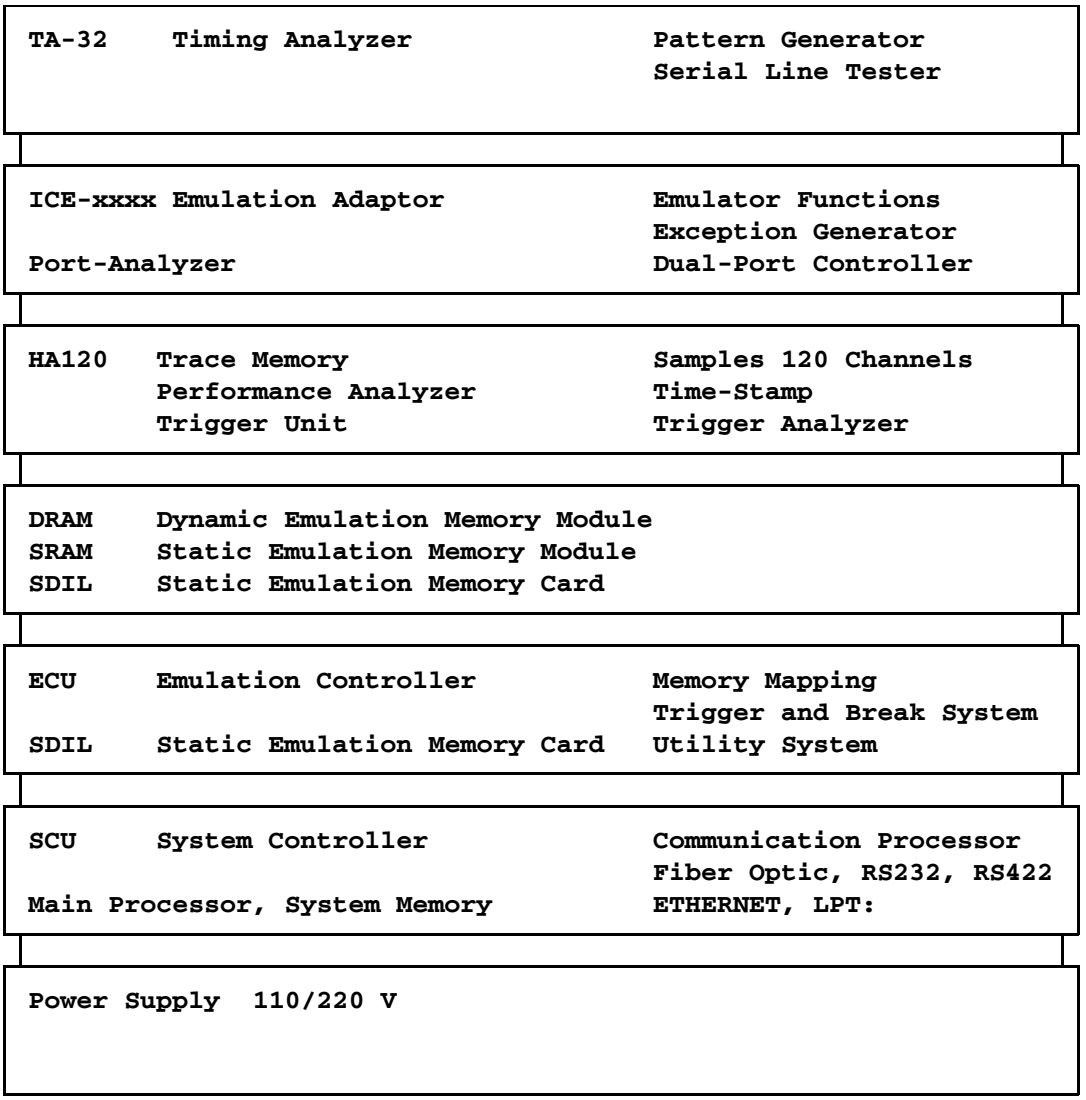
ICE User's Guide

Version 18-Nov-2020

Concept

TRACE32-ICE is a modular, universal microprocessor development system, working with a PC or a workstation. The modular concept allows memory size and analyzer performance to be customized to the developer's needs.

Modules



The SCU is the interface between the host system and the emulator system. A 32-bit CPU, in conjunction with a fast data transfer processor, guarantees an optimum performance. Memory can be expanded up to 56 MBytes by adding memory chips. Usually 4 MBytes of memory are sufficient for working with programs up to 500 KB code length. The system controller contains all software to control the emulator, and the database for symbols and HLL information. Source information is not kept in the memory of the SCU, but cached for faster data transfer. While booting, the whole monitor software is downloaded to the SCU. This takes about 20 seconds, when using a fiber optic interface (5 sec on ETHERNET).

SCU16	16-bit system controller
SCU32	32-bit system controller
RS-232 Interface	This interface is suited to all computers, which do not have fast serial or a parallel interfaces. The transfer rate can be up to 76 KBaud.
RS-422 Interface	This interface is of particular interest for users having UNIX hosts. Maximum transfer rate is MBaud.
Fiber Optic Interface	The Standard interface for PCs and PC compatibles. In addition to a fast transfer rate of 1 MBaud (2 MBaud in ETHERNET card), this interface guarantees excellent performance over long distances, fail-safe transmission, and galvanic isolation.
SCSI Interface	This interface can be used with all workstations. The connection is made by a converter-box from SCSI to Fiber Optic. The transfer rate (with 1 MBaud Fiber) is approximately 70 KByte/sec.
ETHERNET Interface	The ETHERNET interface provides high transfer rates, together with the ability to access the emulator from anywhere in the network. ETHERNET drivers are available for all hosts. TRACE32 provides an AUI connection, for Thin, Thick, Twistet Pair or Fiber-Optic ETHERNETs.
PAR	The parallel interface is used for low-cost applications or together with NOTEBOOK PCs.

ECU32 - Emulation Controller This module contains all emulation components which are not CPU specific. These include the memory manager (MAPPER), the trigger logic, and utility systems such as the universal counter and the pulse generator.

ECC8 - Emulation Compact Controller The compact version of Emulator controller for 8-Bit processors. Contains emulation control unit, emulation memory, state analyzer, performance analyzer and trigger systems.

The differences between ECC8 and ECU32 are:

	ECU32+HA120	**8**
RAM included	-	128K/512K
Bus Width	8..32 Bit	8 Bit
Mapper Classes	4	2
Trace channels	120	88
Timestamp Res.	25 ns	100 ns
Trigger Levels	8	4
Address Selectors	3	3
External Trigger Inputs	16	8
External Trigger Outputs	3	3
Trigger Flags	3	0
Trigger Marks	2	2
Word Data Selector	no	yes
Trigger Counter	3*48 Bit	2*16Bit(32)
Perf.Analyzer Levels	32	32
Perf.Analyzer Modes	address/level/flags	address
Async. Trigger Input	8	0
Trigger Counter	3	1
Event Trigger	yes	no
Glitch Detector	yes	no

SDIL - Static RAM This is a static emulation memory card which contains three sub-memories: emulation memory, break memory and flag memory. Memory expansion is possible in multiples of 128K or 512K. The SDIL is a board placed in the ECU32 or ECC8 modules.

SRAM - Static RAM This module allows large amounts of static memory. The size can be varied from 512K to 8 MBytes. Different speeds and fast access modes are also supported.

DRAM - Dynamic RAM This module allows large amounts of memory at reasonable costs. Memory expansion can be either 4 Myte or 16 MByte per module.

Every emulator system can be upgraded to 16 MByte emulation memory. Memory modules may be combined as follows:

SRAM + SRAM

SDIL + SDIL

SRAM + SDIL

DRAM + DRAM

DRAM + SRAM

DRAM + SDIL

If the DRAM 16+16 MByte is used together with SRAM or SDIL, the usable memory size is reduced to 8 MByte.

HA120 - High-Speed Analyzer	Analyzer for high-speed applications. Includes trace memory, trigger unit, timestamp unit and performance analyzer unit.
SA120 - Trace Memory	All bus cycles of the processor are traced and stored in the trace memory.
STU - Trigger Unit	The trigger unit is a programmable state machine for triggering, selective trace and stimuli generation. The trigger unit includes five trigger counters for complex trigger sequences.
TSU - Performance Analyzer	The performance analyzer marks all trace frames with a time stamp for analyzing program execution times and generation statistical information. The performance measurement tool analyzes the program performance in real time.

	SA120+STU+TSU	**HA120**
Cycle Time	150 ns	50 ns
Trace channels	120	120
Timestamp Res.	5 ns	25 ns
Trigger Levels	8..64	8
Address Selectors	8	3
External Trigger Outputs	6	3
Trigger Flags	7	3
Trigger Marks	4	2
Trigger Counter	5 * 48 Bit	3 * 48 Bit
Perf.Analyzer Levels	64	32
Longtime Trace	no	yes

Emulation Adapter

ICE - Emulation Adaptor	All CPU specific emulation components are contained in the emulation adaptors. These are, in addition to the emulation logic, the wait state and the exception generator.
Refresh Generator	If dynamic memory is used in the target system this memory must be refreshed during emulation by the refresh generator.
Exception Generator	It simulates and releases interrupts, DMA accesses, etc.
Wait-State Generator	Generation of additional wait states.
PA64 - Port Analyzer	The port analyzer samples up to 64 (256) peripheral lines in timing and state mode. It is used to trace or trigger on peripheral lines of the microcontrollers. The board is placed within the emulation adapter module.

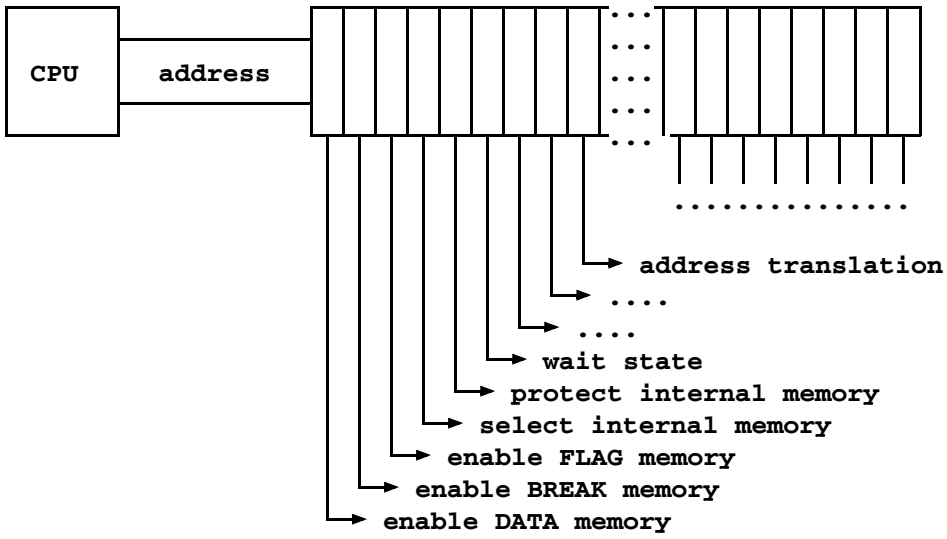
Timing Analyzer

TA32 - Timing Analyzer	This is the high-speed analyzer module. Sampling is done in transient mode, with long time trace possibilities. The resolution is 5 ns. Every channel group can work synchronous or asynchronous. All sampling is correlated to the state and the port analyzer.
PATT16 - Pattern Generator	The pattern generator will produce stimuli signals on 16 channels. The resolution is 100 ns whereas the max. cycle time is bigger than 1 min.
SLT - Serial Line Tester	The serial line tester translates 2 serial channels to a parallel data stream. The timing analyzer can sample and trigger on this data. The line tester can generate cyclic data burst together with the pattern generator.

STG - Stimuli Generator	The stimuli generator includes 64 digital in/out pins for testing digital system. Additional 8 analog in/out pins are available. A line tester indicates floating lines and search for short-circuits within a running target system.
BDM - Background Debug Module	Universal BDM debug and JTAG controller
ESI - EPROM Simulator	ROM Monitor Inteface/EPROM Simulator supports 8- and 16-Bit EPROMs up to 4 MBit

Mapper

The mapper provides the system with address dependent signals. One key function of the mapper is to define the memory allocation.

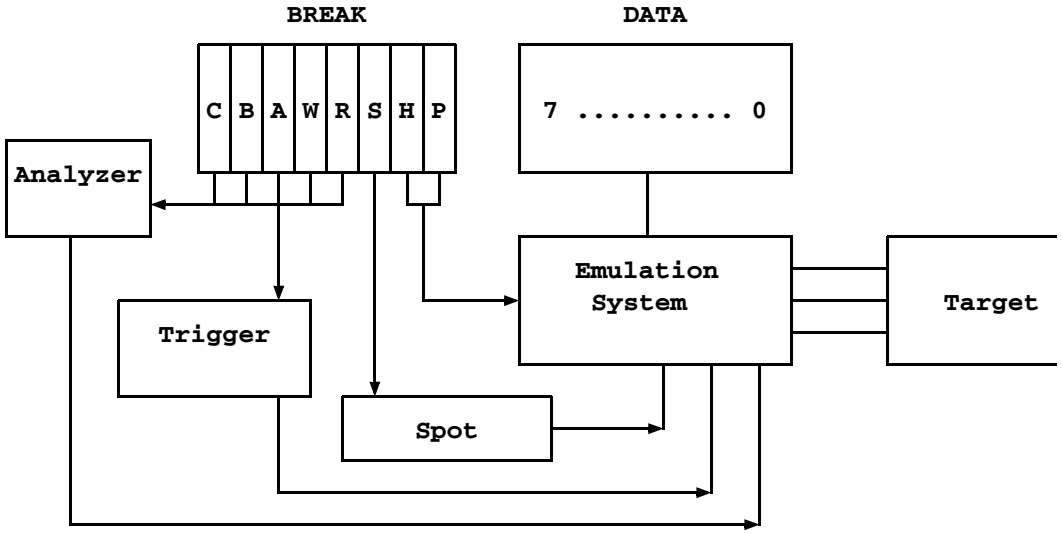


Emulation Memory

During the development phase, emulation memory can be used as a program and data storage. Emulation memory can be made 'write protected' to simulate EPROMs.

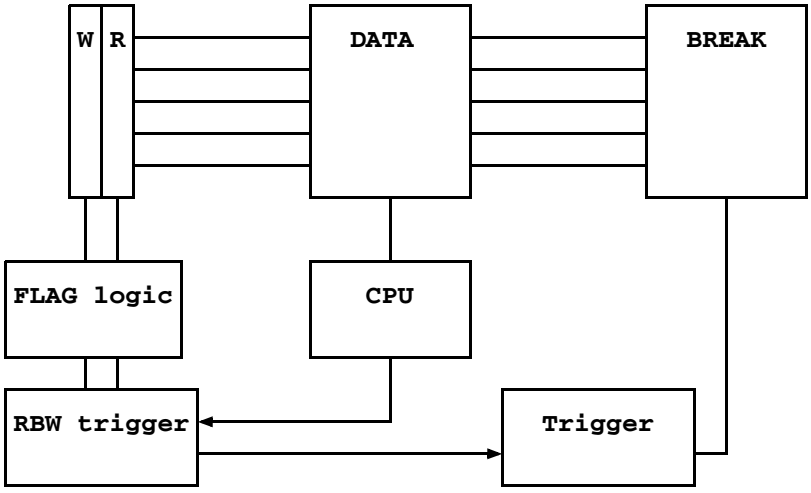
Break Memory

A particular feature of the TRACE32 system is the break memory. It allows the labelling of each address with 8 different markers. Each marker can be used as a breakpoint or as an address qualifier for the logic analyzer.



FLAG Memory

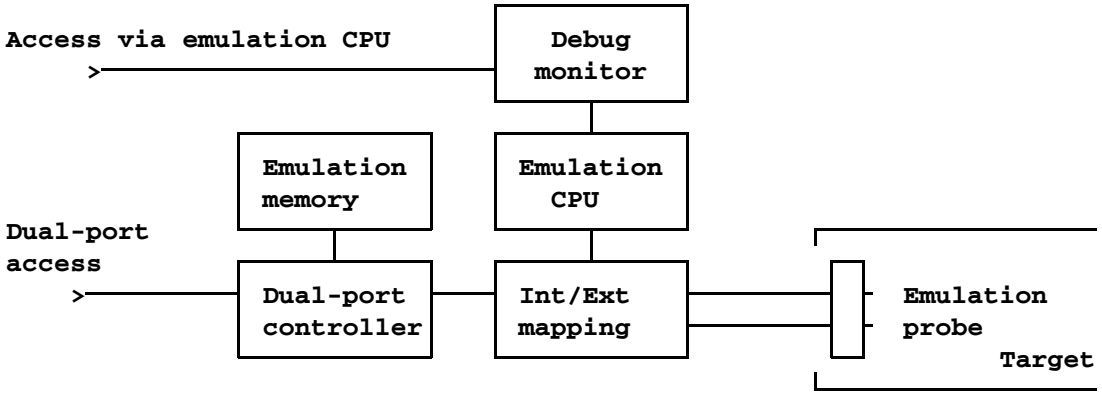
The flag memory is used for marking memory cells to which a read or write access takes place.



FLAG system and READ-BEFORE-WRITE trigger

Dual-Port Controller

All types of memory can be accessed during real-time emulation. A specially designed controller facilitates the dual-port access and ensures the refreshing of dynamic emulation memory.

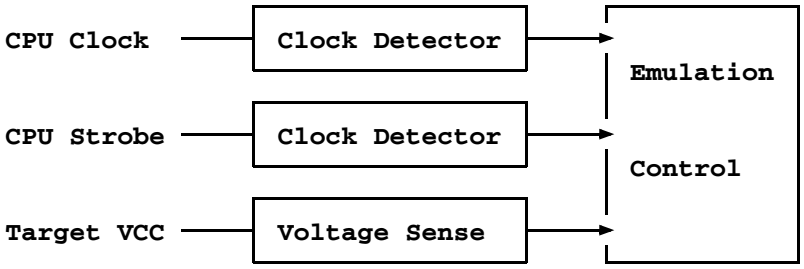


Emulation System

The emulation system offers all functions for debugging user programs. It supports single steps in assembler and high-level languages, as well as the evaluation of program breakpoints. In addition, program execution in single CPU cycle mode is possible.

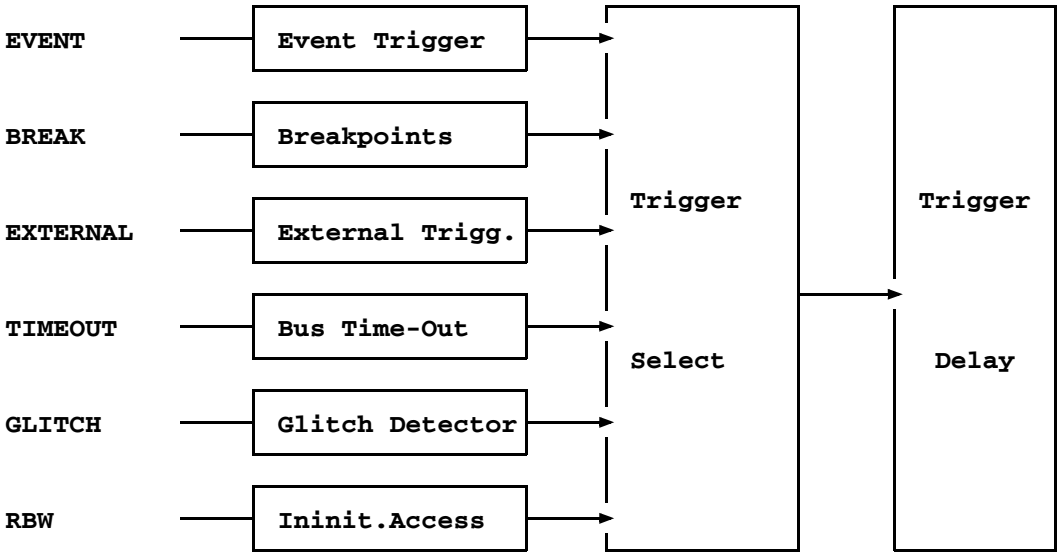
Target System Monitor

The clock of the target system and the supply voltage is monitored by the emulator. If any error occurs, the operating mode of the emulator will be adjusted automatically.



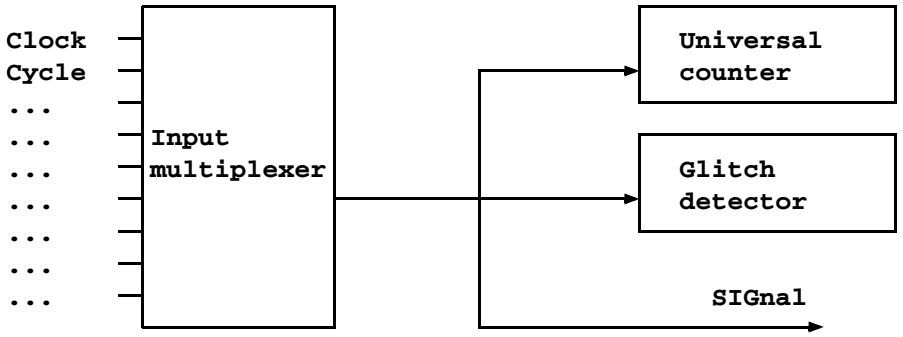
Trigger System

The trigger system can combine all trigger sources and therefore generate the emulator break signals.



