e.g. PC and SP) are not taken from the file.

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 realtime during HLL steps. Only relevant for TRACE32-ICE.
NoClear	Existing symbols are not deleted. This option is necessary if more programs must be loaded (Tasks, Overlays, Banking).
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.
VM	The TRACE32 software provides a virtual memory 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.
PlusVM	The code is loaded into target memory plus into the virtual memory.
Byte, Word, Long, Quad	Data is loaded in the specified width. Non aligned data is saved by a read-modify-write operation. 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.
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.
BSPLIT	Loads only certain bytes of the memory. The first argument defines the bus width in bytes, the second the offset of the byte being loaded. The option "BSPLIT 2 0" loads the lower byte of a 16-bit bus.
SWAP	Swap high and low bytes during load.
SOURCE-PATH	Define a new base directory for the source files in this program. This replaces the current working directory (that will be takes as a default).
PATH	If the sources are not found within the directory of the object file, they are searched within the directories given by this option. The option without parameter searches in the current directory. The option can be specified more than one time to include more directories in the search path. The command **sYmbol.SPATH** can be used to define more and permanent search routes.

StripPATH The basic name is extracted from the source pathes given in the object file. This option can be used when the pathes from the compiler should not be used, or the object files has been compiled on an other host with different file syntax (e.g. VAX/VMS under MS-DOS).

StripPART Parts of the file path stored in the object file are removed. The option takes either a number or a string as parameter. The number defines how many parts are removed. A string is searched in the path of the object file and everything from the first match of the string (including the string itself) is taken as the source path. This allows to change complete file structures from one base directory to another.

LowerPATH The file name is converted to lower-case characters. This option is useful, when object files are compiled under a host which uses only capital letters for file names (e.g. VAX/VMS, MS-DOS) and should be loaded in a unix environment.

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). See the target manual for more description about the usage of this option.

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).

MAP Loads memory load map information for the **sYmbol.List.MAP** command and checks for overlapping memory writes during download. Can be useful if the load map is questionable.

DIAG Used for diagnostic purposes only.

COLumns Loads information for single stepping columns in HLL mode. May not be available in all file formats.

MACRO Loads information from C Macros for HLL debugging. May not be available in all file formats.

CYGDRIVE Use this option to make TRACE32 aware of binary files compiled within a Cygwin environment (e.g. the Xilinx MB compiler). This will strip the prefix c:`\cygdrive\c\` from debug symbol so TRACE32 looks for source files at the correct location in the file system.

Example for option **PATH**.

```
Data.LOAD.Ieee mcc.abs /PATH \ste /PATH \mcc68k /PATH \asm68k
```

Program Loading

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.AsciiHex
- ■ Data.LOAD.AsciiOct
- ■ Data.LOAD.AVocet
- ❏ ADDRESS.DATA()

'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'
'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** **Include** **AnySym** **PACK** **FASTPACK** **RAMINIT** *<generic load option>*

Load 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.
Include	Loads source lines generated from include files.
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.

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.

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.

See also

- Data.LOAD
- sYmbol.AddInfo.LOADASAP2

'Release Information' in 'Release History'

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.

See also

- Data.LOAD

Data.LOAD.Binary

Load binary file

Format:	**Data.LOAD.Binary** <*filename*> [<*address*> | <*range*>] [**/**<*option*>]
<*option*>:	**SKIP** <*offset*> <*generic load option*>

Loads a plain binary file.

If the command is called without <*address*> and <*range*>, the complete file will be loaded to the target beginning from target address 0.

If <*address*> is specified, the complete file will be loaded to target address <*address*>.

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.

If ihe option /SKIP <offset> is specified, the first <offset> bytes of the file are omitted.

See also

- Data.SAVE.Binary

  'Release Information' in 'Release History'

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.

Format:	**Data.LOAD.CDB** *<filename>* [*<class>*] [**/***<option>*]
<option>:	**IntelHexFile** *<binary_file_name>* **NoIntelHexFile** **WarningsAll** **WarningsNo** *<generic load option>*

Load 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 corresconding file and loads debug information and code automaticly together (see options to avoid this behavoir). The binary part is stored in IntelHex-Format and can be load also separatly.

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

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
```

Options:

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.

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>*

Load 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 a 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.

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.

Data.LOAD.CORE — Load linux core dump file

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

Load a linux core dump file into the simulator. The object file have to be loaded before loading the core file. For example:

```
Data.LOAD.ELf object.elf              Load the object file

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

This command is available for ARM, PowerPC, MIPS32 and x86.

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.

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).

Format: **Data.LOAD.DBX** <*filename*> <*code*> <*data*> [<*sym*>] [**/**<*option*>]

<*option*>: **CPP**
Include
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.
Include	Activates the loading of sources from include files.
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.

Format:	**Data.LOAD.Elf** *<filename>* [*<memory_class>* | *<offset>* | *<range>*] \ [**/***<option>*]
<option>:	**FPU | NOFPU** **AnySym | ZeroSym** **PACK** **FASTPACK** **MMU** (only HC11/HC12) **CPP** **CODESEC | LOGLOAD** **GlobTypes** **NoGlobTypes** **MRI | GNU** **STABS | DWARF | DWARF2** **ALTBITFIELDS** **RelPATH** **RelPATH2** **REV** **ModulePATH** **RELOC** *<section>* **AT** *<address>* **RELOC** *<section>* **AFTER** *<section>* **RELOC** *<section>* **LIKE** *<section>* *<generic load option>*

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.

```
; Example for <memory class>
; 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
```

```
; 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
```

Options:

ALTBITFIELDS	This option might solve problems with regards to the display of bitfields.
AnySym	Load all symbols generated by the compiler (defining file names, local labels etc.) which are usually suppressed by the loader.
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, 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.
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, NOFPU	Indicates the debugger that the code for FPU or without FPU was generated by the 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.
LOGLOAD	Takes the logical address of the program table to load the code (instead of the physical address).