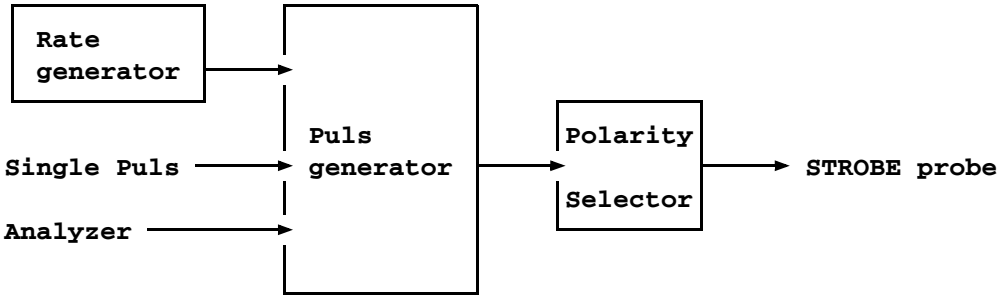
Counter

The universal counter is an utility system for measuring clock frequencies or events.



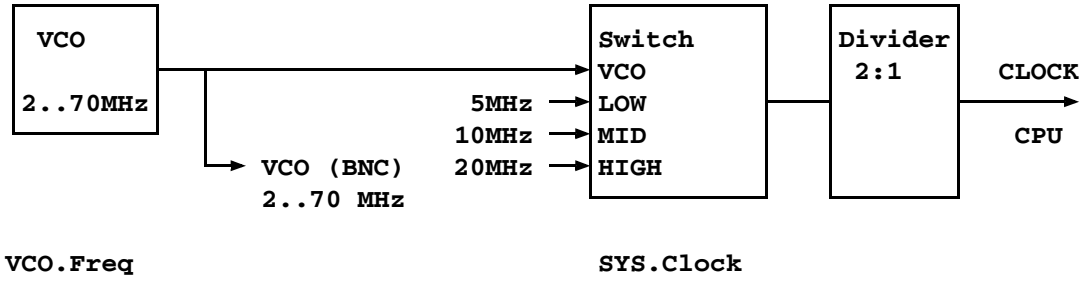
Pulse Generator

The pulse generator is able to supply the target system with short single or periodic pulses.



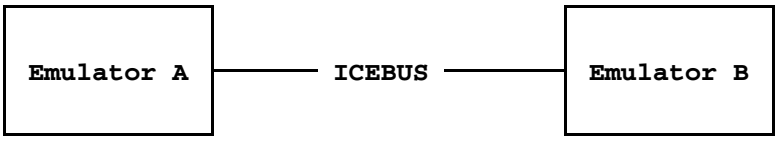
VCO

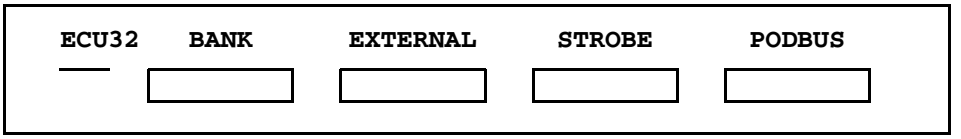
The VCO (voltage controlled oscillator) can be used to generate a system clock, when the internal clock is selected. In addition, however, it can be fed directly into the target system via a BNC socket connector at the back of the ECU32 or ECC8 chassis.



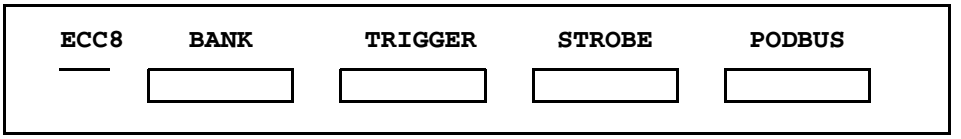
Synchronization

The TRACE32-ICE System allows to synchronize several emulators. In this case the program start and break are executed simultaneously.





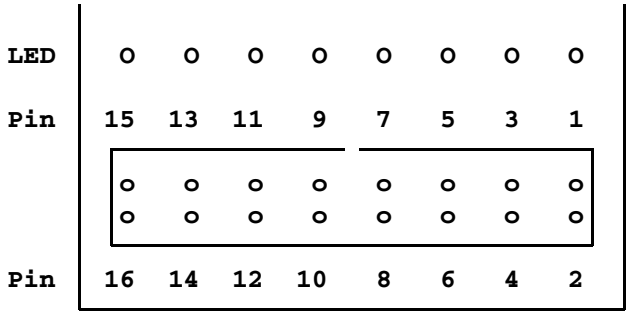
ECU32 Emulator module front view



ECC8 Emulator module front view

BANK	Bank input. In the case of 8-bit CPUs, addresses from an external MMU can be integrated. In the case of 16 or 32-bit CPUs, these lines can be used at your discretion.
EXTERNAL	External trigger inputs.
TRIGGER	External analyzer inputs.
STROBE	Trigger and pulse generator outputs.
PODBUS	Connection for PODBUS devices.

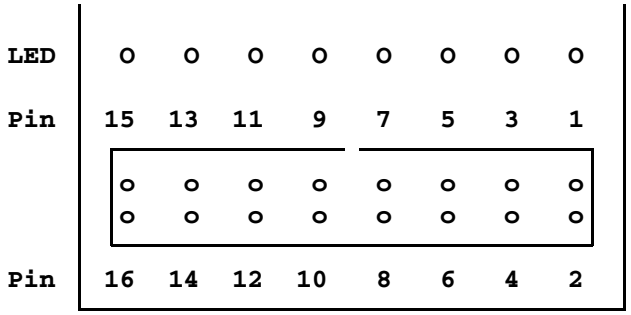
Input Probe Assignments



Pin 1	Line 0	Data0
Pin 3	Line 1	Data1
Pin 5	Line 2	Data2
Pin 7	Line 3	Data3
Pin 9	Line 4	Data4
Pin 11	Line 5	Data5
Pin 13	Line 6	Data6
Pin 15	Line 7	Data7
Pin 2,4,6,8,10,12,14,16	Ground	

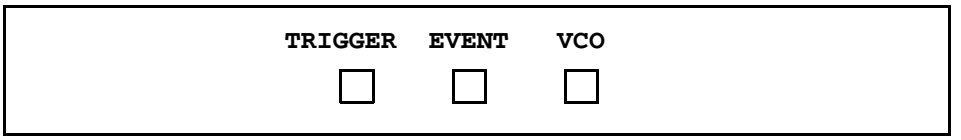
Input probes are used for the TRIGGER inputs, the BANK input and the EXTERNAL input. The threshold level of the probe is 1.4 V.

STROBE Probe Assignments

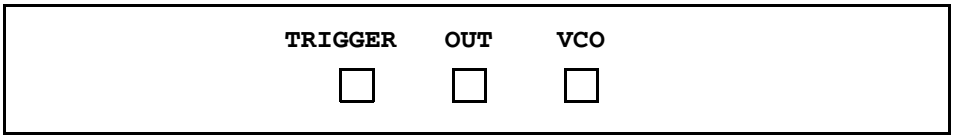


		ECU32	ECC8
Pin 1	Line 0	EVENT	OUT.C
Pin 3	Line 1	Trigger address	OUT.D
Pin 5	Line 2	RUN- (foreground)	RUN-
Pin 7	Line 3	TRIGGER	TRIGGER
Pin 9	Line 4	SIGnal	CharlyBreak
Pin 11	Line 5	RUNCYCLE	RUNCYCLE
Pin 13	Line 6	PULS2	PULS2
Pin 15	Line 7	PULS	PULS
Pin 2,4,6,8,10,12,14,16	Ground		

EVENT	This signal corresponds to the output signal of the EVENT selector. The pulse width is approximately 50 ns.
Trigger Address	This is the output signal of the address selector. The pulse width is approximately 30 ns.
RUN-	Signal is active low, whenever a foreground program is running.
TRIGGER	The trigger signal is active high if a trigger is activated during real-time emulation.
SIGnal	This signal refers to the input selector of the universal counter.
RUNCYCLE	This output generates one pulse per CPU cycle, as long as the emulator is executing a foreground program.
PULSe2	Can be used as a system reset signal for the target.
PULSe	The output signal of the pulse generator.
OUT.C	Output Trigger Unit.
OUT.D	Output Trigger Unit.
CharlyBreak	Breakpoint Signal.



ECU32 Emulator module rear view



ECC8 Emulator module rear view

TRIGGER Trigger signal. Active high as long as trigger is active and real-time emulation is on.

EVENT Event output. Is determined by the event trigger.

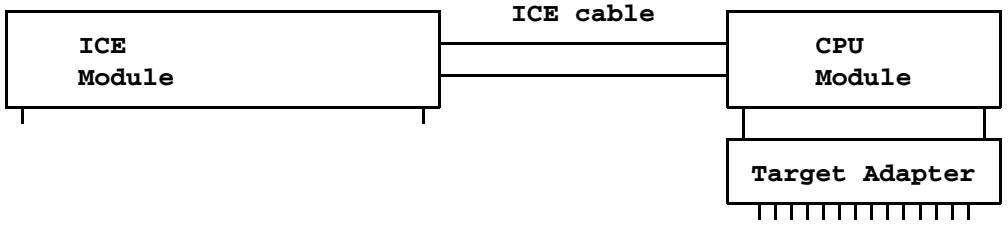
OUT Output of the trigger unit.

VCO VCO output. 1.5 … 50 (70) MHz. All outputs have an impedance of 50 Ω and a 5 V level. For longer lines it is imperative that a termination resistance is attached (50 … 100 Ω).

Modularity

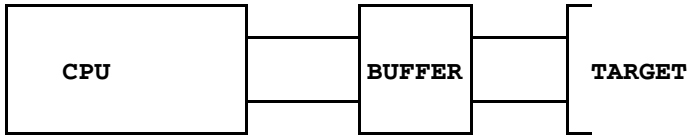
The TRACE32 is a modular emulation system that uses multi-step modular technology to support as many CPU and socket types as possible. Designed as an open system, it offers connections to many workstations and host operations systems and most compiler systems on the market. On the target side it offers standard interface connections to support 3rd-party tools from many manufacturers.

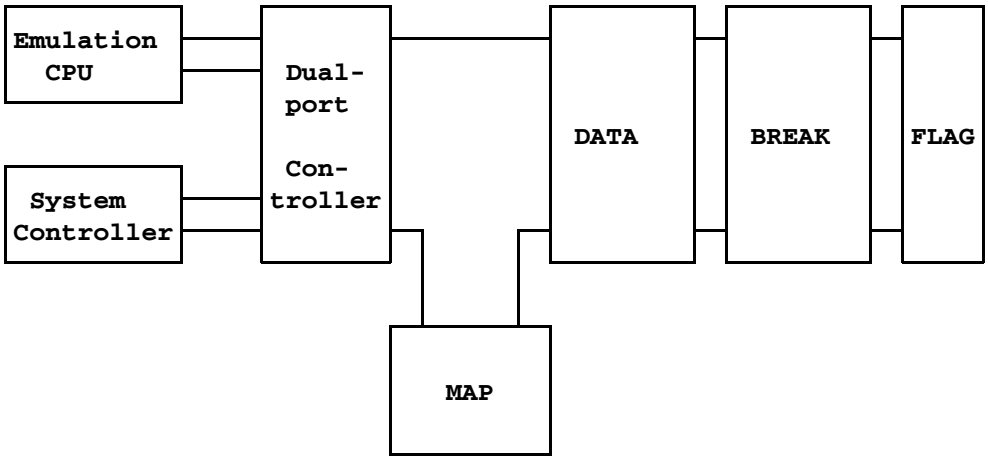
The basic system is designed for 8, 16 and 32-bit CPUs (ECU32) or 8-bit CPUs only (ECC8). The ICE adapter supports as many chips within the same family as possible. The emulation module is the specific part for the CPU, while the target adapter is needed to support different socket connectors (PLCC, DIL, etc.).



Buffered Probes

The probe lines are isolated. To bring-up the emulator system the clock signal and the target power must be valid only. This concept allows debugging target systems with hardware failures. The debug monitor is allocated in a separate memory and needs no space within the target memory. Target software is not to be changed for TRACE32.





TRACE32 uses a memory-based breakpoint technology. In addition to the program or data memory, two parallel memory systems are supplied:

DATA memory

BREAK Memory

FLAG Memory

The BREAK memory is accessible parallel to the program and data memory, and therefore offers an efficient breakpoint structure. Breakpoints may be set on the basis of an address or address range. No indication of the CPU bank access signals is necessary when emulating 16 or 32-bit target systems. A breakpoint is valid if the memory cell is accessed, not just if the address send out by the CPU is valid. The breaksystem automatically reacts to accesses on mirrored addresses and the breakpoint system can easily support banked target system or split memory structures.

The FLAG memory consists of 2 flags. The READFLAG is set as soon as memory data read or program access is occurred. The WRITEFLAG reacts to all data write operations.

Dual-Port Technology

All memories can be accessed by the CPU and the system controller. Different access modes are implemented inside the ICE module. The dual-port technology enables fast down-load of the program and data. On a running system static variables and system setups may be displayed and manipulated. Setting of breakpoints and analyzing flag memory is possible, while the target program is running in real time.

State Display

The state line of the emulator is divided into 4 groups:

Current address — It contains the access mode and the value of the program counter. By clicking with the mouse the **Register** command can be executed.

Flags — Letters are used to give a short form indication of the most important states of the emulator system:

G	Go	Forground program executing
g	go	Program executing not real-time
s	step	Program single stepping
B	Back	Background program executing
C	Cont	OS Awareness continues kernel
H	Halt	CPU inactive for more than 20 us
T	Timeout	Time-out for memory access
S	Spot	Spot breakpoint active
R	Read	Read breakpoint response
W	Write	Write breakpoint response
I	Int	Interrupt request encountered
D	Dma	DMA request encountered
A	Ackn.	Bus is tristate (DMA)
X		Exception control active
!	Reset	Target reset is active

Debug Mode — Current debug mode. By clicking with the mouse, the **Mode** command can be executed.

Emulation Mode — Current emulation mode. By clicking with the mouse, the **SYStem** command can be executed.

Memory Oriented Softkeys

In the case of most memory related commands (eg., **Data.dump** or **Data.List**), an extra softkey list can be accessed by clicking, depressing and when the left mouse button is held down. Softkeys enable several functions, whereby the reference address is defined by the position of the mouse pointer.

```
SD:00202C   \\MCC\mcc\.vdouble+2
```

```
E68::w.d.v 2028 /m rf
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
```

```
E68::
F:wr--------, H:99, D:+153/-103, O:231, B:10011001, A:'¹'
 display   break.s   [view]   [break.s]   [break]   [go.to]
```

- [go.to] — run to this line
- [break] — set temp. breakpoint
- [break.s] — set fixed breakpoint
- [view] — display memory on byte level
- break.s — set or clear breakpoints
- display — display memory

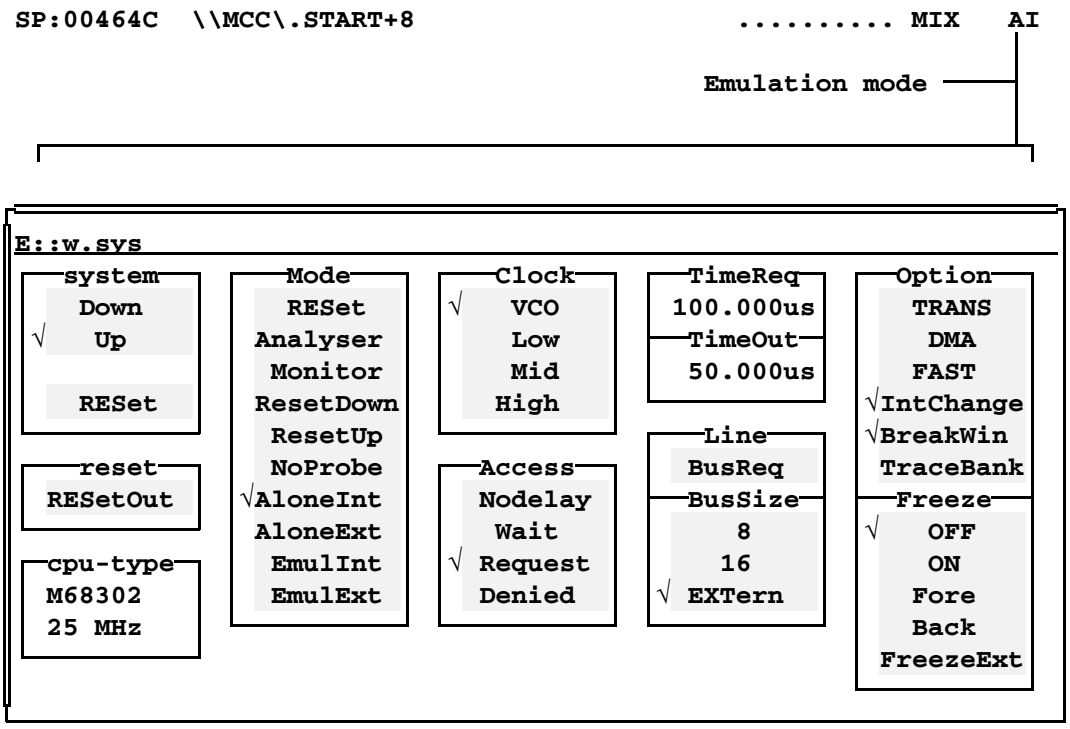
Activation

The emulation system must be started before any CPU emulation function regarding is available. The emulator can run in stand-alone mode without any target system. Usually the internal clock is selected by this mode. The internal clock may be supplied by the **VCO** or 3 fixed frequencies. When running with a target system, the internal or external (target) clock can be selected by the command **SYStem.Mode**.

ResetDown	Target power off
ResetUp	Target power on, but emulation system is stopped
AloneInt	Emulation system is activated, target strobes are blocked
AloneExt	Target clock is selected, target strobes are blocked
EmulInt	Internal clock is selected, full emulation capabilities are available
EmulExt	Target clock is selected, full emulation capabilities are available

```
sys.a request           ; select dual-port mode
sys.c vco               ; select VCO
vco.c 10.               ; define VCO frequency
sys.m ai                ; run in stand-alone mode

sys.m ee                ; run with target clock
```

The emulation mode is displayed within the **state line**. By clicking to this field, the SYS window will be pulled-down.



SYStem.state	Displays system state
SYStem.RESet	Initializes emulation system
SYStem.Up	Activates emulation system
SYStem.Down	Deactivates emulation system
SYStem.Mode	Selects emulation mode

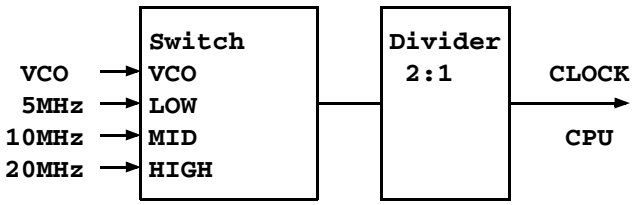
System Errors

After starting-up the emulation system, an internal system test is executed to verify the correct operation of the system. If the initialization fails, some of the following error messages may appear:

CPU clock fail	No clock signal ready, the clock frequency is too low, or the clock oscillator is not working correctly.
emulator cycle fail	The emulation CPU is not generating any bus strobes (DS, ALE, ...)
emulator monitor fail	The self-test function of the emulation monitor is not passed. This may result from a too high clock frequency.
emulator monitor trap	Internal trap detected. Fatal error.
emulator dual-port fail	The communication between the system controller and the emulation CPU is not possible.
emulator debug port fail	The BDM debug port of the CPU is not responding correctly.
refresh error	The dual-port rate is too low for refreshing the dynamic emulation memory. Use static memory or select another dual-port access mode.

Clock Select

When running with internal clock, one variable and three fixed clock signals can be used.



SYStem.Clock	Select clock source for internal clock

Time-Out

The emulator system supports different time-out circuits, which prevent hang-ups of the emulation system in the event of errors. The bus time-out system terminates the bus cycles automatically after a specific time. Bus cycle termination by the emulator is displayed on the state line (T).

The max. response time for internal dual-port accesses is limited to 10 ms. If a dual-port access is not possible within this time, the emulation system will be shut down. Emulation probes with BDM interface will possibly hang-up, if bus cycles are not terminated by the target.

For additional information on the time-out function, refer to the **emulation probe manual**.

SYS.TimeOut	Set bus time-out limit
SYS.TimeReq	Set dual-port time-out value
SYS.TimeDebug	Set time-out for BDM interface

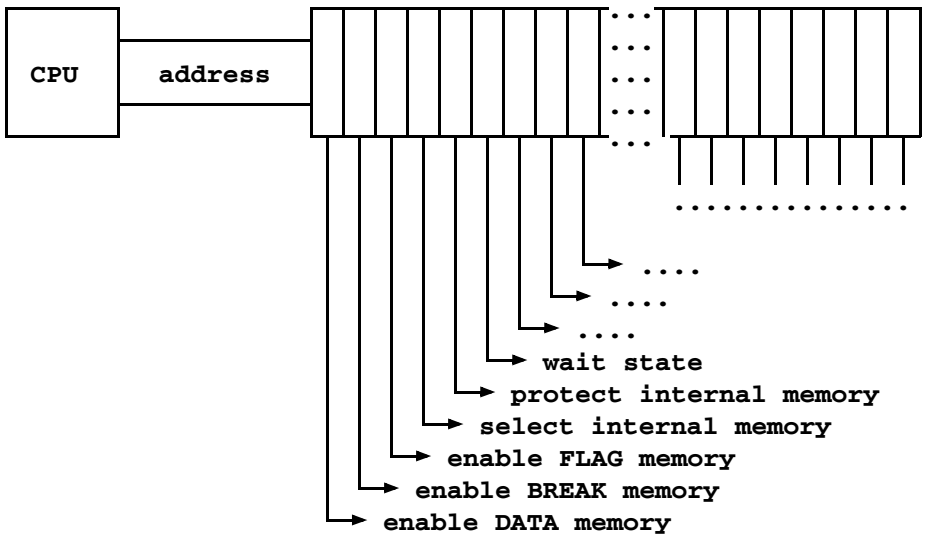
Special Setup

Some emulation probes include additional functions. For more information refer to the **emulation probe manual**.

SYS.Access	Select dual-port access mode
SYS.RESetOut	Initialize target system
SYS.Line	Control emulator line options
SYS.Option	Set options for emulation
SYS.MonFile	Load monitor extensions
SYS.BankFile	Load driver for banked target systems

Basic Function

The mapper is the basic system for controlling all signals depending on the address of the emulation CPU. The basic functions are to set up the overlay memory and protect certain address areas against access. An additional function of the mapper is to generate stimulation signals like acknowledge signals. The mapper consists of a 16K * 52 bit high-speed memory. The address input is connected to the CPU address bus, while the data outputs control the system or enable the chip selects of the emulation memory. All mapping functions are related to an address range and a storage class. Up to 4 different storage classes are possible (e.g. DATA, PROGRAM).



Fine Mapper

All mapping is done in 4 KByte blocks. However some functions can be mapped byte by byte. An additional mapping system (fine mapper) supports this function for internal/external mapping and write protection.

Pre-Mapper

Emulation probes, which support CPUs with more than 24 address lines, include a pre-mapper. This circuit translates the upper address lines to the main mapper. The user must define 16 different 'workbenches' for mapping. Outside of these workbenches breakpoints can be set on 1 MByte borders only.

Mapping is done in 3 steps.

First the mapper mode is selected by the command **MAP.Mode**. Some emulation probes need the **FAST** mode to support the highest clock frequency. 8- and 16-bit probes usually run in **SLOW** mode. For banked target systems the slow mode must be selected. The ECC8 supports the **FAST** mode with the **SRAM** module only.

```
map.mode slow

map.mode fast
```

Emulation probes which support 32-bit address range use a pre-mapper technology. The mapper supports 16 active mapper areas ('workbenches'). Within these workbenches all mapping may be done either in 4-K blocks or byte by byte. Outside this area all address definitions must end on 1-MByte block boarders.

```
map.pre 0x0fff00000--0x0ffffffff
```

The next step is to define the memory segmentation and overlapping by the **MAP.SPlit** and **MAP.MIRROR** commands.

```
map.split  d:0x0--0x0ffff      ; use different memory for program and
p:0x0                          ; data area

map.mirror d:0x0--0x7fff       ; address A15 is not decoded
d:0x8000
```

After the definition all types of memory (DATA, BREAK, FLAG) can be mapped. The DATA memory can store programs and data from the user program. To simulate EPROM structures, a write-protect definition can be made for this type of memory. The allocation of memory and the control of the data bus (internal/external mapping) are two different functions. To replace a target memory by the emulator memory, the **MAP.Data** as well as the **MAP.Intern** function must be activated.

```
map.data   0x0--0x0ffff

map.intern 0x0--0x0ffff
```

Parallel to the DATA memory, the BREAK and flag memory can be mapped in a similar way.

```
map.break 0x0--0x0ffff

map.flag  0x0--0x0ffff
```

Usually the break memory will not be mapped by the user. It is allocated dynamically when setting breakpoints or loading HLL programs.

Attributes for memory protection and wait states can be set additionally:

```
map.protect 0x0--0x0ffff

map.wait 3. 0x0--0x0ffff
```

MAP.RESet	Reset Mapper
MAP.NEW	Reset memory allocation
MAP.Mode	Set up mapper speed
MAP.MIRROR	Set memory mirroring
MAP.SPlit	Splits memory for different storage classes
MAP.PRE	List and set up workbenches
MAP.state	Displays free and used memory
MAP.List	Displays memory allocation
MAP.DEFault	Maps all available memory
MAP.Ram	Map DATA, BREAK and FLAG memory
MAP.NoRam	Un-map all memory types
MAP.Data	Map data and program memory
MAP.NoData	Un-map data and program memory
MAP.Break	Map breakpoint memory
MAP.NoBreak	Un-map breakpoint memory
MAP.Flag	Map flag memory
MAP.NoFlag	Un-map flag memory
MAP.Protect	Map write protection
MAP.NoProtect	Remove write protection
MAP.Ack	Map acknowledge signal generation
MAP.NoAck	Un-map acknowledge signal generation
MAP.Wait	Map additional wait states

Free memory is displayed by the **MAP.state** window:

E68::w.map

KByte	static 70 ns	dynamic 60 ns
Dataram		
total	512	4096
used	0	0
free	512	4096
Flagram		
total	512	4096
used	0	0
free	512	4096
Breakram		
total	512	4096
used	0	0
free	512	4096

The **MAP.List** window displays the allocation of the memory and the mapping of wait states and other CPU specific signals.

E68::w.map.1

Address		UP Flag	UP Rams	UP Wait	UD Flag	UD Rams	UD Wait	SP Flag	SP Rams	SP Wait
0--	1FFF	I P	FDB s	4	I P	FDB s	4	I P	FDB s	4
2000--	3FFF	I	FDB s	4	I	FDB s	4	I	FDB s	4
4000--	7FFF	I	FDB s	2	I	FDB s	2	I	FDB s	2
8000--	EFFF		FDB s	2		FDB s	2		FDB s	2
F000--	FFFE		FDB s			FDB s			FDB s	
FFFF--	FFFF	I	FDB s	30	I	FDB s	30	I	FDB s	30
10000--	FEFFFF									
FF0000--	FFFFFF		FDB s			FDB s			FDB s	

Explanation of the abbreviations within the columns from left to right:

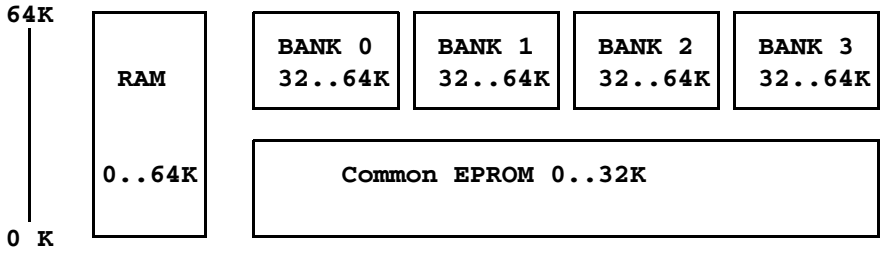
A	**A**cknowledge signal will be generated
I	DATA RAM is mapped **I**nternal
P	memory-write **P**rotection is turned on
B	**B**anking area
F	**F**lag RAM is mapped
D	**D**ata RAM is mapped
B	**B**reak RAM is mapped
s	**s**tatic RAM is mapped
d	**d**ynamic RAM is mapped
1..250	number of wait states

Banking

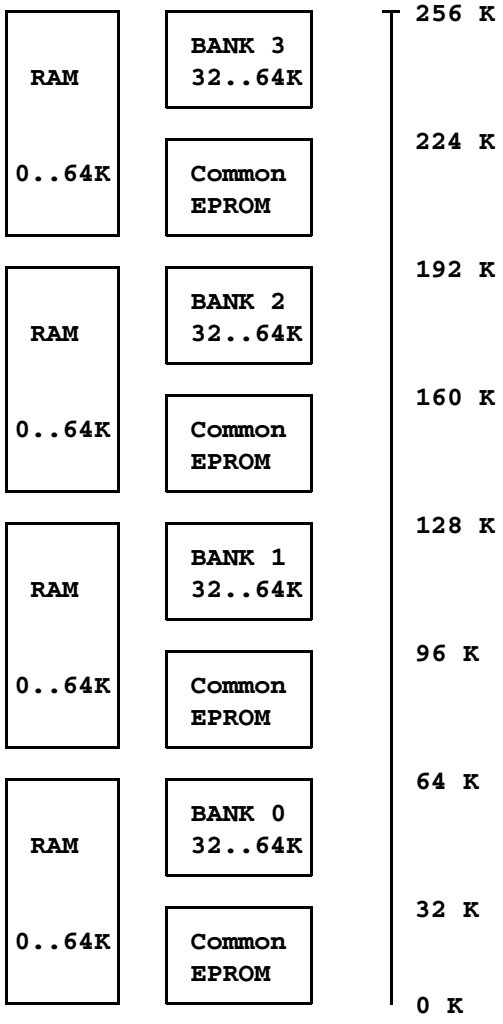
Memory extensions on 8 or 16-bit systems supported by additional logic on the target system are called memory banks. The support of complex banking system is possible by the flexible mapper structure. The internal emulator logic mimics all functions of the target.

Example

A target system uses a non-banked RAM and an EPROM (256 KByte). Half of the EPROM is non-banked and the other half is used for the banked programs. The upper address lines of the EPROM are generated by 2 I/O pins of the microcontroller. The memory structure is like:



By feeding back the upper address lines to the emulator (BANK probe), the memory layout is like this:



For rebuilding the same structure in the emulator memory, the following command setup must be used:

```
map.reset                                ; init mapper
map.split   d:0x0--0x0ffffff             ; separate data and program memory
map.mirror  d:0x0--0x0ffff 0x10000       ; mirror bank 0 with bank 1
map.mirror  d:0x0--0x0ffff 0x20000
map.mirror  d:0x0--0x0ffff 0x30000
map.mirror  p:0x0--0x7fff  0x10000       ; mirror common program area
map.mirror  p:0x0--0x7fff  0x20000
map.mirror  p:0x0--0x7fff  0x30000

map.ram     d:0x0--0x0ffff               ; set up data ram
map.ram     p:0x0--0x7fff                ; set up common program area
map.ram     p:0x8000--0x0ffff            ; map bank 0
map.ram     p:0x18000--0x1ffff           ; map bank 1
map.ram     p:0x28000--0x2ffff           ; map bank 2
map.ram     p:0x38000--0x3ffff           ; map bank 3
```

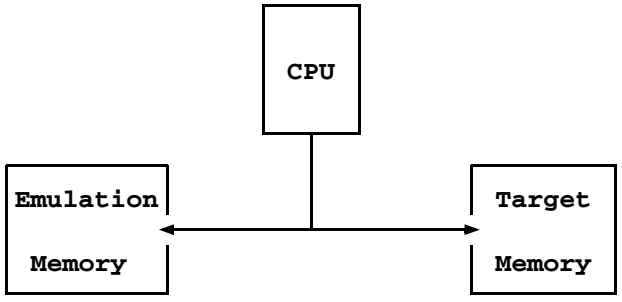
By mirroring the emulator memory, a breakpoint set to the common EPROM on bank 0 is automatically set on bank 1 to 3.

The emulation monitor can access the different bank areas by changing the level I/O pins on the microcontroller chip. This is done by a program module named bank driver, which is loaded by the activation of the emulator. This bank driver is CPU and target-specific. Please refer to the **emulation probe manual** for additional information.

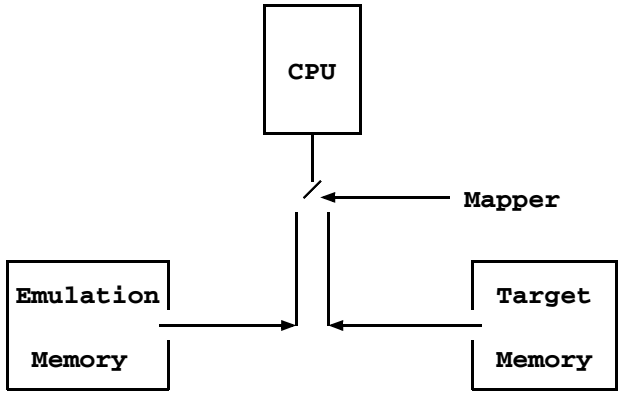
Function

The DATA memory is used for program and data storage during the development phase. Usually it replaces the EPROM on the target system. If no target system is available, the data and stack area must be supplied by the emulator system.

The DATA memory is assigned to the user program and data space by the command **MAP.Data**. Both static or dynamic memory modules are available. DRAM modules contain a lot of memory at lower cost, but must be refreshed by the system controller. Static memory modules have faster access time. When assigning memory, the emulation memory is parallel to the target memory. On write cycles both memories are updated. The wait-state generation results from the logic on the target system. On read cycles, both memories are accessed, the data stream used by the CPU is defined by the **MAP.Intern** or **MAP.Extern** command. Emulation memory can be assigned in 4K blocks. The definition of the Internal/External mapping can be made bytewise.



Memory Write Cycle

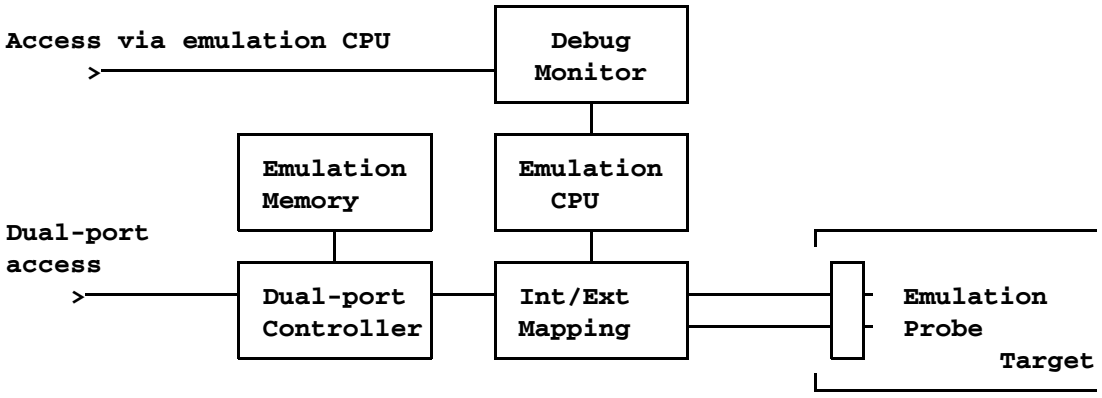


Memory Read Cycle

```
map.data   0x0--0x0ffff        ; define 64K of emulation memory
map.intern 0x0--0x0ffff        ; select internal memory

map.extern 0x100--0x101        ; map 2 bytes to target memory
```

Dual-port Access

The TRACE32 development system features dual-port emulation memory, accessible by both the emulation CPU and the system control CPU. The emulation CPU always reads data from memory 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.



Access Procedures

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

```
D  D:0x1000
```

is used to represent memory in the DATA address space, starting from address 1000, then the command

```
D  ED:0x1000
```

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

On emulation probes, which support CPUs with segmentation or MMU, access to memory can be made on logical or physical level:

```
D  AD:0x0                    ; physical access (Absolute Data)

D  D:0x0                     ; logical access
```

For additional information see chapter **MMU.**

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

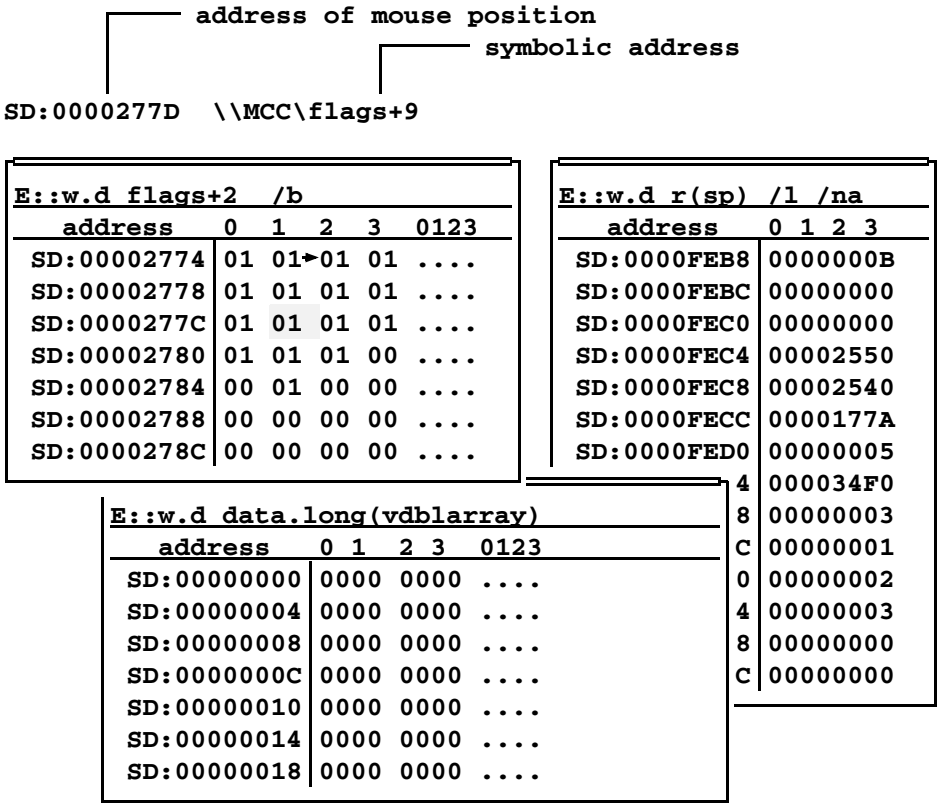
Code	Description
P	Program Space
D	Data Space
C	Access by the CPU
E	Direct access to emulation memory
EP	Direct access to program memory
ED	Direct access to data memory
A	Absolute (Physical addressing)
AD	Absolute Data
AP	Absolute Program
USR	User Specific Access Mode
EEPROM	EEPROM Write Access

Other classes are described within the emulation probe manual.

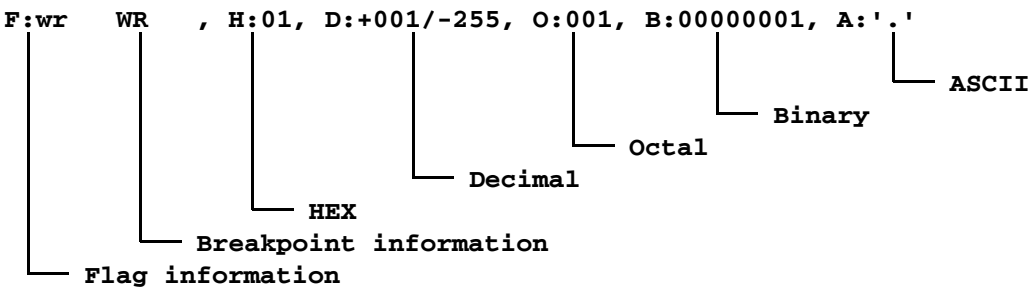
Memory will be displayed either on a HEX, a BINARY or on ASCII level. The display function mostly used is the **Data.dump** command

Data.dump	Display HEX and ASCII
SETUP.DUMP	Define standard format for Data.dump

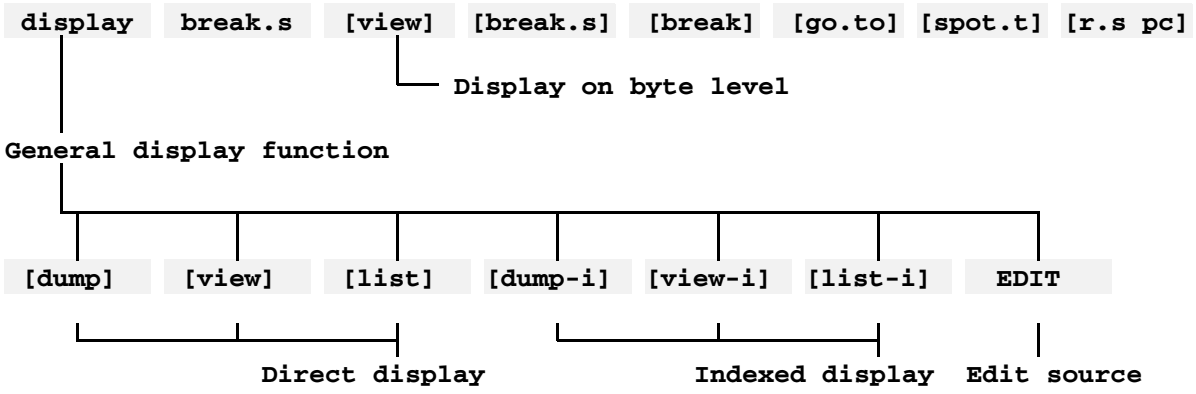
The display format can be set to Byte, Word or Long. The address specifier can be a fixed or variable value. Variable address specifiers lead to tracking windows. If the defined start address doesn't fit with the first displayed value, a pointer will mark this byte or word.