MMU	Loads information and translation tables for onchip MMUs (only HC11, HC12, Star12).
ModulePATH	Keeps the pathname 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.
NoGlobTypes	Can be set when there is no shared debug information in a file format where the loader expects them (e.g. for ARM).
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.
RELOC	Relocates code/symbols of the specified section to the a specified (virtual address) or after the specified section.
RelPATH	The option /RelPATH strips away the compilation directory information. The compilation directory is usually the working directory in which the compiler was invoked. 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 /RelPATH 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.
REV	Reverse bit fields. Must be set when the compiler option was set.
ZeroSym	By default modules linked to address 0x0 aren´t loaded. The option /**ZeroSym** advises the loader to also load all modules linked to address 0x0.

```
; 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
```

Data.LOAD.eXe — Load EXE file

Format:	**Data.LOAD.eXe** <*filename*> [<*access*> | <*address*>] [**/**<*option*>]
<*option*>:	**FPU** **NoMMU** **CPP** <*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.

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)

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.

Data.LOAD.HiTech — Load HITECH file

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.

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 setup 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.

Format:	**Data.LOAD.ICoff** <*filename*> [<*class*>] [**/**<*option*>]
<*option*>:	**FPU** **MOD** **Puzzled** <*generic load option*>

Loads files in Introl ICOFF format.

FPU	Tells the debugger that code for the FPU was generated by the compiler.
Puzzled	If the compiler rearranges the source lines, i.e. the lines are no longer linear growing, this option has to be used.
MOD	Adjusts the loader for INTROL MODULA2 files.

Format:	**Data.LOAD.Ieee** <*filename*> [<*address*> | <*access*>] [**/**<*opt.*>]
<*opt.*>:	**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.

FPU	Tells the debugger that code for a FPU (Floating Point Unit) was generated by the compiler.
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.
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.
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.
INT16	Specifies the size of integers to 16 bits.
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.
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.

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.
NOFRAME	Ignore the stack frame information in the IEEE file. The stack frame information in the file is used for the **Var.Frame** 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.
LIMITED	Doesn't load any type information. Loads the code and the source information only.
CFRONT	Load C++ files converted by a CFront preprocessor. NOTE: This option should not be used for Microtec C++ files.
ALSYS	Must be used when loading ALSYS IEEE files.
ZP2	Must be set, when the ZP2 option was used to compile the file.
MCC3	Must be used when loading MCC68K 3.0 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.
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).

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.

See also

- Data.SAVE.IntelHex

'Release Information' in 'Release History'

Format:	**Data.LOAD.Jedec** <*filename*>
<*option*>:	<*generic load option*>

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

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.

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

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 bedisplayed,
                                       ; 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
```

Options:

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.
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).
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.
NOCONST	Suppresses the load of "const" variables.
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.
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 univeral binary (FAT), the first entry (number=0) will be loaded regardless of its and the others architecture-codes.
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 univeral binary.

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.

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.

Data.LOAD.MCoff — Load MCOFF file

Format:	**Data.LOAD.MCoff** <*filename*> [<*range*> | <*class*>] [**/**<*option*>]
<*option*>:	**ALLLINE** <*generic load option*>

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

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

The following Compilers are accepted:

Format	Compiler	Remarks
OMF-51	Intel-PL/M	Use **LST** or **PLM** option
	Intel-C51	Use **C** option
	Keil-C51	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.
	SPJ-C	Use **PAS** option.
	KSC/System51	
OMF-96	Intel-C96	Use **SPLIT** option when code and data are two separate memory spaces.
OMF-86	Intel-PL/M	Use **LST** or **PLM** or **MIX** or **MIXPLM** option.
	Intel-iC86	Use **IC86** or **IC86OLD** option, **RevArgs** option when PASCAL calling conventions are used.
	Microtec	Use **MRI** option, includes extended type and source file information.
	Paradigm	Use **ParaDigm** option, includes extended type and source file information.
OMF-386	Pharlap	No option required, includes extended register variables information. Use the '-regvars' option to produce register variable information.
	SSI/Intel	No option required. The Codeview debugging format provides more information than the intel format info. No option required.
	SSI/CodeView	
	SSI/Metaware	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/i86/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.SAVE.Omf

Data.LOAD.REAL

Load R.E.A.L. file

Format:	**Data.LOAD.REAL** <*filename*> [**/**<*option*>]
<*option*>:	<*generic load option*>

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

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).

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.

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.

Format:	**Data.LOAD.S1record** <*filename*> [<*addressrange*>] [**/**<*option*>] **Data.LOAD.S2record** <*filename*> [<*addressrange*>] [**/**<*option*>] **Data.LOAD.S3record** <*filename*> [<*addressrange*>] [**/**<*option*>]
<*option*>:	**OFFSET** <*offset*> **<***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.

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.

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.

Specify the global SYM file as <*filename*>. Depending on it's 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
```

Data.LOAD.SysRof — Load RENESAS SYSROF file

Format:	**Data.LOAD.SysRof** <*filename*> [<*access*>] [**/**<*option*>]
<*option*>:	<*generic load option*>

Load a file in Renesas SYSROF object file format.

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.

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.

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 work around 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.