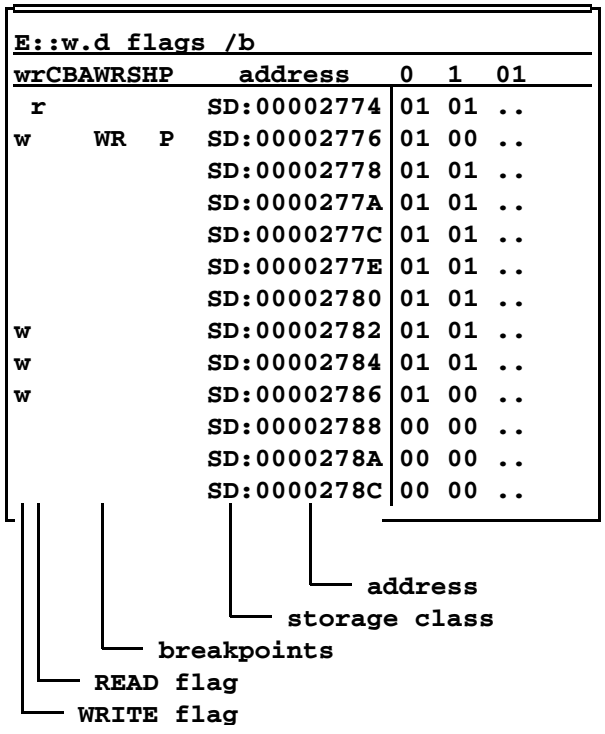
If the left mouse button is pressed, additional information will be displayed on the message line:



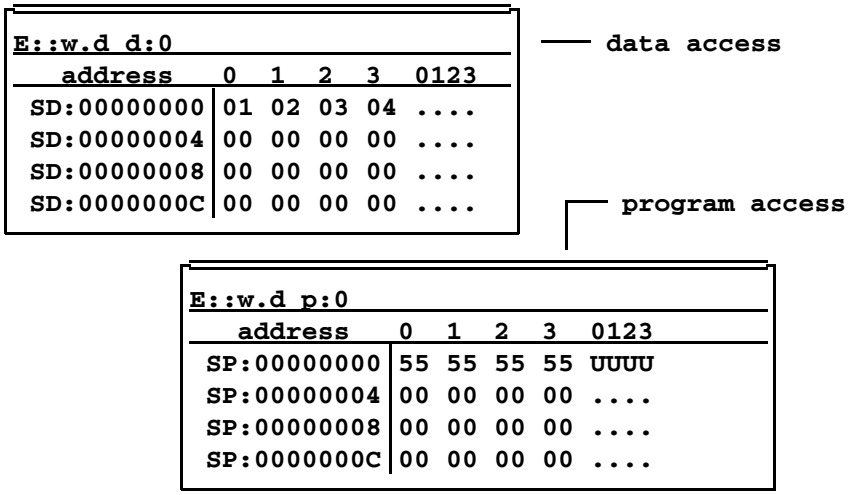
At the same time the address menu is displayed:



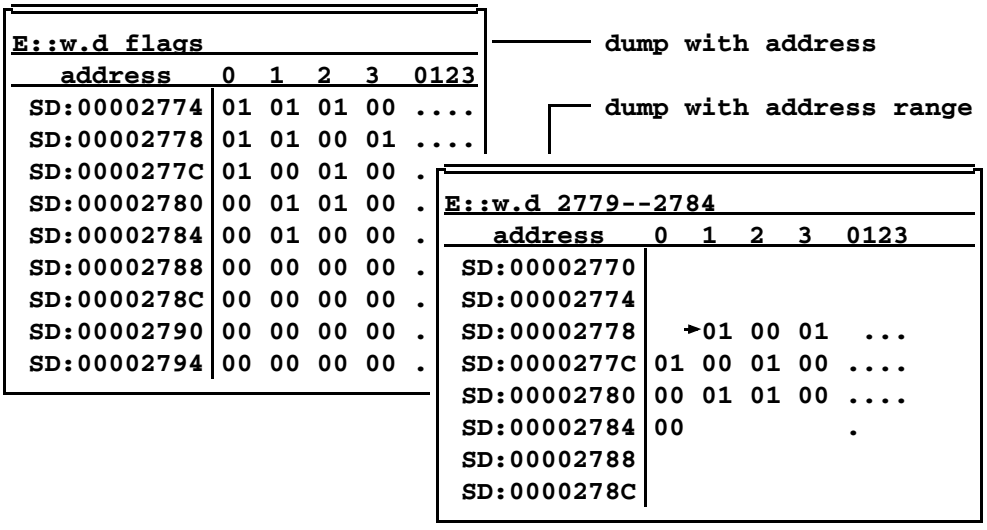
The left scale area contains additional information regarding the FLAG and BREAK memory.



Different memories are accessible through different storage classes:



The accessible memory may be limited by using an address range instead of an address specifier:



The flag and breakpoint information can be signaled by highlighting the displayed byte. Then the option **MARK** must be specified:

```
E::w.d flags /mark writeflag
   address    0  1  2  3  0123
 SD:00002774 01 01 01 00 ....
 SD:00002778 01 01 00 01 ....
 SD:0000277C 01 00 01 01 ....
 SD:00002780 00 01 01 01 ....
 SD:00002784 01 01 01 00 ....
 SD:00002788 00 00 00 00 ....
 SD:0000278C 00 00 00 00 ....
 SD:00002790 00 00 00 00 ....
```

With a short click on the left mouse button, the corresponding modify function can be generated on the command line:

```
E::w.d flags
   address    0  1  2  3  0123
 SD:00002774 01 01 01 00 ....
 SD:00002778 01 01 00 01 ....
 SD:0000277C 01 00 01 01 ....
 SD:00002780 00 01 01 01 ....
 SD:00002784 01 01 01 00 ....
 SD:00002788 00 00 00 00 ....
 SD:0000278C 00 00 00 00 ....
 SD:00002790 00 00 00 00 ....
```

```
E::D.S SD:2780 %BYTE 01
```

The **Data.View** and **Data.Print** windows work on a byte by byte level.

Data.View	Display in HEX and ASCII
Data.Print	Display in HEX and ASCII

```
E::w.data.view %bin flags
wrCBAWRSHP     address    data          value              symbol
 r           SD:00002774 | 01          00000001           \\MCC\flags
             SD:00002775 | 01          00000001           \\MCC\flags+1
             SD:00002776 | 01          00000001           \\MCC\flags+2
w            SD:00002777 | 00          00000000           \\MCC\flags+3
             SD:00002778 | 01          00000001           \\MCC\flags+4
             SD:00002779 | 01          00000001           \\MCC\flags+5
w            SD:0000277A | 00          00000000           \\MCC\flags+6
             SD:0000277B | 01          00000001           \\MCC\flags+7
```

symbol

displayed format

HEX value

address

breakpoints

flags

The display format will be binary, decimal, HEX or ASCII:

```
E::w.data.view %bin flags
ress    data          value
002774 | 01          00000001
002775 | 01          00000001
00
```

BINARY

```
E::w.data.view %d flags
ress    data          value
002774 | 01          1
002775 | 01          1
00277
```

SIGNED DECIMAL

```
E::w.data.view %long  flags
ress    data          value
002774 | 01 01 01 00  1010100
002778 | 01 01 00 01  1010001
00277C | 0
```

LONG

```
E::w.data.view %a  flags
ress    data          value
002774 | 01          '.'
002775 | 01          '.'
002776 | 01          '.'
```

ASCII

A special **VAR** option uses the information from the symbolic database for structured display of the HLL data structures:

```
E::w.d.v %var 2000
ress    data          value              symbol
00256E  FE            vbfield.q / .r     \\MCC\vbfield+1A
00256F  00            vbfield            \\MCC\vbfield+1B
002570  FF            vbfield.enum1 / .enum2 \\MCC\vbfield+1C
002571  E0            vbfield.enum2      \\MCC\vbfield+1D
002572  00                               \\MCC\vbfield+1E
002573  00                               \\MCC\vbfield+1F
002574  00            vshort = 0         \\MCC\vshort
002575  00              |                \\MCC\vshort+1
002576  00
002577  00
002578  00            vdarray[0] = 0     \\MCC\vdarray
002579  00            vdarray[1] = 0     \\MCC\vdarray+1
00257A  00            vdarray[2] = 0     \\MCC\vdarray+2
```

More complex data structures on the assembler level may be displayed with the commands:

Data.TABle	Display assembler structures
Data.CHAIN	Display linked list
Data.DRAW	Graphical array display

The **Data.CHAIN** window displays linked lists on assembler level. During HLL debugging linked lists are displayed with the **Var** command set.

```
E::w.d.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
```

Assembler arrays and data structures will be shown with the **Data.TABle** command. The parameters specify the size and the type of the elements.

```
E::w.d.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
```

Data Modification

All data may be changed by clicking to a data window or by using the **Data.Set** command.

Data.Set	Change memory

```
d.s  d:0x1000 0x12              ; set byte

d.s  d:0x1000  %l 12345678      ; write long word

d.s  d:0x1000  "TEST"           ; write ASCII string

d.s  d:0x1000--0x1fff 0x0       ; fill memory with zero

d.s  d:0x1000  12 /v            ; write with verify

d.s  ed:0x1000  0x12            ; write through dual-port access

d.s  d:0x1000 %f 1.345          ; write floating-point variable
```

Peripheral I/O

Active data windows read the memories with up to 30 accesses every
second. For I/O devices this concept may generate problems, when the chip changes the state on reading one byte (e.g. USARTs). I/O devices should be accessed by the following commands:

Data.In	Read data from port
Data.Out	Send data to port

The IN function reads a byte sequence from a single address, while the OUT function sends all bytes to one interface address.

```
d.in 10 3                       ; read 3 bytes form address 10H

d.out 10 'A' 0d 0a              ; write 3 bytes to address 10H
```

The program code can be displayed on either assembler level (ASM), or with source information (HLL), or both together (MIX). The assembler display re-assembles the code from memory. The HLL information is generated by the symbolic database. If no display mode is specified (**Data.List**), the selected operation mode displayed in the state line will define the display operation. If no address parameter or symbol is specified, the windows will track to the current PC (program counter, instruction pointer) value. The default access class is **P:**.

Data.List	Display symbolic
Data.ListAsm	Display ASM
Data.ListMix	Display ASM and HLL
Data.ListHll	Display HLL
SETUP.DIS	Define format for Data.List
Mode	Select debug mode (ASM, MIX, HLL)

ASM:

```
E::w.d.la
 addr/line    code              label      mnemonic                  comm
 SP:0000179A  13BC00018800                 move.b    #1,0(a1,a0.l*1); #1
 SP:000017A0  7012                         moveq     #12,d0          ; #1
 SP:000017A2  B082                         cmp.l     d2,d0           ; i,
 SP:000017A4  6CF0                         bge       $1796
 SP:000017A6  7400                         moveq     #0,d2           ; #0
 SP:000017A8  2649                         movea.l   a1,a3
 SP:000017AA  4A1B                         tst.b     (a3)+
 SP:000017AC  671E                         beq       $17CC
 SP:000017AE  2802                         move.l    d2,d4           ; i,
 SP:000017B0  D882                         add.l     d2,d4           ; i,
 SP:000017B2  5684                         addq.l    #3,d4           ; #3
 SP:000017B4  2602                         move.l    d2,d3           ; i,
```

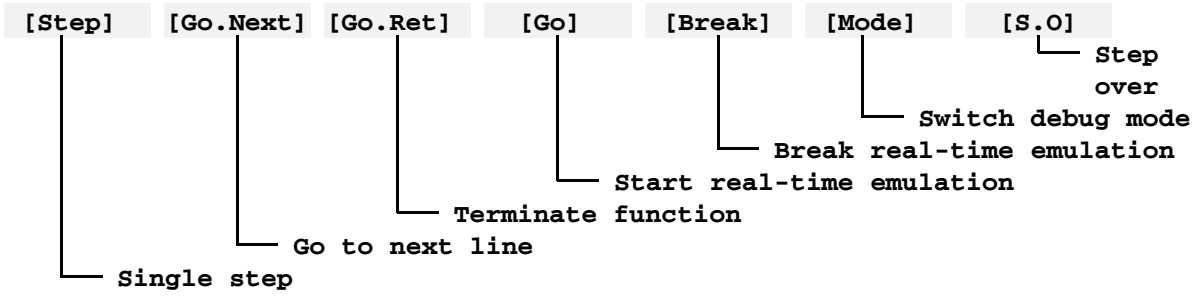
MIX:

```
E::w.d.lm
 addr/line    code              label      mnemonic                  comm
 SP:0000179A  13BC00018800                 move.b    #1,0(a1,a0.l*1); #1
 SP:000017A0  7012                         moveq     #12,d0          ; #1
 SP:000017A2  B082                         cmp.l     d2,d0           ; i,
 SP:000017A4  6CF0                         bge       $1796

         563          for ( i = 0 ; i <= SIZE ; i++ )
 SP:000017A6  7400                         moveq     #0,d2           ; #0
 SP:000017A8  2649                         movea.l   a1,a3
                      {
         565                  if ( flags[ i ] )
 SP:000017AA  4A1B                         tst.b     (a3)+
 SP:000017AC  671E                         beq       $17CC
```

HLL:

```
E::w.d.lh
 addr/line    source
         559          count = 0;

         561          for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE )

         563          for ( i = 0 ; i <= SIZE ; i++ )
                      {
         565                  if ( flags[ i ] )
                              {
         567                          prime = i + i + 3;
         568                          k = i + prime;
                                      while ( k <= SIZE )
                                      {
```

Activated windows will show the emulation menu:



Additional information on the FLAG and the BREAK memory can be displayed in the scale area. The track address (i.e. program counter) is displayed by a bar, the FLAG or BREAK setting may be highlighted.

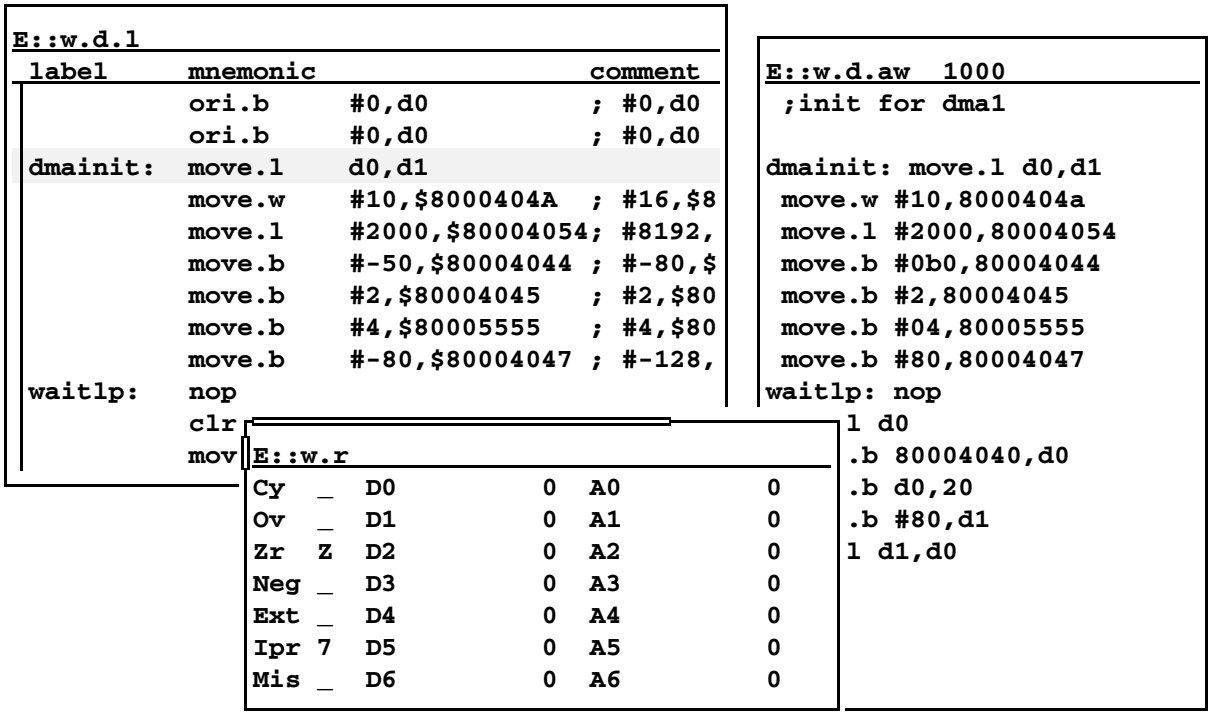
```
E::w.d.l /mark readflag
wrCBAWRSHP   addr/line    code              label      mnemonic
 r           SP:0000179A  13BC00018800                 move.b    #1,0(a1,
 r           SP:000017A0  7012                         moveq     #12,d0
 r           SP:000017A2  B082                         cmp.l     d2,d0
 r           SP:000017A4  6CF0                         bge       $1796
         H   SP:000017A6  7400                         moveq     #0,d2
             SP:000017A8  2649                         movea.l   a1,a3
         H   SP:000017AA  4A1B                         tst.b     (a3)+
             SP:000017AC  671E                         beq       $17CC
         H   SP:000017AE  2802                         move.l    d2,d4
             SP:000017B0  D882                         add.l     d2,d4
             SP:000017B2  5684                         addq.l    #3,d4
         H   SP:000017B4  2602                         move.l    d2,d3
```

The code may be changed symbolically by the in-line or on-screen assembler. The in-line assembler is called by a short mouse-click (left button) on the assembler line within the window.

Data.Assemble	In-line assembler
Data.PROGRAM	On-screen assembler

The assembler code is stored immediately in the emulation or target memory. This function can be used to create hardware drivers within HLL programs.

```
d.a 0x1000 jmp 0x1000          ; in-line assembler

w.d.aw 0x1000                  ; on-screen assembler
```



Data.LOAD.<*file_format*>	Load file to memory
Data.SAVE.Binary	Save memory to binary file
Data.SAVE.S3record	Save memory to file (S3 records)

Usually the memory is loaded from the compiler output. Nearly all standard format can be accepted:

AOUT	BSO/Tasking
AsciiOct **AsciiOctP** **AsciiOctA** **AsciiOctS**	Octal formats
AsciiHex **AsciiHexP** **AsciiHexA** **AsciiHexS** **AsciiHexC**	Hex formats
Binary	Binary format
BounD	Marc-Williams format
COFF	UNIX COFF format Intermetrics,Sierra
DBX	Berkley-UNIX format Native SUN GNU-C/C++
eXe	Paradigm LOCATE
HiCross	HIWARE format
HP	Hewlett Packard format (64000)
ICoff	INTROL format
Ieee	IEEE-695 format Microtec, Tasking, Intermetrics ALSYS ADA, Telesoft ADA SDS/XDADA
IntelHex	Intel-Hex format
MCDS	HIWARE MODULA-2 format
MCoff	Freescale Semiconductor DSP

OMF	Intel/Keil/Franklin OMF format (8051) Intel/Paradigm/Microtec (80186) Intel (80196) Intel,Pharlap (80386) Keil (80166)
ROF	OS/9 format Microware ULTRA-C
S1record	Motorola S1 format
S2record	Motorola S2 format
S3record	Motorola S3 format
SDS	Software Development Systems
sYm	Symbol table format
Ubrof	IAR / Archimedes format
VersaDos	Versados format
COSMIC	Whitesmiths / Cosmic format
XCOFF	PowerPC,GNU

When loading an HLL debug file, a database containing all HLL information is generated. Some compilers support symbols only, but no information on the data structure of the program. The BREAK memory is automatically mapped, when loading HLL information. Different options are available for loading the code without symbols, or symbols without code. Refer to the specified load command for additional information.

```
data.load.i  mcc.abs                  ; load code and data

data.load.i  mcc.abs /NoCODE          ; load source and symbols only

data.load.i  mcc.abs /NosYmbol        ; load no symbols
```

If more than one program has to be loaded, the **NoClear** option will prevent clearing the symbol and source database. It is possible to load different programs in different languages. The database supports C, C++, PASCAL, Modula2, ADA and PL/M.

```
data.load.i  a1.abs                   ; load first file
data.load.i  a2.abs  /NoClear         ; load second file
```

Usually the program loading is not verified.

```
data.load.i  mcc.abs  /verify         ; load with verification
```

If the source file is not found in the directory referred to by the debug file, another source directory can be specified.

```
data.load.i  mcc.abs /path \user\source    ; define path in load command

y.spath   \user\source                     ; define path for all load
                                           ; commands
d.load.i  mcc.abs                          ; load code and source
```

Program loading is done via the emulation CPU. If this is not possible, the program code can be loaded to the emulation memory directly:

```
data.load.i mcc.abs                        ; load through emulation CPU

data.load.i mcc.abs  e:                    ; load to emulation memory
```

Loading direct to the emulation memory is faster than loading by the emulation CPU, especially if the debugging is done by the BDM interface.

Some microcontrollers (i.e. 8051) are not able to write to the program memory. The program is loaded automatically into the emulation memory.

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. When downloading programs the CPU writes to this memory area, which will force bus errors or bus time-out. To avoid this problems the program can by loaded directly to the emulation memory by defining the destination address with **E:**. 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

Verify	Data memory is checked after write
Compare	Data is compared, no memory is changed
NoCODE	No code is loaded (symbols only).
NoClear	Symbols are not deleted before being loaded. This option is necessary if more programs must be loaded (Overlays, Banking).
NosYmbol	No symbols will be saved (even no program symbol). This option should be used, when data files are loaded.
NoBreak	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 breakpoints are executed in real time during HLL steps.
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 permanent search routes.
StripPATH	The basic name is extracted from the source paths given in the objectfile. This option can be used when the paths from the compiler should not be used, or the objectfiles have been compiled on another host with a different file syntax (e.g. VAX/VMS under MS-DOS).
LowerPATH	The filename is converted to lower-case characters. This option is useful when object files are compiled under a host which uses only capital letters for filenames (e.g. VAX/VMS, MS-DOS) and need to be loaded in a unix environment.

Find and Compare

Some additional data commands are listed below. For more information use the reference manual.