Data.LOAD.VersaDos

Load VERSADOS file

Format: **Data.LOAD.VersaDos** <*filename*> [<*access_class*>] [**/**<*option*>]

<*option*>: **NoLO**
NoDB
<*generic load option*>

The command loads first the ".lo" file, which contains the code, and then the ".db" symbol file (if existent). The file name of the ".lo" file must be given. With the option **NoDB** the loading of the symbols is suppressed. The option **NoLO** suppresses the loading of the data file. The symbol file name has to be given as an argument in this case.

Data.LOAD.XCoff

Load XCOFF file

Format: **Data.LOAD.XCoff** <*filename*> [<*class*>] [**/**<*option*>]

<*option*>: **ALLLINE**
<*generic load option*>

Load a file in the IBM-RS6000/XCOFF format (PowerPC).

Data.LOG

Log data accesses done by TRACE32

Format:	**Data.LOG** *<size>* [*<filename>*]

Records the read and write accesses that the TRACE32 tool performs to the target hardware and stops when an error occurred.

The *<size>* parameter is used to define the number of data accesses displayable in a Data.LOG window. It could be interpreted as history depth of the Data.LOG window too.

```
B::Data.LOG
 Start   Stop   Clear
access|address                  |width |data                             |time
READ  |   P:003F9924--003F9943  |   4. |48 00 12 49 48 00 11 3C ...      | 5402.696
READ  |   D:00000008--0000000B  |   4. |48 00 11 02                      | 5402.697
READ  |   P:003FA764--003FA783  |   4. |94 21 FF D8 7C 08 02 A6 ...      | 5402.781
READ  |   P:003FA784--003FA7A3  |   4. |39 20 00 02 99 2A AF 24 ...      | 5402.782
READ  |   P:003FA7A4--003FA7C3  |   4. |4B FF F1 DD 4B FF F2 6D ...      | 5402.783
READ  |   P:003F9980--003F999F  |   4. |94 21 FF F0 7C 08 02 A6 ...      | 5402.785
READ  |   D:00000024--00000027  |   4. |60 00 00 00                      | 5402.786
READ  |   D:00000000--00000003  |   4. |48 00 00 02                      | 5402.786
READ  |   D:48000006--48000009  |   4. |38 00 00 00                      | 5402.787
emulator berr error
```

The **Data.LOG** command can be used to diagnose why errors like “emulator debug port fail”, “emulator berr error” etc. occurred.

The **Data.LOG** command logs all data accesses of the TRACE32 tool to the target that are initialized by the host software. This means for example that data accesses initialized by the software in the Power Debug Module can´t be logged. Examples for such accesses are the handling of the software breakpoints, accesses initialized by the **TERM** command, accesses initialized by the SNOOPer command.

Data.MSYS

M-SYSTEMS FLASHDISK support

Format:	**Data.MSYS** *<dllfile>* *<cmdline>*

Starts a flashdisk support utlity to program, view or format M-Systems flashdisks. The utitlity is supplied by M-Systems in form of a DLL module. The syntax of the command depends on the DLL module.

Format:	**Data.Out** <*address*> [**%**<*format*>] <*string*> [**/Repeat**]
<*format*>:	**Byte** | **Word** | **Long** | **Quad** | **TByte** | **TWord** **BE** | **LE**

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 …). The **Repeat** option repeats the input endless, e.g. for external measurements.

```
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
```

See also

■ Data.In ■ Data.Set ■ Data.Test ❏ ADDRESS.ACCESS()
❏ ADDRESS.DATA() ❏ ADDRESS.OFFSET() ❏ ADDRESS.ONCHIP() ❏ ADDRESS.PROGRAM()
❏ ADDRESS.SEGMENT() ❏ ADDRESS.SPACE() ❏ ADDRESS.WIDTH()

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

Format:	**Data.PATTERN** *<addressrange>* [**/***<option>*]
<option>:	**Verify | ComPare** **ByteCount | WordCount | LongCount** **ByteShift | WordShift | LongShift** **RANDOM | PRANDOM** **Byte | Word | Long | Quad | TByte | TWord**

This command fills the memory with a predefined pattern in the defined address range.

Verify	Verify the data by a following read operation.
ComPare	Compare the data against memory, don't write to memory.
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, Long, Quad, TByte, TWord	Define the memory access width used to write the pattern.

The default pattern is as follows:

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

This pattern allows easy testing of memory address translations (MMU):

```
Data.PATTERN EAD:0x0--0x7ffff          ; ill memory with pattern (physical
…                                      ; addr.)
Data.dump 0x0                          ; display logical view of memory
Data.dump 0x4000
…
Data.PATTERN 0x0--0x7fff /ComPare      ; compare memory against pattern
```

See also

- Data.dump
- Data.Set
- Data.Test

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

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.

DecimalU Decimal Unsigned.

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.

```
Data.Print 0x1000--0x10ff            ; display fixed range

Data.Print V.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
```

```
E::Data.Print %Decimal 0x23cc %Hex.Byte flags %Var flags %var flags+4 %Decia
wrCBAWRSHP     address  data            value                  symbol
wr            SD:0023CC 01              1                      \\MCC\mcc\.flags
wr    W       SD:0023CC 01              1                      \\MCC\mcc\.flags
wr            SD:0023CC 01              \\MCC\mcc\.flags[0] = 1 \\MCC\mcc\.flags
w             SD:0023D0 01              \\MCC\mcc\.flags[4] = 1 \\MCC\mcc\.flags+
wr            SD:002000 00              0                      \\MCC\mcc\.func2\.fstati
wr            SD:002000 00              00000000               \\MCC\mcc\.func2\.fstati
```

```
E::Data.Print %Decimal flags--(flags+2) %var flags--(flags+3) %Hex.Word fsta
  address  data          value                  symbol
SD:0023CC 01             1                      \\MCC\mcc\.flags
SD:0023CD 07             7                      \\MCC\mcc\.flags+1
SD:0023CE 01             1                      \\MCC\mcc\.flags+2
SD:0023CC 01             \\MCC\mcc\.flags[0] = 1 \\MCC\mcc\.flags
SD:0023CD 07             \\MCC\mcc\.flags[1] = 7 \\MCC\mcc\.flags+1
SD:0023CE 01             \\MCC\mcc\.flags[2] = 1 \\MCC\mcc\.flags+2
SD:0023CF 00             \\MCC\mcc\.flags[3] = 0 \\MCC\mcc\.flags+3
SD:002000 23 00          2300                   \\MCC\mcc\.func2\.fstatic
SD:002000 23             00100011               \\MCC\mcc\.func2\.fstatic
```

The scale area contains addresses and memory classes, as well as Flag and Breakpoint information. The status 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.dump
- ■ Data.View
- ❏ ADDRESS.ACCESS()
- ❏ ADDRESS.DATA()
- ❏ ADDRESS.OFFSET()
- ❏ ADDRESS.ONCHIP()
- ❏ ADDRESS.PROGRAM()
- ❏ ADDRESS.SEGMENT()
- ❏ ADDRESS.SPACE()
- ❏ ADDRESS.WIDTH()
- ❏ Data.AL.ERRORS()
- ❏ Data.Byte()
- ❏ Data.Float()
- ❏ Data.Long()
- ❏ Data.Long.BigEndian()
- ❏ Data.Long.LittleEndian()
- ❏ Data.LongLong()
- ❏ Data.LongLong.BigEndian()
- ❏ Data.LongLong.LittleEndian()
- ❏ Data.Quad()
- ❏ Data.Quad.BigEndian()
- ❏ Data.Quad.LittleEndian()
- ❏ Data.Short()
- ❏ Data.Short.BigEndian()
- ❏ Data.Short.LittleEndian()
- ❏ Data.SLong()
- ❏ Data.STRing()
- ❏ Data.SUM()

❏ Data.TByte()
❏ Data.Word()
❏ Data.Word.BigEndian()
❏ Data.Word.LittleEndian()
❏ Data.WSTRING()

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

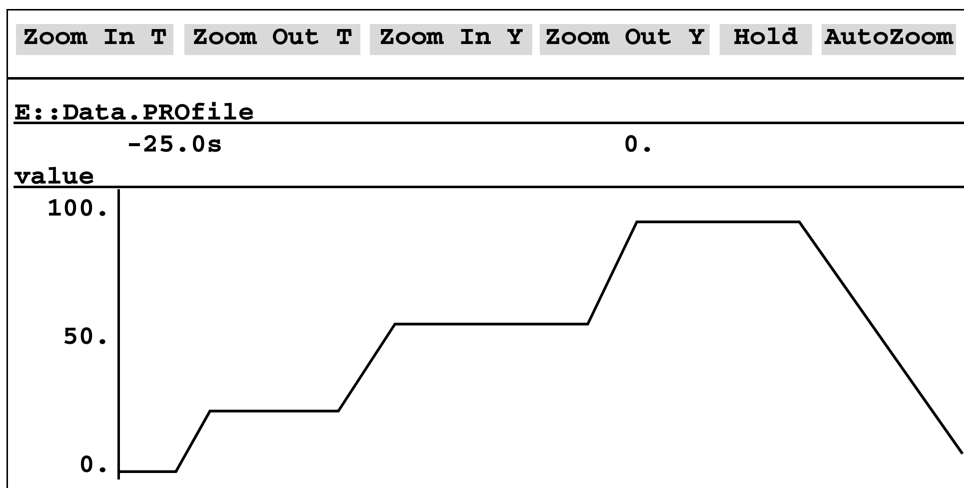
Data.PROfile

Graphic variable display

Format:	**Data.PROfile** [%<*width*>][<*address*> …] [<*gate*>] [<*scale*>]
<*width*>:	**Byte | Word | Long | Quad | TByte | TWord**
<*gate*>:	**0.1s | 1.0s | 10.0s**
<*scale*>:	**1. … 32768.**

The value of the memory location is displayed in graphical mode. The display is updated and shift every 100 ms or slower. An opened window may be zoomed by the function keys. An auto zooming feature displays the results always with the best vertical scaling. The auto zoom is switched off by supplying a scale factor, manual zoom or vertical scrolling. The scale factor must be a power of 2. Up to three variables may be shown in a single window. The function requires dual-port access if the variable should be displayed while the target is running.

```
d.pro %w e:BatteryVoltage                ; show battery voltage
```



Buttons	
Zoom In T	Zoom in vertical axis by factor 2.
Zoom Out T	Zoom out vertical axis by factor 2.

Zoom In Y	Zoom in horizontal axis by factor 2.
Zoom Out Y	Zoom out horizontal axis by factor 2.
Hold	Stop Updating.
AutoZoom	Auto zooming of the vertical axis.

Format:	**Data.PROGRAM** [<*address*>	<*addressrange*>] [<*file*>] [<*line*>]

This command creates a window for editing and assembling a short assembler program. Without a specified filename 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.Assemble
- Data.Set

Data.PROLOG

Sequence before start

The command **Data.PROLOG** allows to define a sequence of **Data.Set** commands, that are automatically executed by the TRACE32 software directly before the program execution is started by **Go** / **Step**.

The two commands **Data.EPILOG** and **Data.PROLOG** offer the option to switch off timers, communication interfaces etc. every time the program execution is stopped (**Data.EPILOG**) and to restart those components every time the program execution is started again (**Data.PROLOG**).