Data.Find	Search for HEX string
Data.COPY	Move data
Data.ComPare	Compare within data memory or to file
Data.GOTO	Set data cursor
Data.Test	Memory test
Data.SUM	Compute checksum

```
d.f  0x0--0x0ffff  0d 0a              ; find hex string
d.f  0x0--0x0ffff  "test"             ; find ASCII string
d.f  0x0--0x0ffff  %l 12345678        ; find long word
d.f                                   ; find next
d.copy  0x0--0x0fff  0x1000           ; copy within memory
d.copy  0x0--0x0fff   e:              ; copy to emulation memory
d.copy  e:0x0--0x0fff c:              ; copy to target memory
d.copy  0x0--0x0fff                   ; read and write-back
d.compare 0x0--0x0fff 0x1000          ; compare within memory
d.compare 0x0--0x0fff /file           ; compare with file
d.t 0x0--0x0ffff /p                   ; test memory
d.sum  0x0--0x0fff                    ; compute checksum
print data.sum()                      ; print it
```

Database Structure

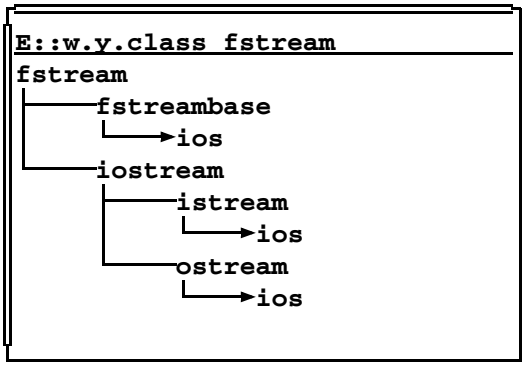
Symbolic information is stored in a indexed database:

Statics	Symbols with fixed address
Functions	Program functions
Locals	Local symbols
Moduls	Module names
Types	Type definitions
Sources	Source text information
Lines	HLL lines
Sections	Physical program ranges
Programs	Programs
Compilers	Compiler information

Symbol Display

Usually the symbol information is loaded together with the binary code by the **Data.LOAD** command (HLL debugging). The database information can be displayed by different commands:

sYmbol.INFO	List all information related to one symbol
sYmbol.Class	List symbol classes
sYmbol.name	List all information
sYmbol.List.Function	List program functions
sYmbol.List.LINE	List program lines
sYmbol.List.Local	List locals
sYmbol.List.Program	List program names
sYmbol.List.SOURCE	List source information
sYmbol.List.Static	List static symbols
sYmbol.List.Type	List type information




```
display C++ classes

baseclass
virtual baseclass
```

The symbol list commands can be used together with wildcard characters. The search patterns are:

'*'	Matches any string, empty strings, too.
'?'	Matches any character, except an empty character.
""	Can be used to input special characters like '*' or '?'

Examples:

```
y                       ; displays all symbols

y *\*                   ; displays all local symbols

y \*\*                  ; displays all local symbols of globalfunctions and
                        ; module local symbols

y \mcc\*                ; displays all symbols local to module 'mcc'

y func9\*               ; displays all symbols local to function 'func9'

y i                     ; displays all symbols with the name 'i'

y \mcc\*\i              ; displays all local symbols in module 'mcc' with
                        ; the name 'i'

y \m*\f*\i*             ; displays all local symbols with the symbol name
                        ; beginning with 'i'in all functions with function
                        ; names beginning with 'f' in all modules beginning
                        ; with 'm'

y * *                   ; displays all symbols with HLL type information

y * *ptr                ; displays all symbols, which have an HLL type that
                        ; ends with 'ptr', e.g. 'intptr'

y * char *              ; displays all symbols, which have an HLL type that
                        ; begins with the text 'char ', e.g. 'char *', 'char
                        ; [10]', 'char &'

y * char "*"            ; displays all symbols with HLL type of 'char *'

y * *"*"*               ; displays all symbols with type names, that
                        contain
                        ; a '*'
```

```
y ops::operator*      ; displays all operators defined for the C++ class
                      ; 'ops'

y operator*           ; will search for all symbols, beginning with the
                      ; string 'operator'
                      ; NOTE: the demangler is not active, so no operators
                      ; of C++ classes are listed!

y                     ; display all operator of all classes, that have
*::operator*(int)     ; only one argument of type 'int'
```

Symbol Macros

The pre-command **sYmbol.ForEach** repeats one command for all symbols, which match to the symbol pattern.

sYmbol.ForEach

```
y.fe "b.s *" *PACK                ; set breakpoints on all symbols ending
                                  ; with "PACK"

y.fe "b,s * /C" *PACK             ; set 'C' breakpoints on all symbols
                                  ; ending with "PACK"

y *func* /c "b.s"                 ; will execute the command B.S <symbol>
                                  ; for all symbols containing 'func'

y * * /c "b.s v.end(""*"") /c"    ; will execute the command
                                  ; B.S v.end(<symbol>)/C and set 'C'
                                  ; breakpoints on the last address of
                                  ; each HLL function or variable

y * /c "b.s"                      ; will execute the command B.S <symbol>

y * /c "b.s * /a"                 ; will execute the command
                                  ; B.S <symbol> /A

y * /c "v.v ?"                    ; will build a command line V.V <symbol>
                                  ; and leave the symbol window
```

Load Source HLL

Source information is normally loaded automatically by the **Data.LOAD** command. If the NOSOURCE option is used, source information will be loaded later and for the required modules only. The command below loads the source information separately. Usually this function will be needed to load source information if source paths were changed since the last compilation, e.g. if the compiler and debugger run on different host systems.

sYmbol.SourceLOAD.source	Reload source information
sYmbol.SourcePATH	Define path for symbol information

Loading Assembler Source

To support assembler source debugging, list files from some compilers will be loaded directly. Debugging is then done in HLL mode. All source information including comments will be displayed.

sYmbol.LSTLOAD.IAR	IAR/ARCHIMEDES assembler format
sYmbol.LSTLOAD.HPASM	HP assembler format
sYmbol.LSTLOAD.MicroWare	Microware assembler format
sYmbol.LSTLOAD.MRI68K	Microtec assembler format

Special Options

Some special options simplify the user interface. For use of this options refer to the compiler manual and the **Data.LOAD** command for this compiler output.

Frame.CONFIG.Asm	Define stack backtrace mode
sYmbol.CASE	Select case distinction
sYmbol.CUTLINE	Limit size of text blocks
sYmbol.POINTER	Define stack and frame pointer
sYmbol.POSTFIX	Define symbol postfix character
sYmbol.PREFIX	Define symbol prefix character
sYmbol.RELOCate	Relocate symbols (position independed code)
sYmbol.STRIP	Cut symbol length
sYmbol.DEMangle	C++
sYmbol.LANGUAGE	Select language for HLL expressions

Accessing Variables

HLL data structures are displayed and accessed through the **Var.** command set.

Var.set	Display and set variable

Every visible variable can be shown by the **Var.set** command. If no new value is given, the actual value will be displayed.

```
E::v vdouble

vdouble = 1.0
```

Variables may be changed by definition of a new value.

```
v vdouble = 2.0+0.4E3

v \mod1\venumvar = enum4       ; assignment of value 'enum4' to the
                               ; variable 'venumvar' in module 'mod1'

v charptr[4] = 'x'             ; Content of the 5th element of an array
                               ; 'charptr' is set value 'x'.

v charptr[i+4] = 'x'           ; Content of the i+4 element of an array
                               ; 'charptr' is set value 'x'.

v s1 := ['x'..'z','Y']         ; Assignment to a PASCAL set.
```

Symbol Prefix and Postfix

Most of the compilers add a special character (for example '.' or '_') in front of or behind the users symbol names. The user need not enter this character. The symbol management will automatically add this character, if necessary.

Example for the processing of prefix/postfix characters:

Symbol Table	Entry	HLL Windows	Assembler windows
_vfloat	_vfloat or vfloat	vfloat	_vfloat

Symbol Paths

There are two modes for the entry of a symbol name: entering a complete symbol path or solely a symbol name. If only a symbol name is used, the access will occur to the symbol valid for the current program part only. If symbol names are used more than once, local symbols will be preferred to symbols of higher blocks.

By specifying a complete symbol path, access to any symbol is possible. Each part of the symbol path is separated by a '\'. A complete path has to begin with a '\'. The following path versions are allowed:

```
\\program\module\function ...
\module\function ...
function ...
```

If the specified symbol represents a function, the access to local variables of this function and of nested functions will be possible :

```
... \function\local
... \function\function ...
```

If using PASCAL, as many functions as chosen will be nested.

Line numbers can be specified in the following way:

\linenumber

\module\linenumber

\\program\module\linenumber

... \linenumber\column

... symbol\linenumber_delta

The address of the high level language block containing the specified line number is returned by this operation.

```
d  flags                        ; symbol name

d  func1\i                      ; symbol 'i' in 'func1'

d  \1234                        ; line number 1234

d  \mod1\1234                   ; line number 1234 in module 'mod1'

d  func1\12                     ; twelve lines after symbol 'func1'
```

Search Paths

If no complete path is entered, the symbol will be searched in the following sequence

1. Local symbols (interior block ... exterior block)
2. Static symbols of block
3. Static symbols of module
4. Global symbols of current program
5. All other static symbols

Mangled Names and C++ Classes

The class of a method can be left out, if this method exists only in one class. The class is always required, if the constructor, destructor or an overloaded operator will be accessed. The quotation marks ` can help to allow special characters if the C++ name is used in the regular TRACE32 syntax. They are not required in the **Var** command group.

```
E::d.l class1::method1

E::d.l method1                          ; access to same method (not for
                                        ; all compilers possible)

E::d.l class1::class1                   ; creator of class class1
E::d.l class1::~class1                  ; destructor of class class1

E::d.l `class1::operator++`             ; overloaded operator

E::d.l `class1::operator+(int)`         ; overloaded operator with prototype
```

Function Return Values

The return value of a function is entered in the symbol list as a local variable of the function. It has always the name 'return'.

```
v return                               ; display return value
```

Special Expressions

The expression interpreter accepts some extensions to the language. All typechecks and rangechecks are handled as freely as possible. The accessing of the data beyond the array limits is allowed.

A de-reference of a plain number will assume that it is a pointer to character:

```
v *2000 = 1                 ; set byte at location 2000 (decimal)
```

All labels (typeless symbols) can be used in expressions. They are taken as variables of the type void. They can be casted directly to the required type.

```
v __HEAP                     ; displays nothing (if __HEAP is a label)
v *__HEAP                    ; assumes __HEAP as a pointer to character
v (long)__HEAP               ; takes __HEAP as a 'long' variable
```

Function calls can be made to plain addresses or typeless symbols. The return value is assumed to be 'void'.

```
v (0x2000)(1,2,3)               ; calls the function at 2000 (hex)
v __HEAP(1,2,3)                 ; calls the function at the label __HEAP
```

Extracts of arrays can be made with 'range' expressions. The operations allowed with such extracts are limited. This allows the display of zero sized arrays and display of pointers to arrays.

```
v flags[2..4]                   ; display elements 2 to 4
v vdblarray[2..4][i-1..i+1]     ; display part of two-dimensional
                                ; array
v vpchar[0..19]                 ; display array at pointer 'vpchar'
v (&vchar)[0..19]               ; takes the location of one element
                                ; to build up an array
```

Assigning strings can cause two different reactions. If the string is assigned to a NULL pointer, the target function 'malloc' is called to gather memory for the string and the resulting address is assigned to the pointer variable. If the string is assigned to a non zero pointer or an array, then the contents of the string are copied over the old contents of the array.

```
v vpchar = 0x0
v vpchar = "abc"                ; will call the 'malloc' function

v vpchar = 0x100
v vpchar = "abc"                ; copy the string "abc" to location 0x100
```

Strings used in arguments to functions are allocated on the stack.

```
v strlen("abc")                 ; the string will reside on the stack
```

Comparing a pointer or array against a string compares the contents of the string.

```
v.g.t pname=="TEST"             ; execute program till string equal
```

A type alone can be an expression. This is especially useful for the **Var.TYPE** command to display the layout of a structure or C++ class.

```
v.type %m Tree                  ; displays the layout of class 'Tree'
```

Elements of unions can be accessed by indexing the union like an array. The first element of the union is accessed with index 0.

```
struct
{
     enum evtype type;
     union
     {
          strcut sysevent sys;
          struct ioevent io;
          struct winevent win;
          struct lanevent lan;
     }
     content;
}
signal;

w.v.v signal.content[signal.type]
```

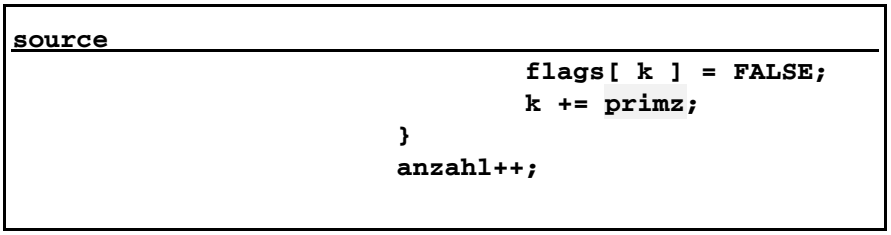
The syntax for MODULA2/PASCAL expressions has been extended for type casts and hexadecimal numbers.

```
v.v flags[0] := 12H            ; standard MODULA hexadecimal syntax
v.v flags[0] := 0x12           ; also accepted (like 'C')

v.v CARDINAL(1.24)             ; typecast like 'C': (CARDINAL) 1.23
v.v ^CARDINAL(0x10)            ; typecast like 'C': (CARDINAL *) 0x10
```

Lower and upper case letters are distinguished in symbol names. The command **sYmbol.CASE** switches off this differentiation. The length of symbol names is limited to 255 characters. The amount of symbols depends on the size of the system memory.

Displaying Variables

Variable structures may be displayed in windows. By clicking with the mouse to one variable or element, the corresponding **Var.set** command will be executed.




```
E::V primz =

  [ok]    formats    <var>
```

Var.View	Display variables by name
Var.Ref	Display actual variables
Var.Watch	Display variables selected by Var.AddWatch
Var.AddWatch	Display variables in watch window
Var.Local	Display local variables in function
Frame.view	Display stack frames
Var.TYPE	Display type of expression
Var.CHAIN	Display linked list
Var.TABle	Display data structure and pointers
Var.DRAW	Graphical array display

The **Var.View** or **Var.Watch** command shows local and static variables defined by their name. Many options are possible to define the format or to print-out additional information.

Format: [**%**<*format*>] …

<*format*>: **all**
DEFault
Type[**.on** | **.OFF**]
Multiline[**.on** | **.OFF** | **.2** | **.3** | **.4**]
Compact[**.on** | **OFF**]
Fixed[**.on** | **OFF**]
PDUMP[**.on** | **.OFF**]
DUMP[**.on** | **.OFF**]
Index[**.on** | **.OFF**]
String[**.on** | **.OFF**]
Decimal[**.on** | **.OFF**]
Hex[**.on** | **.OFF**]
Ascii[**.on** | **.OFF**]
BINary[**.on** | **.OFF**]
sYmbol[**.on** | **.OFF**]
Recursiv[**.on** | **.OFF** | **.2** | **.3** | **.4**]
Location[**.on** | **.OFF**]
HIdden[**.on** | **.OFF**]
INherited[**.on** | **.OFF**]
MEthods[**.on** | **.OFF**]
SPaces[**.on** | **.OFF**]
E[**.on** | **.OFF**]

For more detailed information about display formats refer to the Emulator Reference Manual.

The **Multiline** option displays arrays or structures in a 2-dimensional way.

```
E::w.v.v flags i flags[i] vdblarray
flags = (1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1   ── structure
i = 0                                                ── variable
flags[0] = 1                                         ── expression
vdblarray = ((0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0,     ── array
```

```
E::w.v.v flags i flags[i] %m vdblarray
flags = (1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
i = 0
flags[0] = 1
vdblarray = ((0, 0, 0, 0, 0, 0),                     ── array
             (0, 0, 0, 0, 0, 0),
             (0, 0, 0, 0, 0, 0),
             (0, 0, 0, 0, 0, 0),
             (0, 0, 0, 0, 0, 0))
```

The **Location** option displays the address or the register name, where the variable is located.

```
E::w.v.v %l flags i flags[i] %m vdblarray
[C:2774] flags = (1, 1, 1, 0, 1, 1, 1, 1, 1, 1,   — static array
[D2] i = 0                                         — register variable
[C:2774] flags[0] = 1                              — static variable
[C:24F0] vdblarray = ((0, 0, 0, 0, 0, 0),
                      (0, 0, 0, 0, 0, 0),
                      (0, 0, 0, 0, 0, 0),
```

The **Type** option shows the variable definition.

```
E::w.v.v %t flags i flags[i] %m vdblarray
(char [19]) flags = (1, 1, 1, 0, 1, 1, 1, 1, 1,   — char array
(register int) i = 0                               — register variable
(char) flags[0] = 1                                — char variable
(char [6] [5]) vdblarray = ((0, 0, 0, 0, 0, 0),    — array of 2 dim.
                            (0, 0, 0, 0, 0, 0),
                            (0, 0, 0, 0, 0, 0))
```

The **Recursive** option shows linked lists.

```
E::w.v.v %m.3 %r.2 str
str = (word = 0x0 → NULL,
       count = 12345,
       left = 0x2540 → (word = 0x0 → NULL,
                        count = 12345,
                        left = 0x2540 → (word = 0x0,
                                         count = 12345,
                                         left = 0x2540,
                                         right = 0x0,
                                         field1 = 1,
                                         field2 = 2),
                        right = 0x0 → NULL
                                                    ,
                        field1 = 1,
                        field2 = 2),
       right = 0x0 → NULL
```

The value of the variable may be shown in **Hex**, **BINray**, **Ascii** or **Decimal**.

```
E::w.v.v %d %bin %hex  %ascii   flags[i]
flags[0] = 1 = 0x1 = '►' = 00000001
```

Either the value of a variable or the destination of a pointer may be displayed in dump mode.

```
E::w.v.v %dump i
i = 8 = <00 00 00 08>
```

```
E::w.v.v %pdump &flags
&flags = 0x2774 → <01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01>
```

Local variables are displayed within the **Var.Local** window. All variables within the actual function are displayed.

```
E::w.v.local
sieve()
i = 0
prime = 3
k
co
```

local variables

```
E::w.v.local %type
sieve()
(register int) i = 0
(register int) prime = 3
(re
(re
```

variables and type declaration

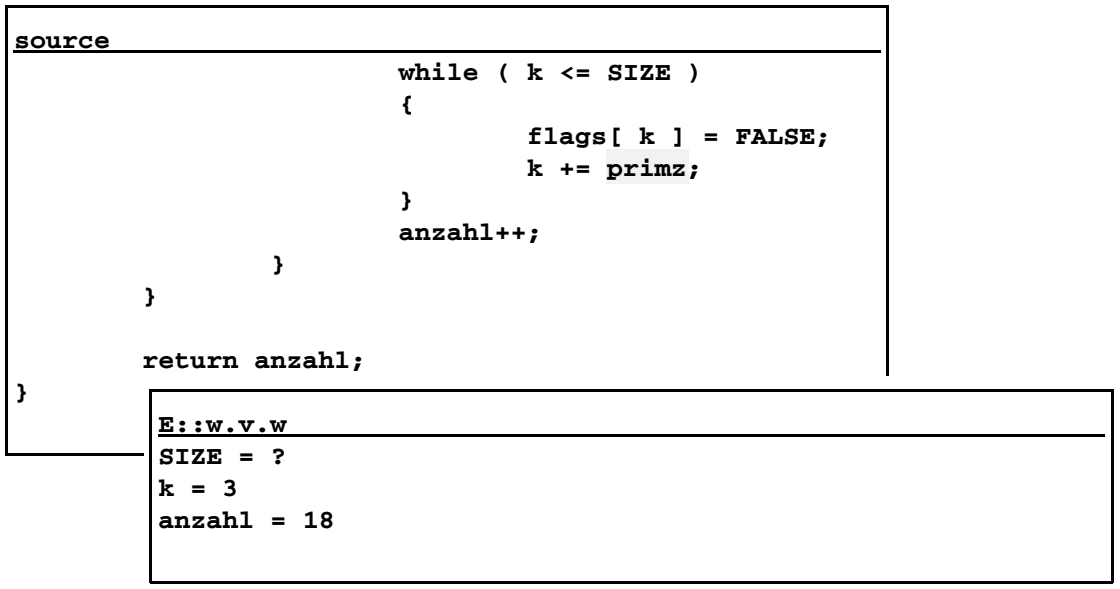
```
E::w.v.local %bin
sieve()
i = 00000000.00000000.0000000
prime = 00000000.00000000.000
k =
cou
```

binary display

```
E::w.v.local %l
sieve()
[D2] i = 0
[D4] prime = 3
[D3] k = 3
[D1] count = 0
```

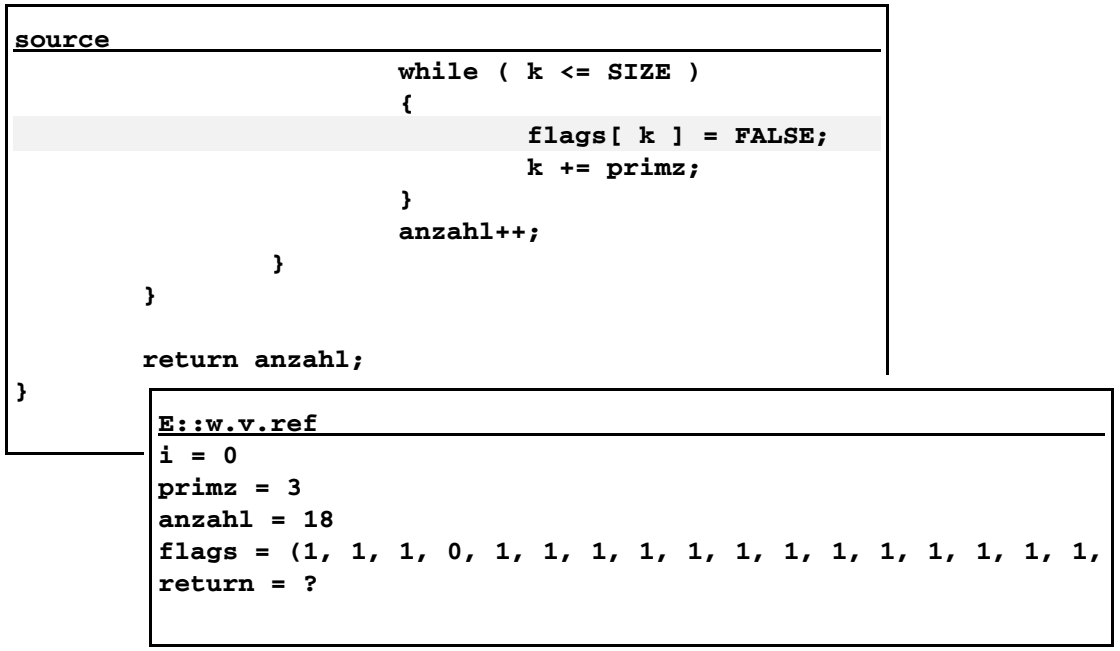
location and variable

The **Var.Watch** window displays variables, which are selected by the mouse or by the **Var.AddWatch** command.