These commands are helpful if the processor doesn't automatically freeze those hardware components when the program execution is stopped.

See also

- Data.PROLOG.CONDition
- Data.PROLOG.OFF
- Data.PROLOG.ON
- Data.PROLOG.RESet
- Data.PROLOG.SEQuence
- Data.PROLOG.state
- Data.PROLOG.TARGET

'Release Information' in 'Release History'

Data.PROLOG.CONDition

Define PROLOG condition

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

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**.

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

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

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

See also

- Data.PROLOG

Data.PROLOG.OFF

Switch PROLOG off

Format:	**Data.PROLOG.OFF**

Switches the **Data.PROLOG** feature off.

See also

- Data.PROLOG

Data.PROLOG.ON

Switch PROLOG on

Format:	**Data.PROLOG.ON**

Switches the **Data.PROLOG** feature on.

See also

- Data.PROLOG

Data.PROLOG.RESet

Reset PROLOG

Format:	**Data.PROLOG.RESet**

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

See also

- Data.PROLOG

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

The command **Data.PROLOG.SEQuence** 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 — Write *<data>* to *<address>* directly before the program execution is started by **Go** / **Step**.

SETI — Write *<data>* to *<address>* the first time before the program execution is started by **Go** / **Step** after **Data.PROLOG.ON**.
Then *<data>* is incremented by *<increment>* every time the program execution is started by **Go** / **Step**.

SETS — Write the data that was saved with a previous GETS back to *<address>*. **Data.PROLOG.SEQuence SETS** can also use data saved with **Data.EPILOG.SEQuence GETS**.

GETS — Save the data at *<address>* directly before the program execution is started by **Go** / **Step**. Saved data can later be restored by **Data.PROLOG.SEQuence SETS** or **Data.EPILOG.SEQuence SETS**.
It is possible to store several data.

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

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

Data.PROLOG.SEQuence SETS 0x3faf60

Data.PROLOG.SEQuence GETS 0x3faf60
```

See also

■ Data.PROLOG

Data.PROLOG.state

PROLOG state display

Format: **Data.PROLOG.state**

```
B::Data.PROLOG
 prolog          CONDition
    OFF
 √  ON           SEQuence
                  SET 0x3faf30 %Word 0xa0a0a0

 count           TARGET
 17.
```

See also

■ Data.PROLOG

Data.PROLOG.TARGET

Define PROLOG target call

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

The command **Data.PROLOG.TARGET** 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.

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

See also

■ Data.PROLOG

Data.REF

Display current values

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**

Display 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.

Data.ReProgram

Assembler

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

This command 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.

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

The command **Data.ReRoute** 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.

```
; 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 Y.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 Y.SECRANGE(.text) malloc T32_malloc \t32mem
```

Data.SAVE.<format>

Save data in file with specified forma

Format:	**Data.SAVE.<format>** <*filename*> [<*addr-range*>] [**/**<*option*>]
<*filename*>:	**File name and (optional) path**
<*addr-range*>:	**Address range to be saved. It is possible to use access classes, e.g. A:**
<option>:	**Format-specific options.**

Saves the data from the specified address range in a file with the specified file format. Example:

```
; 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.ASAP2
- Data.SAVE.AsciiHex
- Data.SAVE.AsciiOct
- Data.SAVE.Binary
- Data.SAVE.IntelHex
- Data.SAVE.Omf
- Data.SAVE.Srecord

'Release Information' in 'Release History'

Data.SAVE.ASAP2

Save octal file

Format:	**Data.SAVE.ASAP2** <*filename*> [<*addressrange*>] [**/**<*option*>]

Save a file in ASAP2 file format.

See also

- Data.SAVE.<format>

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*> **BSPLIT** <*width*> <*offset*> **Byte | Word | Long | Quad**

Save a file in a simple ASCII file format.

Byte	Data is loaded bytewise. As default the data is loaded with word or long accesses where possible without overhead.
Word, Long, Quad	Data is loaded only with 16, 32 or 64-bit accesses.
BSPLIT	Loads only certain byte of the memory. The first argument defines the bus width in bytes, the second the offset of the byte being loaded. The option "BSPLIT 2 0" loads the lower byte of a 16-bit bus.

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.AscciOctP** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AscciOctA** *<filename>* [*<addressrange>*] [**/***<option>*] **Data.SAVE.AscciOctS** *<filename>* [*<addressrange>*] [**/***<option>*]
<option>:	**OFFSET** *<offset>*

Save a file in a simple ascii file format.

See also

■ Data.SAVE.<format>

'Data Access' in 'EPROM/FLASH Simulator'

Data.SAVE.Binary

Save binary file

Format:	**Data.SAVE.Binary** <*filename*> [<*addressrange*>] [**/**<*option*>]
<*option*>:	**ZIP : Saves the file**

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).

See also

- Data.LOAD.Binary
- Data.SAVE.<format>

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

Data.SAVE.IntelHex

Save INTEL-HEX file

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

Save 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.

See also

- Data.LOAD.IntelHex
- Data.SAVE.<format>

Data.SAVE.Omf — Save OMF file

Format: **Data.SAVE.Omf** <*filename*> <*addressrange*>

Saves memory in OMF file format. The command is implemented for the following formats: OMF-96.

See also

- Data.LOAD.Omf
- Data.SAVE.<format>

Data.SAVE.Srecord — Save S-Record file

Format: **Data.SAVE.S1record** <*filename*> <*addressrange*> **[/RECORDLEN]**
Data.SAVE.S2record <*filename*> <*addressrange*> **[/RECORDLEN]**
Data.SAVE.S3record <*filename*> <*addressrange*> **[/RECORDLEN]**

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.

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.

See also

- Data.SAVE.<format>

'Release Information' in 'Release History'

Format:	**Data.Set** [<*addressrange*>] **%**<*format*> <*string*> [**/**<*option*>]
<*format*>:	**Byte | Word | Long | Quad | TByte | TWord** **Float. [Ieee | IeeeDbl | IeeeeXt |** <*others*>] **BE | LE**
<*option*>:	**Verify | ComPare**

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.

Byte, Word, Long, Quad, TByte, TWord	Data size for integer or string constants.
Float	Data format for floating point constants.
BE, LE	Define display direction, big endian or little endian.
Verify	Verify the data by a following read operation.
ComPare	Compare the data against memory, don't write to memory.

```
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 V.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 the string starting
                                          ; at D:0x200
                                          ; Do not mix-up the command D.S
                                          ; (data.set) with the function
                                          ; D.S (data.string).
```

See also

- Data.Out
- Data.PATTERN
- Data.PROGRAM
- Data.Test

'Data Access' in 'EPROM/FLASH Simulator'
'Program and Data Memory' in 'ICE User's Guide'
'Registers' in 'Debugger Basics - Training'
'Registers' in 'Training FIRE Basics'
'Registers' in 'Training ICE Basics'

Data.STARTUP — Startup data sequence

The command **Data.STARTUP** allows to define a sequence of **Data.Set** commands, that are executed when the debugger or emulator is activated.

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.

```
; 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
```

Data.STARTUP.OFF — Switch startup sequence off

Format: **Data.STARTUP.OFF**

Switches the **Data.STARTUP** feature off.

Data.STARTUP.ON

Switch startup data sequence on

Format:	**Data.STARTUP.ON**

Switches the **Data.STARTUP** feature on.

Data.STARTUP.RESet

Reset startup data sequence

Format:	**Data.STARTUP.RESet**

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

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>*

The command **Data.STARTUP.SEQuence** defines a sequence of **Data.Set** commands, that are executed when the emulation system in activated.

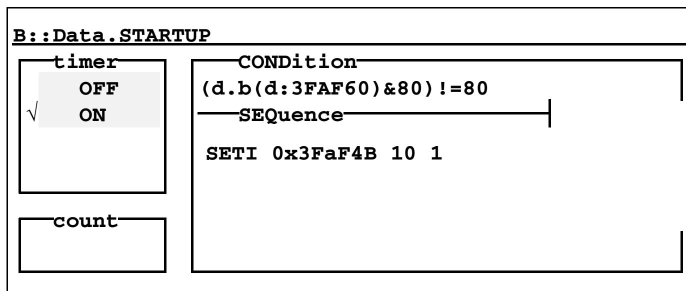
SET	Write *<data>* to *<address>* when emulation is activated.
SETI	Write *<data>* to *<address>*. Then *<data>* is incremented by *<increment>*.
SETS	Write the data that was saved with a previous GETS back to *<address>*.
GETS	Save the data at *<address>*.

```
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
```

Data.STARTUP.state

Startup data state display

Format:	**Data.STARTUP.state**



Data.STRING

ASCII display

Format:	**Data.STRING** [<*addressrange*>] [**/**<*option*>]
<*option*>:	**PC8**

Displays a string in the selected AREA. The character set is host specific. The option **PC8** converts graphic characters from IBM-PC character set to a host independent graphic character set. Linefeed characters are interpreted.

```
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.dump

Data.SUM

Memory checksum

Format:	**Data.SUM** <*range*> [**/**<*format*>]
<*format*>:	**Byte | Word | Long** **RotByte | RotWord | RotLong** **XorWord9 | RotWord9** **CRC16|CRC32|CRC** **Even | Odd**

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.

<table>
<tr><th>Checksum</th><th>Algorithms</th></tr>
<tr><td>Byte</td><td>add bytes</td></tr>
<tr><td>Word</td><td>add words</td></tr>
<tr><td>Long</td><td>add longs</td></tr>
<tr><td>RotByte</td><td>add byte and rotate byte left circular</td></tr>
<tr><td>RotWord</td><td>add word and rotate word left circular</td></tr>
<tr><td>RotLong</td><td>add long and rotate long left circular</td></tr>
<tr><td>CRC16</td><td>16-bit CRC algorithm (ZMODEM variant)</td></tr>
<tr><td>CRC32</td><td>32-bit CRC algorithm (PKZIP variant)</td></tr>
<tr><td>CRC
<params></td><td>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.</td></tr>
<tr><td>Even</td><td>add even bytes</td></tr>
</table>

Odd	add odd bytes
RotWord9	rotate words special (OS-9 compatible)
XorWord9	exor of all words, start-value = 0xffff (OS-9 compatible)

```
; 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

; make a 32-bit check sum,
; starting at 0 and stopping
; at but including 1FFFBH.
Data.SUM 0x0--0x1fffb /Byte
; place resulting checksum in memory
Data.Set 0x1fffc %Long Data.SUM()
```

See also

- Data.Test

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

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

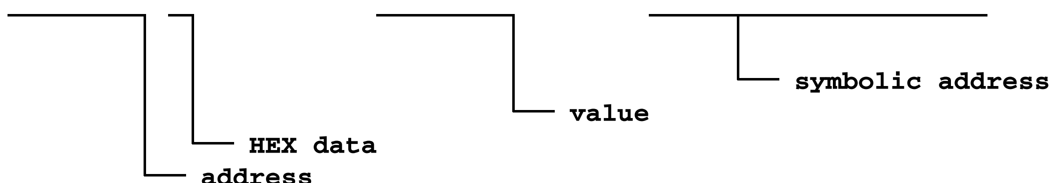
Displays an array without high-level information. If an address is given, it will specify the base address of an array of unlimited size. A range specifies an array of limited size.

DecimalU	Decimal Unsigned.
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.Lomg 0x0--0x7 %Decimal.Word 0x8
%Float.Ieee 0x0a
```