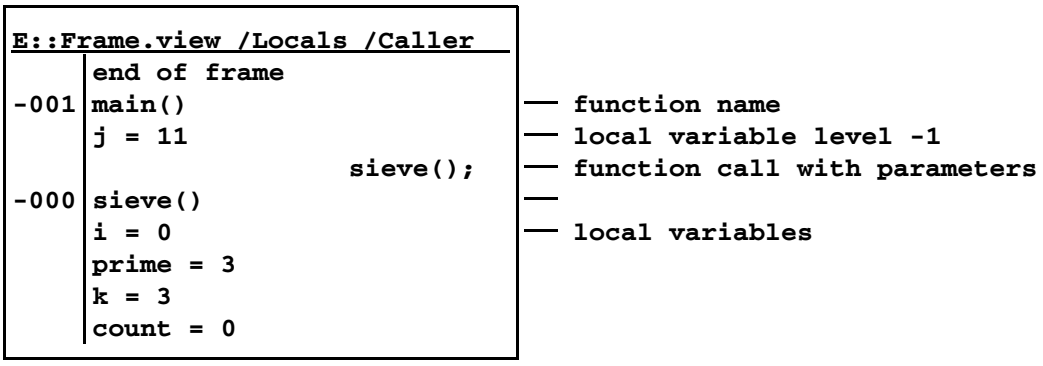


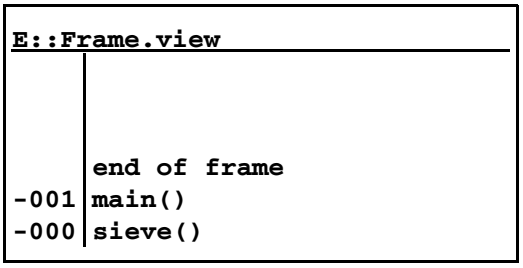

```
E::

  [V.V]    [V.AW]    [D.V]    [V.B.S/R]  [V.B.S/W]    [V.G]
             |
             |_____ Display variable in watch window
```

The **Var.Ref** window displays all variables referenced by the actual program counter position. Every variable accessed by this line is pushed to the window.



The **Frame.view** window displays the stack hierarchy. The user can see the nesting of functions, the local variables, the input parameters and the line which calls the subroutine.

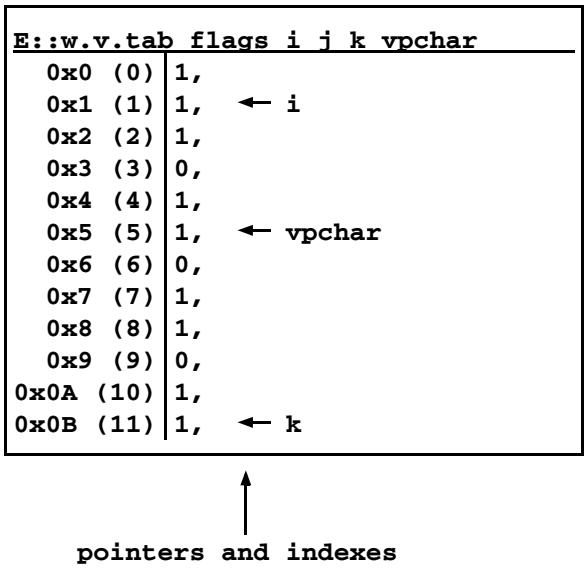




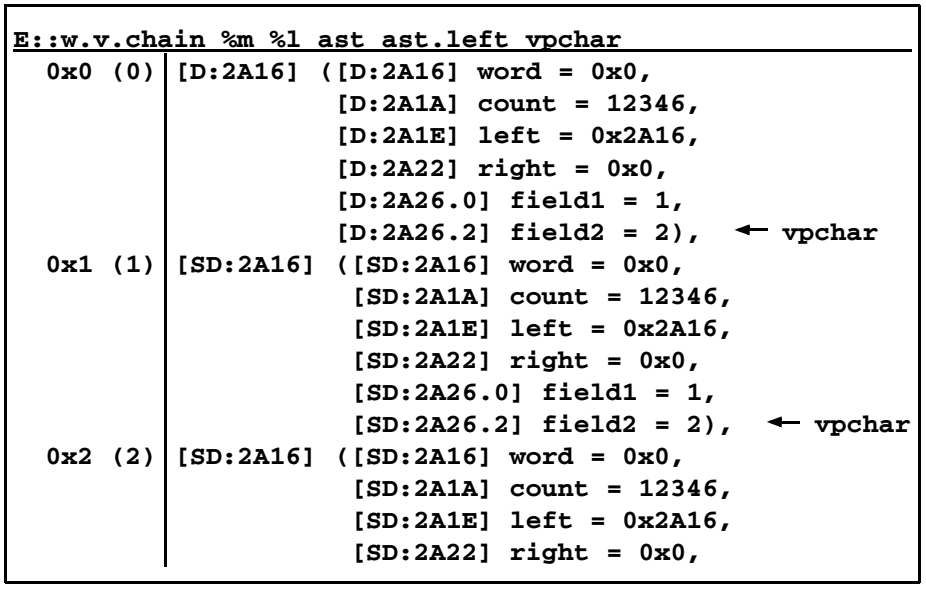
The **Var.Type** information displays the type information on data structures:

```
E::w.v.type %m struct1
struct1 struct1 struct(char * word,
                       int count,
                       struct1 * left,
                       struct1 * right,
                       int field1:2,
                       unsigned int field2:3)
```

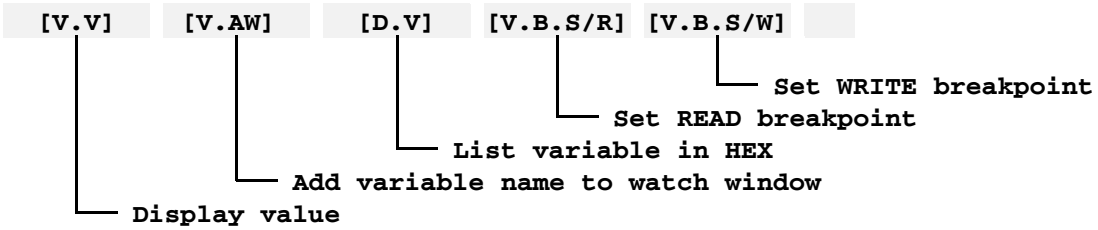
Arrays and Pointers to this arrays can be displayed by the **Var.TABle** window:

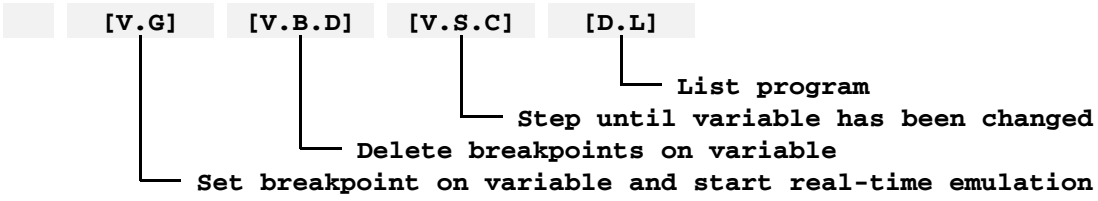


Linked lists are displayed with the **Var.CHAIN** command:



If the mouse pointer is placed on a variable name and the left mouse button is pressed, the variable based softkey line will appear:

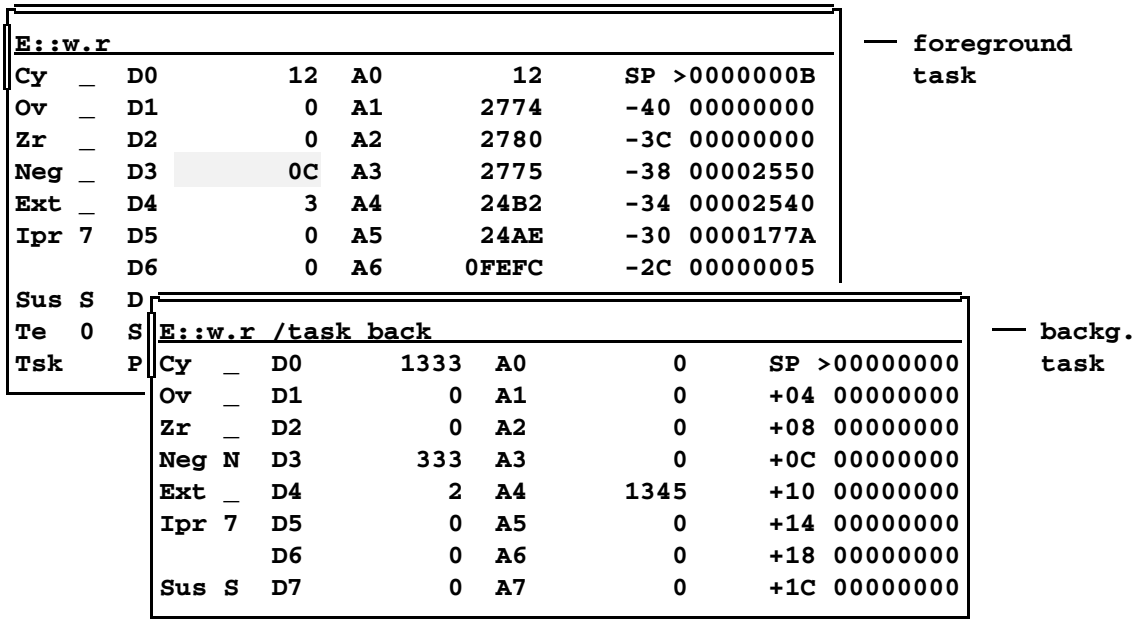




The state of the CPU and the stack can be displayed by the **Register** and **Frame** command. The emulator system offers up to 10 different shadow register sets. One register set (Foreground) is the default register set for debugging, while all others are used by the OS Awareness for background tasks.

Register.view	Show CPU register and stack
Register.Set	Change CPU register
Register.Init	Init CPU register
Frame.SWAP	Swap CPU registers with system registers
Frame.COPY	Copy register set to system register set

```
(register int) k        ── additional information for mouse position
```

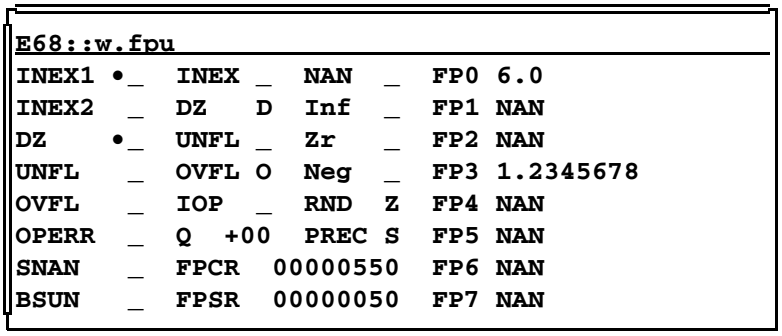



```
E::
H:0000000C, D:+0000000012/-4294967284, O:00000000014, A:'•••'
```

By clicking with the left mouse button, the name of register variables will be displayed in the state line. The message line shows the ASCII and binary information.

The FPU registers are displayed in a separate window. The FPU operations must be activated first.

FPU.ON	Activate FPU operations
FPU.OFF	Disable FPU operations
FPU.view	Show FPU register set
FPU.Set	Set FPU register
FPU.RESet	Init FPU registers



Internal or external peripheral systems can be shown on a logical level. The definition of this window is made by an ASCII file. The file is on disk for all internal peripherals. It can be upgraded with target-specific signal names or additional display information for external peripherals.

PER.view	Show peripherals of microcontrollers
PER.Program	Define layout of PER window
PER.ReProgram	Activate definition

```
E::w.per
TIMER
TSR    80 OV0 yes  MA1 no   CA1 no   OV1 no   MA2 no   CA2 no   OV2
TCR    00 ET1 inhibit       M1  inhibit       ET2 inhibit       M2
RR     0000
T0     8A88
T1     0000
T2     0000
```

Display window

```
E::w.per.p per68070.t32
group sd:80002020--80002029 "TIMER"
 line.byte 0 "TSR,Timer Status Register"
  bit 7 "OV0,Timer 0 Overflow" "no,yes"
  bit 6 "MA1,Timer 1 Match" "no,yes"
  bit 5 "CA1,Timer 1 Capture" "no,yes"
  bit 4 "OV1,Timer 1 Overflow" "no,yes"
  bit 3 "MA2,Timer 2 Match" "no,yes"
  bit 2 "CA2,Timer 2 Capture" "no,yes"
  bit 1 "OV2,Timer 2 Overflow" "no,yes"
 line.byte 1 "TCR,Timer Control Register"
  bit 6--7 "ET1,Timer 1 Event control" "inhibit,low-to-high
  bit 4--5 "M1,Timer 1 Mode control" "inhibit,match mode,ca
  bit 2--3 "ET2,Timer 2 Event control" "inhibit,low-to-high
```

Definition file

Preparations

After starting up the emulator by the SYS command, mapping memory by MAP and loading the program by the Data.LOAD command, the emulation system is ready for debugging. A typical start-up sequence is shown below:

```
e::                          ── select emulator device
winclear                     ── clear all windows
sys.up                       ── start emulation system
map.def 0--0ffff             ── map memory
d.load.i mcc.abs             ── load debug file
r.s ssp 0ff00                ┐  init registers
r.s pc main                  ┘
w.d.l                        ── list program
w.r                          ── display registers
enddo
```

On assembler level, the Data.List and the Register window should be used:

```
E::w.d.l
 addr/line     code              label      mnemonic                    comment
 SP:000017B0   D882                         add.l     d2,d4
 SP:000017B2   5684                         addq.l    #3,d4             ; #3,d4
 SP:000017B4   2602                         move.l    d2,d3
 SP:000017B6   D684                         add.l     d4,d3
 SP:000017B8   45F13800                     lea       0(a1,d3.l*1),a2; 0(a1,d3
 SP:000017BC   6006                         bra       $17C4
 SP:000017BE   4212                         clr.b     (a2)
 SP:000017C0   D5C4                         adda.l    d4,a2
 SP:000017C2   D684                         add.l     d4,d3
 SP:000017C4   7012                         moveq     #12,d0            ; #18,d0
 SP:000017C6   B083                         cmp.l     d3,d0
```

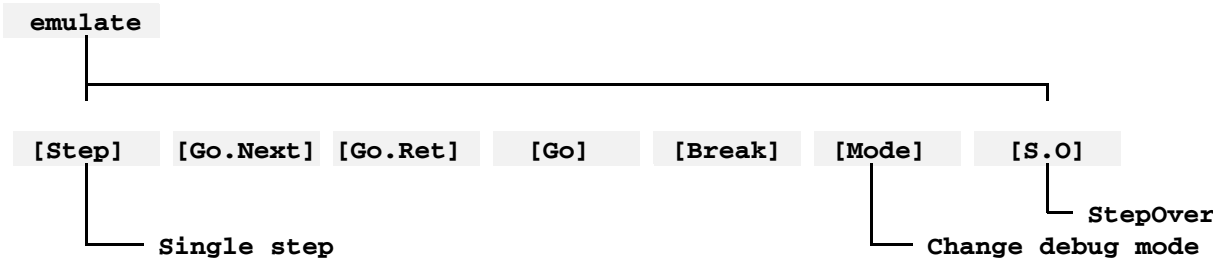
```
E::w.r
Cy   _   D0          12   A0          12      SP >0000000B
Ov   _   D1           7   A1        2774      -40 00000000
Zr   _   D2          0A   A2        278F      -3C 00000000
Neg  _   D3          0A   A3        277F      -38 00002550
Ext  _   D4          17   A4        24B2      -34 00002540
Ipr  7   D5           0   A5        24AE      -30 0000177A
         D6           0   A6      0FEFC       -2C 00000005
```

```
E::w.d flags /b
   address     0   1   01
 SD:00002774  01  01  ..
 SD:00002776  01  00  ..
 SD:00002778  01  01  ..
 SD:0000277A  00  01  ..
 SD:0000277C  01  00  ..
 SD:0000277E  01  00  ..
```

After this preparations the emulator system is ready for debugging.

Step	Single step
Step.Asm	Single step on assembler level
Step.Mix	Single step on mixed level
Step.Hll	Single step on HLL level
Step.Over	Step over call
Mode	Switch debug mode ASM, MIX, HLL

The bar within the Data.List window displays the actual program counter position. The register window is updated after every step. The main debug commands can be reached through the « emulate» path. This menu is freely configurable by the **SETUP.EMUPATH** command.



Multiple steps can be done by defining an argument. The break function (Ctrl-C) will stop operation.

```
s 10.                                   ; do 10 steps

s 0.                                    ; step endless
```

Single Step on HLL

Single stepping on HLL level is done by the hardware. The emulation runs to the next activated HLL line. Usually the register window is not needed for HLL debugging. Global and local variables can be shown directly.

```
E::w.d.l
 addr/line    source
         559         count = 0;

         561         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

         563         for ( i = 0 ; i <= SIZE ; i++ )
                     {
         565                 if ( flags[ i ] )
                             {
         567                         prime = i + i + 3;
         568                         k = i + prime;
                                     while ( k <= SIZE )
                                     {
```

```
E::w.v.f                                  E::w.v.l
                                          sieve()
                                          i = 11
                                          count = 0
     end of frame
-001 main()
-000 sieve()
```

The debug mode can be selected by clicking to the corresponding field in the state line or by the Mode command

```
SP:00001794   \\MCC\mcc\sieve+0C                          ........... HLL   AI
                                                                       |
                                                        Debug mode ----
```

Cycle Step

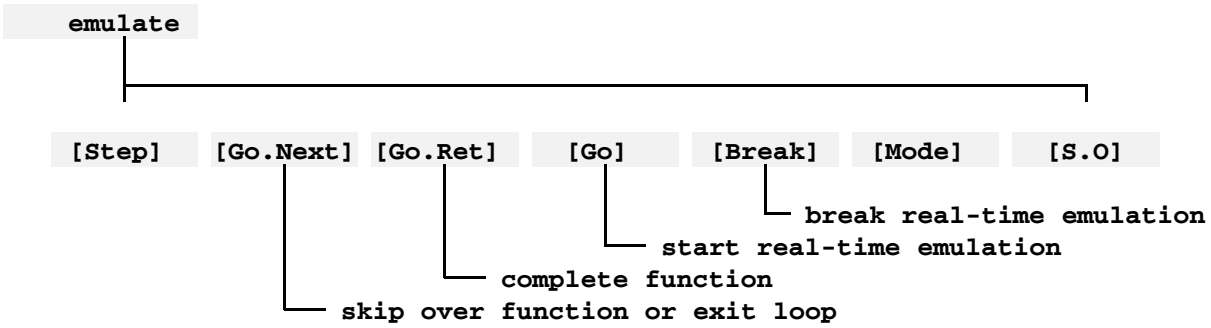
Some emulators support stepping on a cycle-by-cycle basis. This is useful for analyzing hardware:

Step.CycleWait	Cycle step (hold within bus cycles)
Step.CycleReq	Cycle step (hold between bus cycles)

```
s.cw                            ; start emulation with cycle step
s.cw
…
s.cw
b                               ; break emulation
```

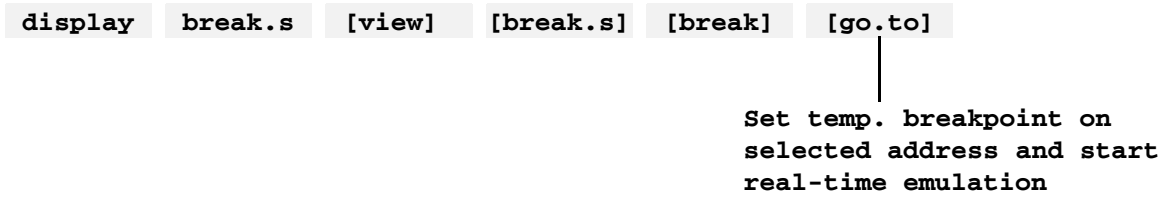
Real-time emulation is started with the **Go** command and stopped with the **Break** command or by the corresponding function key within the « emulate» path. In HLL mode, the program is stopped at the next HLL line, while in ASM and MIX mode the real-time emulation stops immediately.