Sample window for displaying an array.

```
E::Data.TABle flags 10. %BINary.b 0 %Decimal.b 1 %Float.Ieee 2
 address    data          value            symbol
 00 (000.)|
D:0001004C|01            00000001         \\MCA\_flags
D:0001004D|01            1                \\MCA\_flags+1
D:0001004E|45 00 01 01   2.048062e+3      \\MCA\_flags+2
 01 (001.)
D:00010056|01            00000001         \\MCA\_flags+0A
D:00010057|00            0                \\MCA\_flags+0B
D:00010058|44 23 01 01   6.520156e+2      \\MCA\_flags+0C
 02 (002.)
D:00010060|00            00000000         \\MCA\_flags+14
D:00010061|00            0                \\MCA\_flags+15
D:00010062|02 10 01 01   1.057973e-37     \\MCA\_flags+16
```



The scale area contains addresses and memory classes, as well as Flag and Breakpoint information. The status line displays all current addresses, both in hexadecimal and symbolic format. By clicking on a data word, or by means of “Set”, a **Data.Set** command can be executed on the current address. By holding down the left mouse button the most important memory functions can be executed via softkeys. If the **Mark** option is on, the relevant bytes will be highlighted. For more information see **Data.dump-Window**.

See also

- Data.CHAIN
- Data.dump
- Data.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

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.UNTAGFunc

Format:	**Data.Test** [<*addressrange*>] [**/**<*option*>]
<*option*>:	**Toggle** **Prime** **RANDOM | PRANDOM** **Repeat** [<*count*>] **WriteOnly | ReadOnly** **NoBreak** **Byte | Word | Long | Quad**

If no address parameter is specified the total address space will be checked (until an error occurs). The error message is displayed in the message line. Memory test can be aborted at any time by pressing the STOP button.

Toggle (Default)	Memory contents are read in one block at a time, and inverted twice, thereby not altering memory contents.
Prime	The defined range is completely filled with a prime number sequence and is subsequently compared. Original memory contents are lost. This test detects address line failures or mirrored partions 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-to only.
ReadOnly	Memory read only.
NoBreak	Even in the case of memory error, the memory test does not abort.
Byte, Word, Long, Quad	Make 8, 16 or 32 bit memory access. Default is 8 bit access.

```
Data.Test 0x0--0x0ffff /Prime                   ; memory test prime number
                                                ; sequence

Data.Test 0x0--0x0ffff /Repeat                  ; memory test until memory
                                                ; error occurs or abort by
                                                ; means of keyboard

Data.Test 0x0--0x0ffff /Prime /Repeat 3         ; write prime-number
                                                ; sequence3 times only

Data.Test 0x0--0x0ffff /WriteOnly /Repeat       ; make only write cycles
                                                ; to test memory strobes
                                                ; by a scope

Data.Test 0x0--0x0ffff /ReadOnly /NoBreak       ; make only read cycles to
                                                ; test memory strobes by a
                                                ; scope
```

See also

- Data.dump
- Data.Out
- Data.PATTERN
- Data.Set
- Data.SUM
- SETUP.TIMEOUT

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

Data.TestList

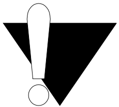
Test for memory type

Format: **Data.TestList** [<*address_range*>]

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

Data.TestList is non-destructive test to find out which memory type is at which address in your target. The smallest resolution is 4K. 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.

Data.TIMER

Periodically data sequence

The command **Data.TIMER** allows to define a sequence of **Data.Set** commands, that are executed periodically when the program execution is stopped.

This commands can be used e.g. to trigger a watchdog while the program execution is stopped.

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.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.

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

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

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

See also

- Data.TIMER

Data.TIMER.OFF — Switch timer off

Format:	**Data.TIMER.OFF**

Switches the **Data.TIMER** feature off.

See also

- Data.TIMER

Data.TIMER.ON — Switch timer on

Format:	**Data.TIMER.ON**

Switches the **Data.TIMER** feature on.

See also

- Data.TIMER

Data.TIMER.RESet — Reset timer

Format:	**Data.TIMER.RESet**

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

See also

- Data.TIMER

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

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

The command **Data.TIMER.SEQuence** 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.

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

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

Data.TIMER.SEQuence SETS 0x3faf60

Data.TIMER.SEQuence GETS 0x3faf60
```

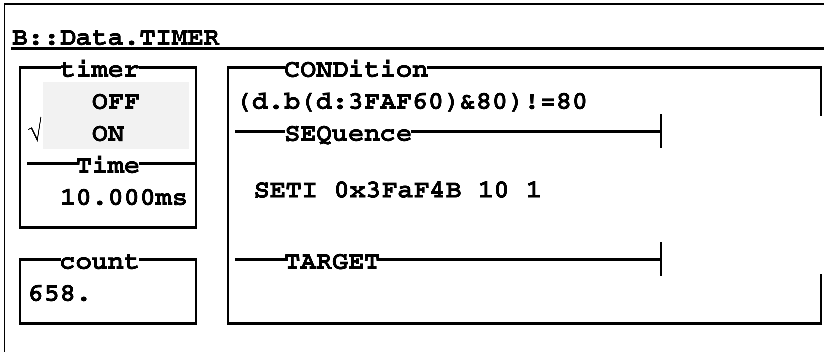
See also

■ Data.TIMER

Data.TIMER.state

Timer state display

Format: **Data.TIMER.state**



See also

- Data.TIMER

Data.TIMER.TARGET

Define timer target call

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

The command **Data.TIMER.TARGET** 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.