Go	Start real-time emulation
Go.NoBreak	Start real-time emulation without program breakpoints
Go.Asm	Run in ASM mode
Go.Mix	Run in MIX mode
Go.Hll	Run in HLL mode
Go.Next	Run to next line
Go.Return	Terminate HLL function
Go.Up	Run to specific HLL nesting
Break	Stop real-time emulation
Break.Halt	Activate NMI and break



Real-time emulation is displayed in the state line by the character 'G':

The program execution can be started with temporary breakpoints from the address menu:



Several commands are implemented to control the emulation by complex events:

Break.Pass	Pass while expression is true
Go.Change	Run until expression changes
Go.Till	Run until expression is true
Step.Change	Step until expression changes
Step.Till	Step until expression is true

```
b.s flags                    ; set breakpoint
b.pass r(d0)>0x0             ; set pass condition
g                            ; start real-time emulation

b.s func1_end                ; set breakpoint to end of assembler routine
g.c r(d0)                    ; run until register D0 has been changed

b.s sieve                    ; set breakpoint
go.till r(sp)<0x2000         ; run until stackpointer less than 2000H

s.c data.byte(pointer)       ; step until the expression changes

s.till  r(d0)>1000.          ; step until register D0 higher 1000.
```

To control real-time emulation from complex data structures, several conditional emulation commands are implemented:

Var.Break.Pass	Define pass condition
Var.Go.direct	Set breakpoint on data structure and start real-time emulation
Var.Go.Change	Run until expression has changed
Var.Go.Till	Run until expression is true
Var.Step.Change	Step until expression has changed
Var.Step.Till	Step until expression is true
Var.Call	Call function with parameters

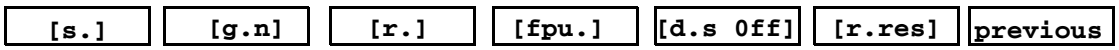
```
b.s var_float                ; set breakpoint on float variable
v.b.pass var_float > 2.0     ; break only if value less than 2.0

v.g flags                    ; set breakpoint on data structure and
                             ; start real-time emulation

v.b.s flags                  ; set breakpoints on data structure
v.g.change flags             ; run until data structure has been changed

v.b.s flags                  ; set breakpoints on data structure
v.g.till flags[i]==0x0       ; stop if expression is true

v.s.c flags                  ; step until data structure has been changed

v.s.till flags[3]==0x0       ; step until expression is true

v.call func2(1.0,2.0)        ; call function with parameter passing
```

The function keys in the « emulate» path can be defined by the user:

SETUP.EMUPATH Define softkeys

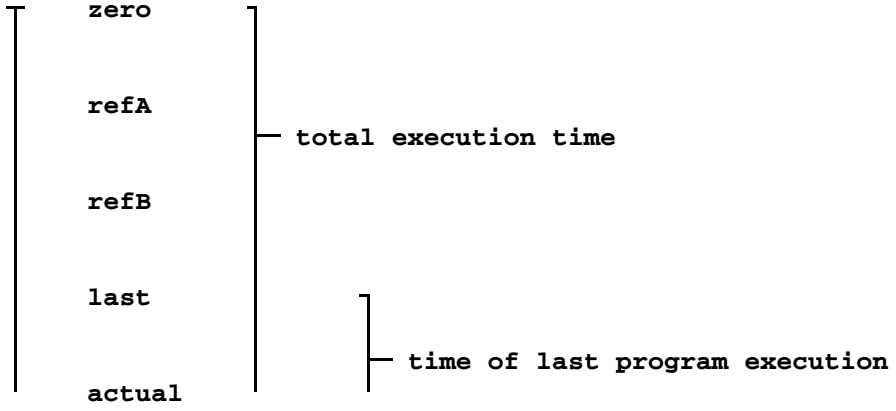
```
setup.emupath "s." "g.n" "r." "fpu." "d.s 0x0ff000 0" "r.res"
```

The following softkeys are generated:



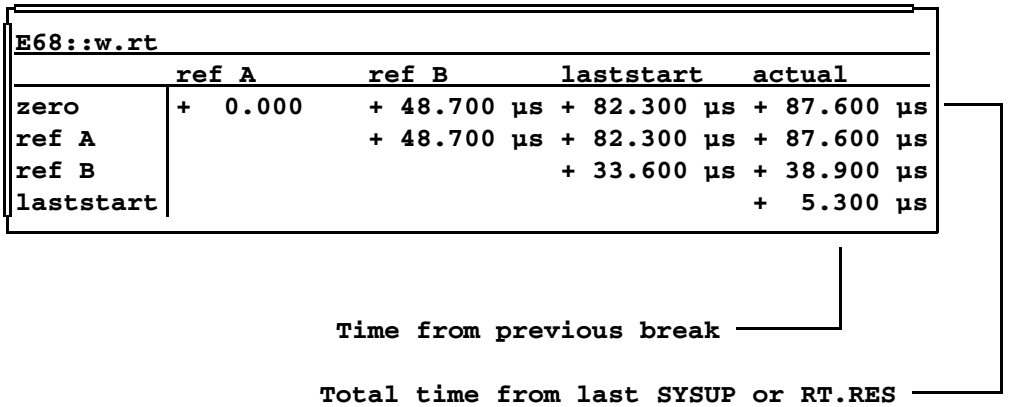
Function

The runtime analyzer enables checking of program execution times between two breakpoints. Therefore, in addition to the state analyzer, a second, independent timing analyzer is available to user. The resolution of runtime measurement is 100 ns. Solely the execution time of the foreground program is measured. The execution time of the background program is not taken into consideration. Two reference points may be set in order to evaluate timing differences.




```
rt.res                          ; reset timer
r.s pc main                     ; set PC to start of program
g func1                         ; run to func1
rt.refb                         ; set reference point
g func2                         ; run to func2
s                               ; single step
```

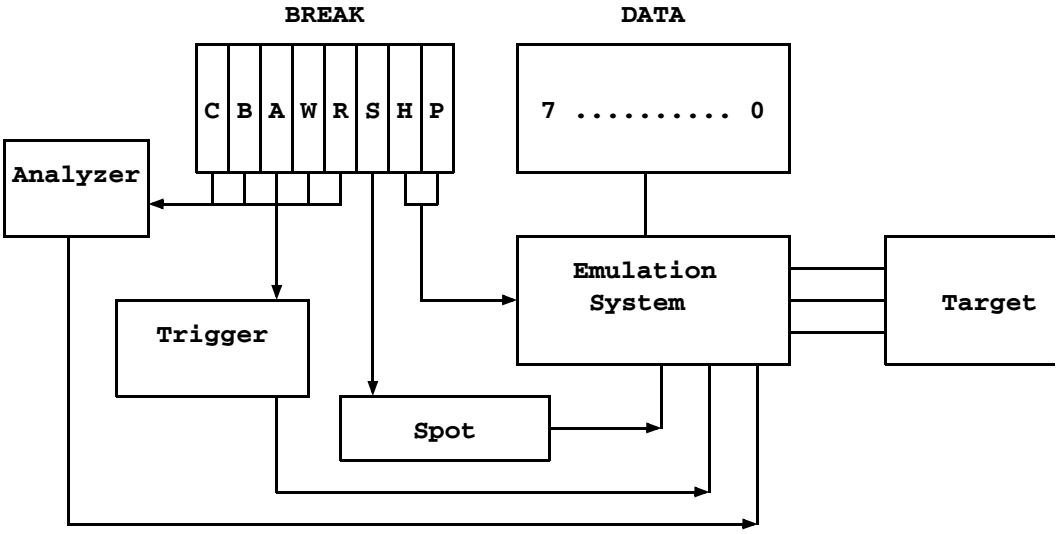
Total time, as well as the previous emulation command execution time, is automatically recorded. The differences between the individual reference points are displayed in tabular form.



RunTime.Init	Reset timers
RunTime.RESet	Reset timers
RunTime.refA	Set reference point
RunTime.refB	Set reference point
RunTime.state	Display execution times

Function

The TRACE32 development system has an extra memory for the breakpoint system. It can be activated separately from emulation memory and used for marking breakpoints or breakpoint ranges. Breakpoint memory is mapped parallel to target or emulation memory automatically on setting breakpoints. Unused memory can be freed for re-allocation if required. Breakpoint mapping is done in 4K blocks. A maximum of 8 different breakpoints can be set at any given program address or data address. The breakpoint should not be considered as breakpoints only, but also as address identifiers used by the analyzer trigger system or the performance analyzer.



Program

This is the default breakpoint for the program code range. Program breakpoints can only be set at the beginning of commands. Address ranges are allowed only to protected memory against unintentional program execution. When data is accessed, this breakpoint is ignored, unless it has been activated in the trigger system as an asynchronous breakpoint (**TrMain.Set Program** command).

HII

High-level language debugging is supported by hardware. To this end, all lines of high-level language code are flagged appropriately. Flagging is done automatically, when the debug information is loaded together with the program.

Spot

Spot breakpoints are temporary and used solely for saving the register state. The program execution is then continued. Spot breakpoints can be set in both program or data ranges. For program ranges, breakpoints can be set at the beginning of a command only (see **SPot** command).

Read, Write

Both of these breakpoint groups are used for flagging program variables. If the **Read Data** and **Write Data** options are set in the **TrMain.Set** command, then triggering will occur only when data is accessed.

Alpha, Beta, Charly

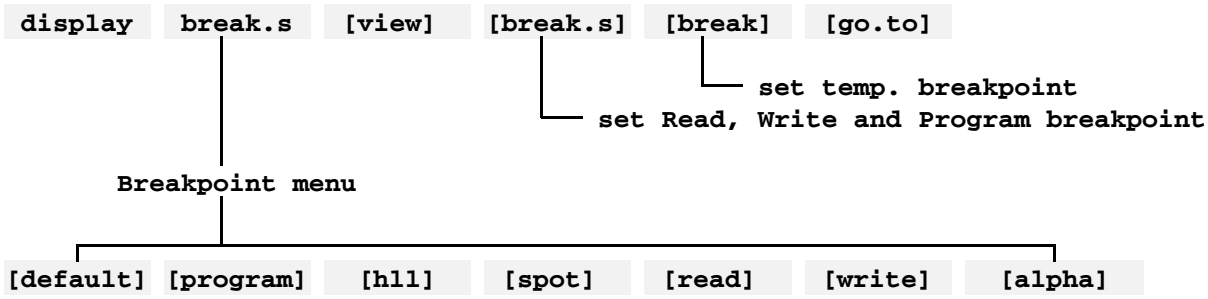
Alpha, Beta und Charly breakpoints are address selectors which can be used either as a breakpoint, or as an address selector for the analyzer. In addition, **Spot**, **Read**, **Write**, **Alpha**, **Beta** and **Charly** breakpoints are used for address flagging in conjunction with the performance analyzer. Thus, 63 non-overlapping memory ranges can be defined. The breakpoint function is disabled whenever the performance analyzer needs the BREAK memory. Program and high-level language breakpoints can be set still.

Set and Delete Breakpoints

Breakpoints can be set on single addresses, address ranges or complex program or data structures. On default the **Read**, **Write** and **Program** breakpoints bits are set or cleared.

Break.Set	Set breakpoints
Break.Delete	Delete breakpoints
Break.SetSec	Set breakpoints on data types
Break.SetFunc	Set breakpoints on functions entries
Break.SetLine	Set breakpoints on HLL lines
Break.SetHll	Set HLL breakpoints
Break.DeleteHll	Delete HLL breakpoints
Var.Break.Set	Set breakpoints on HLL structures
Var.Break.Delete	Delete breakpoints on HLL structures

Breakpoints can be set by the mouse to every address. Press the left mouse button to call-up the address menu:




```
b.s 0x1000                    ; Breakpoint set at address 1000H

b.s 0x10000--0x1ffff          ; Breakpoint set in address range

b.s  var                      ; Breakpoint set at address "var"

b.s var /w                    ; Data write protection breakpoint set
                              ; at address "var"

b.s 0x10000--0x1ffff /w       ; Break at write-to in address range

b.s  main /p                  ; Program breakpoint set at label "main"

b.s  \module\100 /p           ; Program breakpoint set at source line 100

b.s  main, main1 /p           ; Program break at Label "main" or "main1"

b.d                           ; Erase all program and data breakpoints
```

```
b.d 0x1000--0x1fff            ; Erase all program and data breakpoints
                              ; in the range of 1000H to 1fffh

b.d  /all                     ; Erase all breakpoints

b.s  var                      ; Erase breakpoints (P,R,W) at variable
b.d  var /r                   ; main Erase read breakpoints

b.d  /h                       ; Erase all high-level language breakpoints

b.dh INTERRUPT                ; Erase HLL breakpoints in interrupt
                              ; routine

b.d  v.range(INTERRUPT) /h    ; Erase HLL breakpoints in interrupt
                              ; routine

b.d  /s                       ; Reset all spot breakpoints
```

Break.List Display breakpoints

```
E68::w.b.l /def /s /a /b /c
                          C
 C:00000100--00000100      S     \\MWC\MWC\func4_+14
 C:000003D4--000003D4     WR  P  \\MWC\MWC\func10_
 C:0000074C--0000074C     WR  P  \\MWC\MWC\main_
 C:000065CA--000065CA  CBAW S    \\MWC\vshort_
 C:00006658--00006667      R     \\MWC\ast_
 C:0000675E--0000675E      W     \\MWC\flags_
 C:0000675F--0000675F      R     \\MWC\flags_+1
```

The display of the breakpoints is possible with the mouse. Pressing down the left button will show the breakpoint information on the message line. All data windows will display breakpoint information in the scale area.

```
Breakpoints
          D:000008 00 00 00 00 ....
   AW     D:00000C 00 00 00 00 ....
     H    D:000010 39 30 2F 30 90/0
          D:000014 30 2F 30 30 0/00
     H    D:000018 28 26 41 C4 (&A.
          D:00001C 96 FF FF FF ....
          D:000020 FF FF FF FF ....

E::
F:    AW    <-- Breakpoints
```

Temporary breakpoints

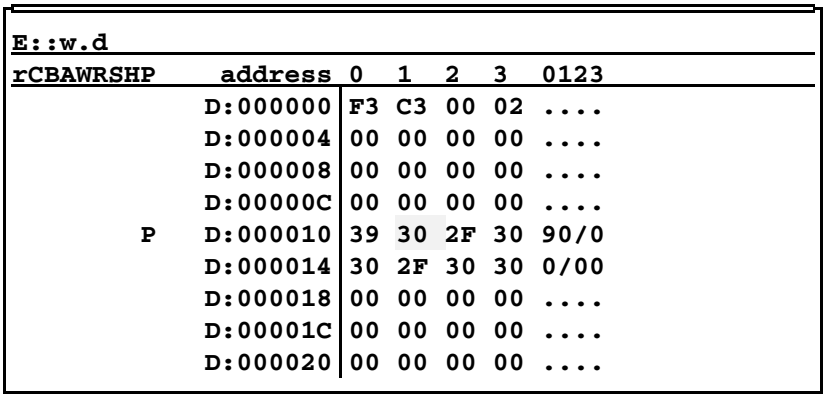
Temporary breakpoints are erased as soon as the program execution stops. They can either be set before, or during program execution. Temporary breakpoints are set by the **Break** command while fixed breakpoints are controlled using **Break.Set** or **Break.Delete** commands.

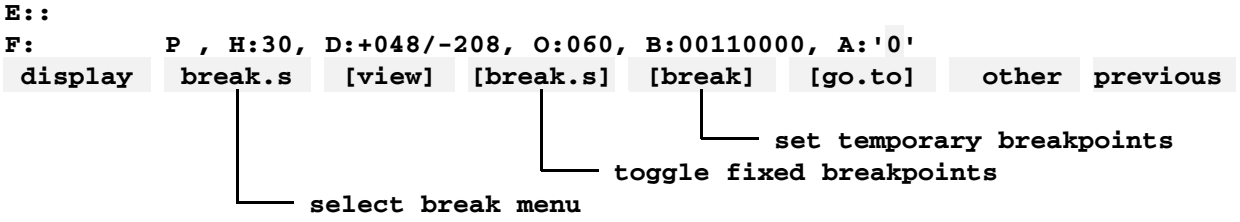
Break	Set temp. breakpoints
Go	Set temp. breakpoints and run
Break.Init	Clear temp. breakpoints
Var.Go	Set temp. breakpoints on variable and run

```
g                       ; Program start
b                       ; Absolute emulation stop

b  0x1000               ; Stop at address 1000H

b  var                  ; Break read variable "var"

b  var /w               ; Break write-to variable "var"

b  main /p              ; Program break at label "main"

b  \main\100            ; Program break at line 100 module "main"

b  main, main1 /p       ; Program break at label "main" or "main1"

b  main /p              ; Set temporary breakpoints
b  main1 /p
b  var /w
g                       ; Start program

v.go flags              ; Run until the array 'flags' is accessed

v.go flags[0..2]        ; Run until the first elements of 'flags'
                        ; are accessed
```

Mouse

The most common way to set breakpoints is using the mouse. Press the left mouse button to select this function, while the mouse cursor is on the place where you like to set breakpoints.



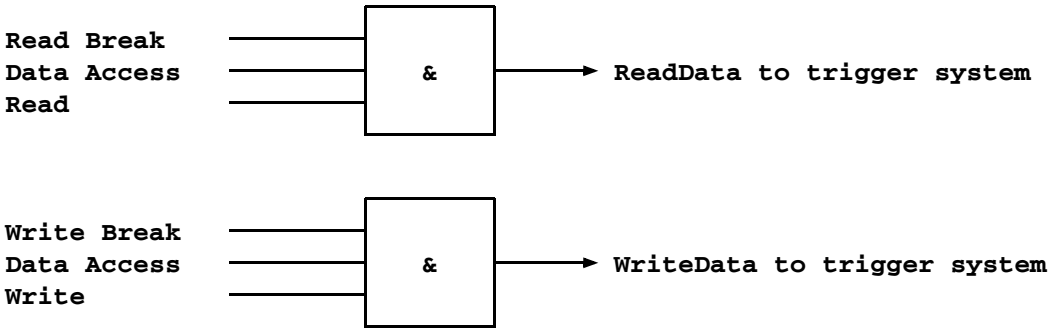


Execution Breakpoints

The program breakpoint logic is placed on the emulation head (ICE-xxxxx). Emulation is stopped without executing the command on the breakpoint address. Program breakpoints are allowed only on the first address of an instruction. Wrong breakpoint setting may lead to malfunction of the emulation system.

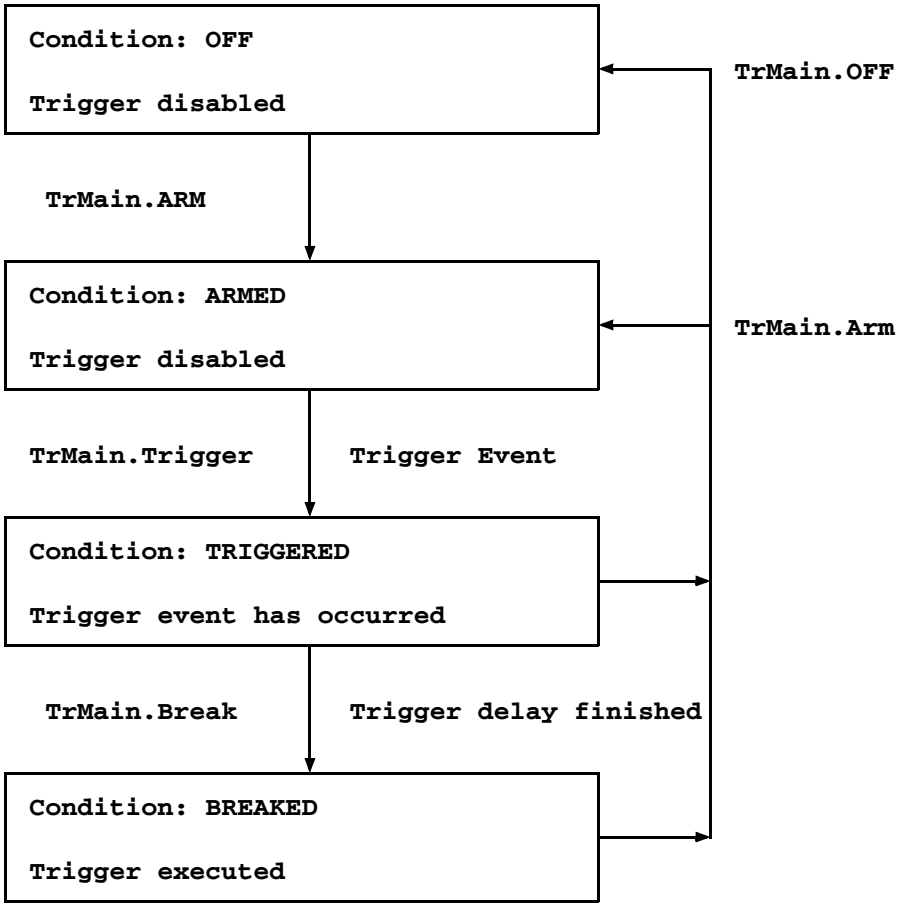
For HLL debugging it is imperative to stop before executing the instruction.

Breakpoints on data areas stop program execution after some CPU instructions. In HLL debugging mode real-time emulation is stopped on the next HLL line. READ and WRITE breakpoints will only be acknowledged if the access cycle is a data access. The settings of the trigger system are valid for data breaks.



Function

The trigger system is used for collecting asynchronous events, converting them into a trigger signal, and passing this trigger signal on to the analyzer and emulation controller. The trigger system can exhibit the following conditions: OFF, ARMED, TRIGGERED or BREAKED. If the trigger system is in the ARMED condition and a trigger event occurs, the TRIGGERED condition will result. After the trigger delay sequence is completed the BREAK state will result.