```
Data.TIMER.TARGET 0x3fa948--0x3faa07 0x1000--0x11ff
```

See also

- Data.TIMER

Data.TIMER.Time — Define period for timer

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

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

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

See also

- Data.TIMER

Data.UNTAGFunc — Remove code tags

Format:	**Data.UNTAGFunc**

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

See also

- Data.TAGFunc

Data.UPDATE — Target memory cache update

Format:	**Data.UPDATE**

Triggers and update of memory buffered by the debugger. Memory is only buffered when the address range is declared by **MAP.UpdateOnce** command.

Format: **Data.USRACCESS** *<code_range> <data_range>*

The command **Data.USRACCESS** introduces a special memory class named **USR:**. If a TRACE32 command accesses a memory range using this memory class
e.g. `Data.dump USR:1000` the TRACE32 software uses a user specified target program to perform this memory access. With the command **Data.USRACCESS** it is now possible to access very special memories e.g. flashdisc via the TRACE32 software.

<code_range> defines the address range for the user specified target program. *<data_range>* defines the address range used for the data of the target program.

```
; Load the user specified target program to <code_address> and
; introduce the memory class USR:

Data.LOAD.Binary flashdisc.bin
Data.USRACCESS 0x3fa948--0x3faa07 0x1000--0x11ff
Data.dump USR:1000
```

An example for the command **Data.USRACCESS** can be found on the TRACE32 software CD under ***\dos\demo\powerpc\etc\usraccess***.

See also

- Data.dump

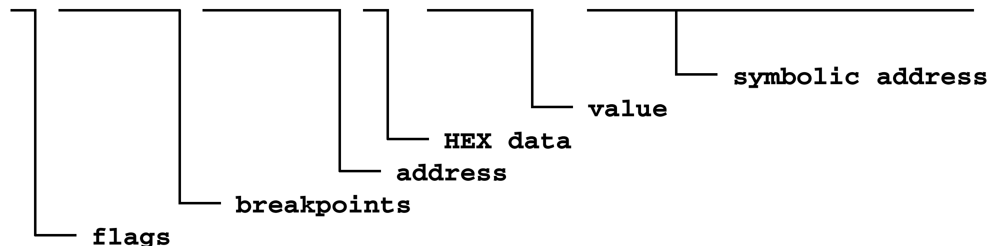
'Release Information' in 'Release History'

Format:	**Data.View** [**%**<*format*>] [<*address*> | <*range*>] [**/**<*option*> …]
<*format*>:	**Decimal.** [<*width*>] **DecimalU.** [<*width*>] **Hex.** [<*width*>] **Ascii** [<*width*>] **Binary** [<*width*>] **Float.** [**Ieee** | **IeeeDbl** | **IeeeeXt** | **IeeeMFFP** | …] **sYmbol.** [<*width*>] **Var** **DUMP**
<*width*>:	**Byte | Word | Long | Quad | TByte | TWord**
<*option*>:	**Mark** <*break*> **Flag** <*flag*> **Track** **CACHE**
<*flag*>:	**Read | Write | NoRead | NoWrite**
<*break*>:	**Program | Hll | Spot | Read | Write | Alpha | Beta | Charly | Delta | Echo**

If a single address is selected then this address does no more than define the windows' initial position. Scrolling makes all memory contents visible. When selecting an address range only the defined data range can be dumped. Range definition is useful whenever those addresses following on are read protected (e.g., in the case of I/O) or when more than one page is to be printed.

DecimalU	Decimal Unsigned.
Var	Display in HLL format.
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 status line displays all current addresses, both in hexadecimal and symbolic format. By clicking on a data word, or by means of "Set", a **Data.Set** command can be executed on the current address. By holding down the left mouse button the most important memory functions can be executed via softkeys. If the **Mark** option is on, the relevant bytes will be highlighted. For more information see **Data.dump-Window**.

See also

- ■ Data.CHAIN
- ■ Data.dump
- ■ Data.In
- ■ Data.Print
- ■ Data.TABle
- ❏ ADDRESS.ACCESS()
- ❏ ADDRESS.DATA()
- ❏ ADDRESS.OFFSET()
- ❏ ADDRESS.ONCHIP()
- ❏ ADDRESS.PROGRAM()
- ❏ ADDRESS.SEGMENT()
- ❏ ADDRESS.SPACE()
- ❏ ADDRESS.WIDTH()
- ❏ Data.AL.ERRORS()
- ❏ Data.Byte()
- ❏ Data.Float()
- ❏ Data.Long()
- ❏ Data.Long.BigEndian()
- ❏ Data.Long.LittleEndian()
- ❏ Data.LongLong()
- ❏ Data.LongLong.BigEndian()
- ❏ Data.LongLong.LittleEndian()
- ❏ Data.Quad()
- ❏ Data.Quad.BigEndian()
- ❏ Data.Quad.LittleEndian()
- ❏ Data.Short()
- ❏ Data.Short.BigEndian()
- ❏ Data.Short.LittleEndian()
- ❏ Data.SLong()
- ❏ Data.STRing()
- ❏ Data.SUM()
- ❏ Data.TByte()
- ❏ Data.Word()
- ❏ Data.Word.BigEndian()
- ❏ Data.Word.LittleEndian()
- ❏ Data.WSTRING()

'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'