Trigger Sequence

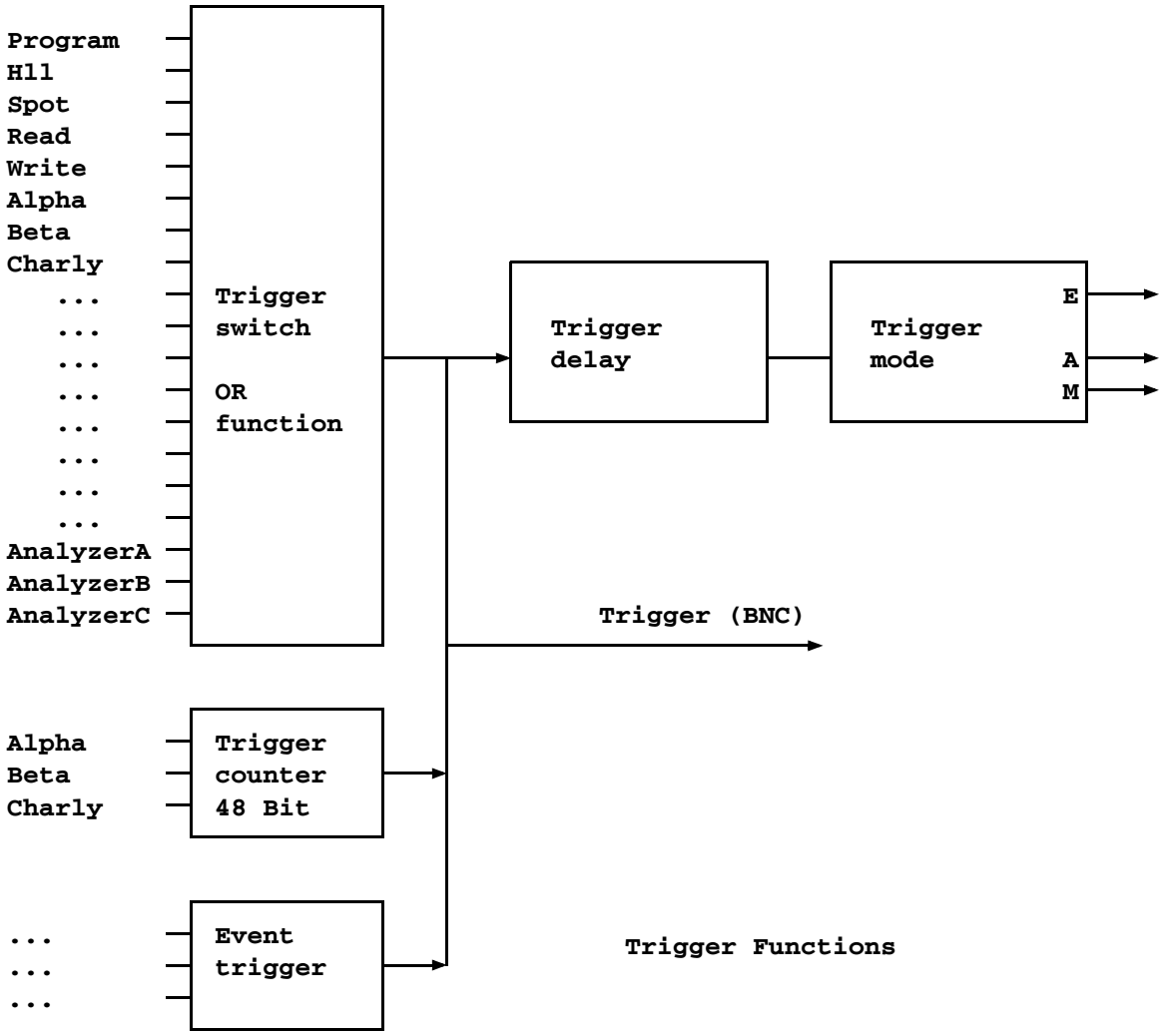
Reaching the break state the emulation system or the analyzer will be stopped.The trigger source and trigger address (not ECC8) will be displayed.

There are three trigger modes:

Emulator Trigger — The emulation system is breaked. The real-time emulation will be breaked at the next HLL or ASM line. If the analyzer slave mode is selected, it will be stopped too.

Analyzer Trigger — The state analyzer is stopped, the real-time emulation is not affected.

Memory Trigger — In memory trigger mode the state of emulation memory is locked (write protection). This function will be useful only if emulation memory is 'shadowed' to the target memory.



NOTE:

The trigger system will be blocked, if the PERF function is activated.

```
E68::w.t
┌────────────────────────────────state──────────────────────────────────┐
│ breaked, triggerd at rd-word UP:000004 direct                         │
└───────────────────────────────────────────────────────────────────────┘
┌trigger─┐  ┌─────────Set──────────┐  ┌─────────────state──────────────┐
│  OFF   │  │ Program     ExtData  │  │ Armed     Triggerd  √ Breaked  │
│√ ALways│  │   H11      ExtSynch  │  └────────────────────────────────┘
│  Init  │  │  Spot       ExtComp  │  ┌─────────────Count──────────────┐
│  RESet │  │  Read                │  │ Alpha   +     0. +         0.  │
│AutoInit│  │  Write    eXception  │  │ Beta    +     0. +         0.  │
│AutoStart│ │  Alpha     TrInput   │  │ Charly  +     0. +         0.  │
└────────┘  │  Beta      Glitch    │  └────────────────────────────────┘
┌─Mode───┐  │ Charly     TimeOut   │  ┌─────────────Delay──────────────┐
│√Emulator│ │           AnalyzerA  │  │√ Cycle  +  1000. -         9.  │
│ Analyzer│ │√ReadData  AnalyzerB  │  │  TRace  +     0. +         0.  │
│  Memory │ │√WriteData   RBW      │  │  TIme   + 0.000    -634.800 µs │
└────────┘  └──────────────────────┘  └────────────────────────────────┘
```

```
E68::
breaked, triggered at rd-word UP:000004 direct
   │              │        │         │
   │              │        │         └──── trigger source
   │              │        └──── trigger address
   │              └──── trigger cycle type
   └──── trigger state
```

TrMain.state	Display trigger state
TrMain.OFF	Disable trigger system
TrMain.ALways	Trigger immediately
TrMain.Init	Initialize trigger system
TrMain.RESet	Reset trigger system
TrMain.Arm	Arm trigger system
TrMain.Trigger	Go to trigger state
TrMain.Break	Go to break state
TrMain.AutoInit	Initialize trigger system on every program start
TrMain.AutoStart	Re-init trigger after break
TrMain.Mode	Set trigger mode
TrMain.Set	Activate trigger source
TrMain.Count	Trigger counter
TrMain.Delay	Trigger delay counter

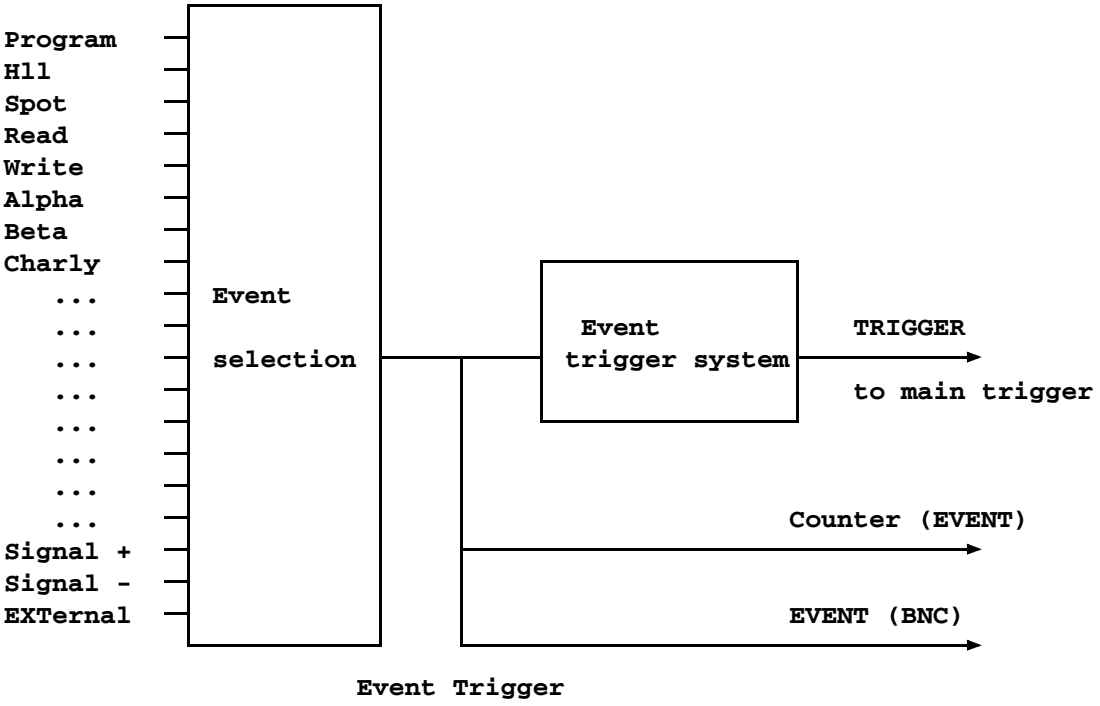
Trigger Sources

Program	Breakpoint P
Hll	Breakpoint H
Spot	Breakpoint S
Read	Breakpoint R
Write	Breakpoint W
Alpha	Breakpoint A
Beta	Breakpoint B
Charly	Breakpoint C
ReadData	Breakpoint R and 'Data Read Cycle'
WriteData	Breakpoint W and 'Data Write Cycle'
ExtData	External trigger input
ExtSynch	External trigger input
ExtComp	External trigger input
eXception	Exception trigger
TimeOut	Trigger on bus time-out
AnalyzerA	Trigger from Analyzer
RBW	Read-before-write trigger

Examples

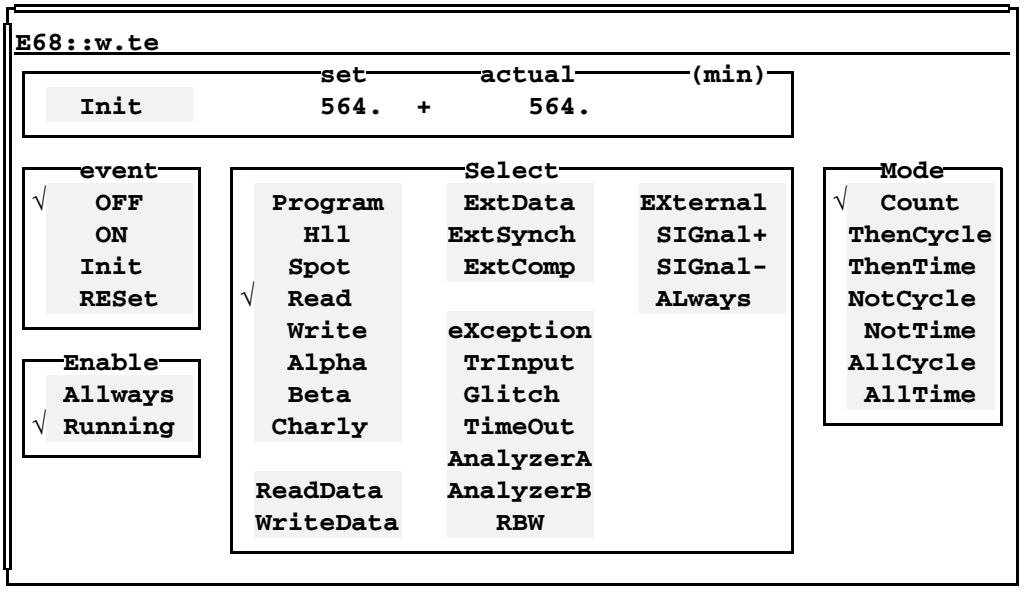
```
t.off                          ; Switch off trigger system

t.arm                          ; Activate trigger system

t.s timeout on                 ; Activate trigger source

t.delay time 10.ms             ; Define trigger delay

t.c alpha 1000.                ; Set trigger counter for alpha
```

Function



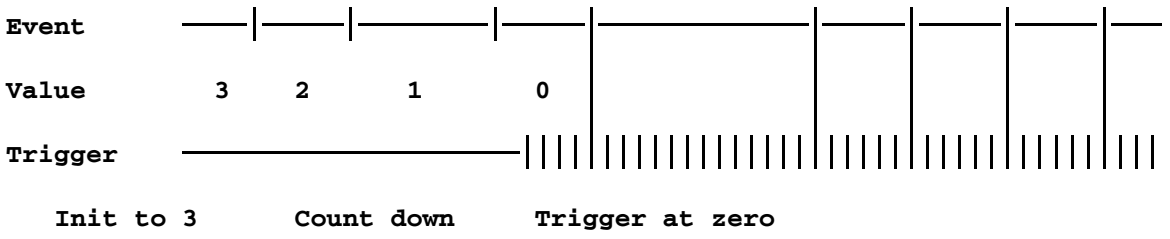
Event Trigger

The event trigger function facilitates the connection between a time counter, or an event counter, and individual trigger events. This function can be useful whenever a trigger should take place after a defined number of events, or when events fail to take place. On ECC8, there is no event trigger system available, but the analyzer trigger system will work in a similar fashion.



Count

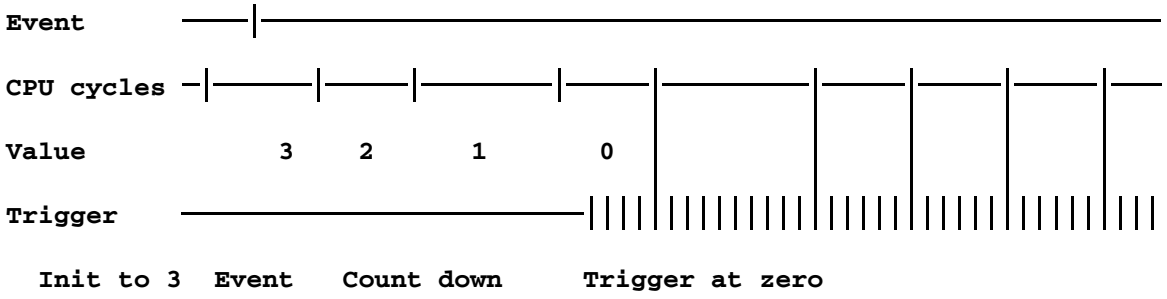
Events are counted, trigger is activated when the event counter becomes zero.




```
; -------------------------------------------------------------------
; Trigger after 1000. rising edges on input probe T0

c.s T0                          ; select input probe by counter
te.s signal+                    ; route counter signal to event trigger
te.m count                      ; use counter function
te.d 1000.                      ; define delay
te.on                           ; arm

; -------------------------------------------------------------------
; Trigger after 1000. accesses to breakpoint alpha

te.s alpha                      ; route counter signal to event trigger
te.m count                      ; use counter function
te.d 1000.                      ; define delay
te.on                           ; arm
```

ThenCycle

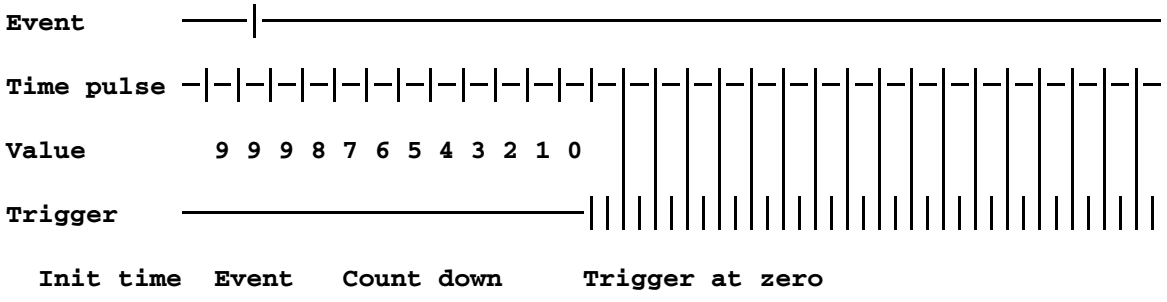
Event is recorded, trigger is executed after a given number of cycles.




```
; ------------------------------------------------------------------
; Trigger 1000 cycles after breakpoint ALPHA is reached

te.s alpha                      ; select breakpoint
te.m thencycle                  ; use delay trigger function
te.d 1000.                      ; define delay value
te.on                           ; arm
```

ThenTime

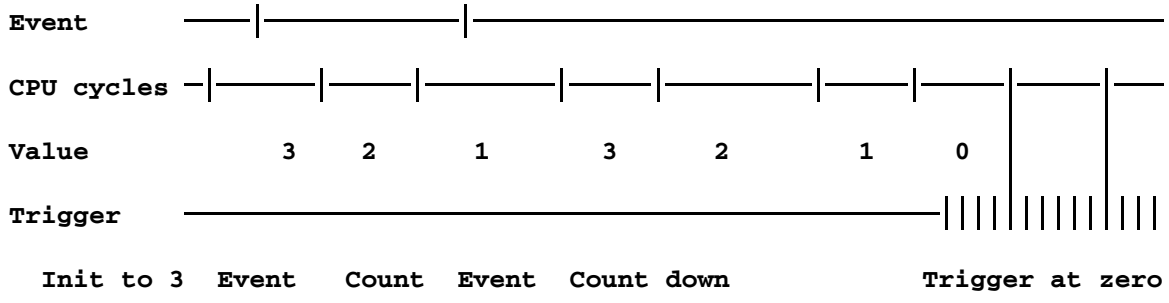
Event is recorded, trigger is executed after the delay time.




```
; ------------------------------------------------------------------
; Trigger 1000 cycles after falling edges on input probe T0

c.s T0                          ; select input probe by counter
te.s signal-                    ; route counter signal to event trigger
te.m thentime                   ; use delay trigger function
te.d 1.ms                       ; define delay value
te.on                           ; arm
```

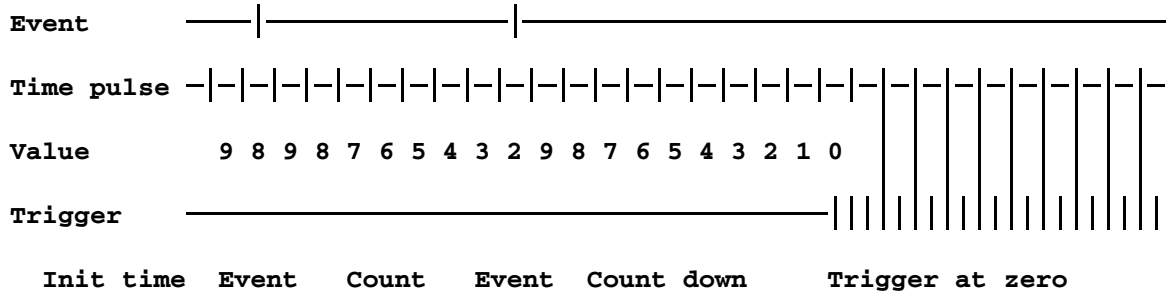
NotCycle

If no event occurs during a certain number of cycles, the trigger signal will be set.



NotTime

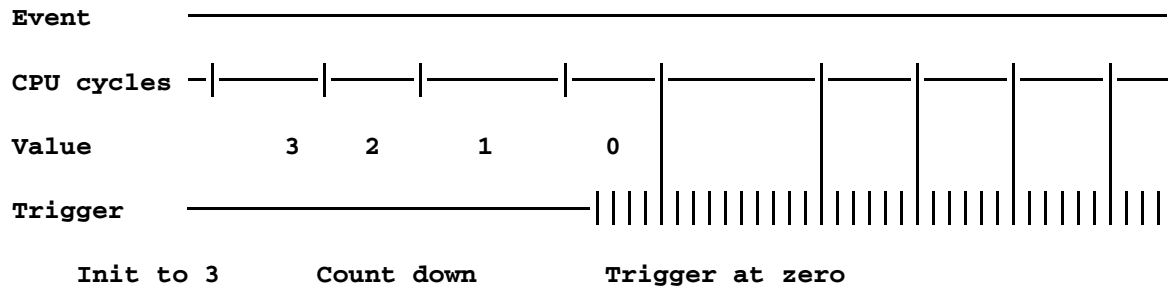
If no event occurs within a specific amount of time, the trigger signal will be activated.




```
; ------------------------------------------------------------------
; Trigger if the distance between two CPU cycles increases 100 us,
; e.g. by DMA cycles or CPU reset

te.m nottime                    ; set mode
te.s always                     ; select cycle signal for retrigger
te.d 100.us                     ; define time-out value
te.on                           ; arm event trigger system

; ------------------------------------------------------------------
; Trigger if the distance between two timer interrupts increases
; 1.5 ms, e.g. if the interrupt will be disabled for long time

te.m nottime                    ; set mode
te.s alpha                      ; select breakpoint
b.s timerinterrupt /a           ; set breakpoint to interrupt routine
te.d 1.5 ms                     ; define time-out value
te.on                           ; arm event trigger system
```

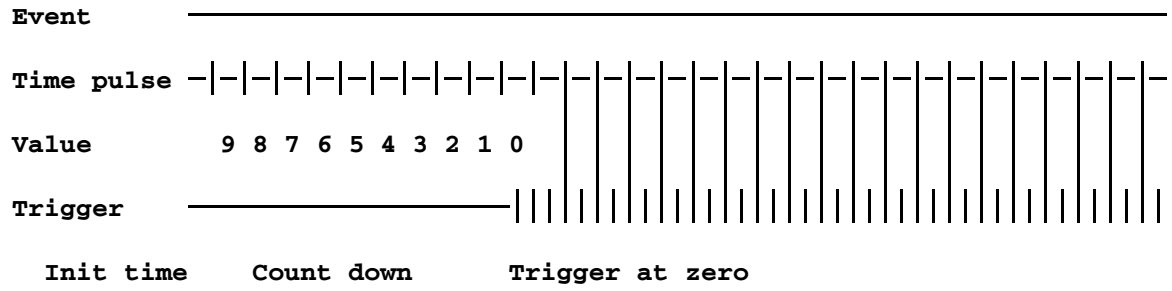
AllCycle

Trigger is always activated after a certain number of CPU cycles.



AllTime

Trigger is always executed after a fixed time.

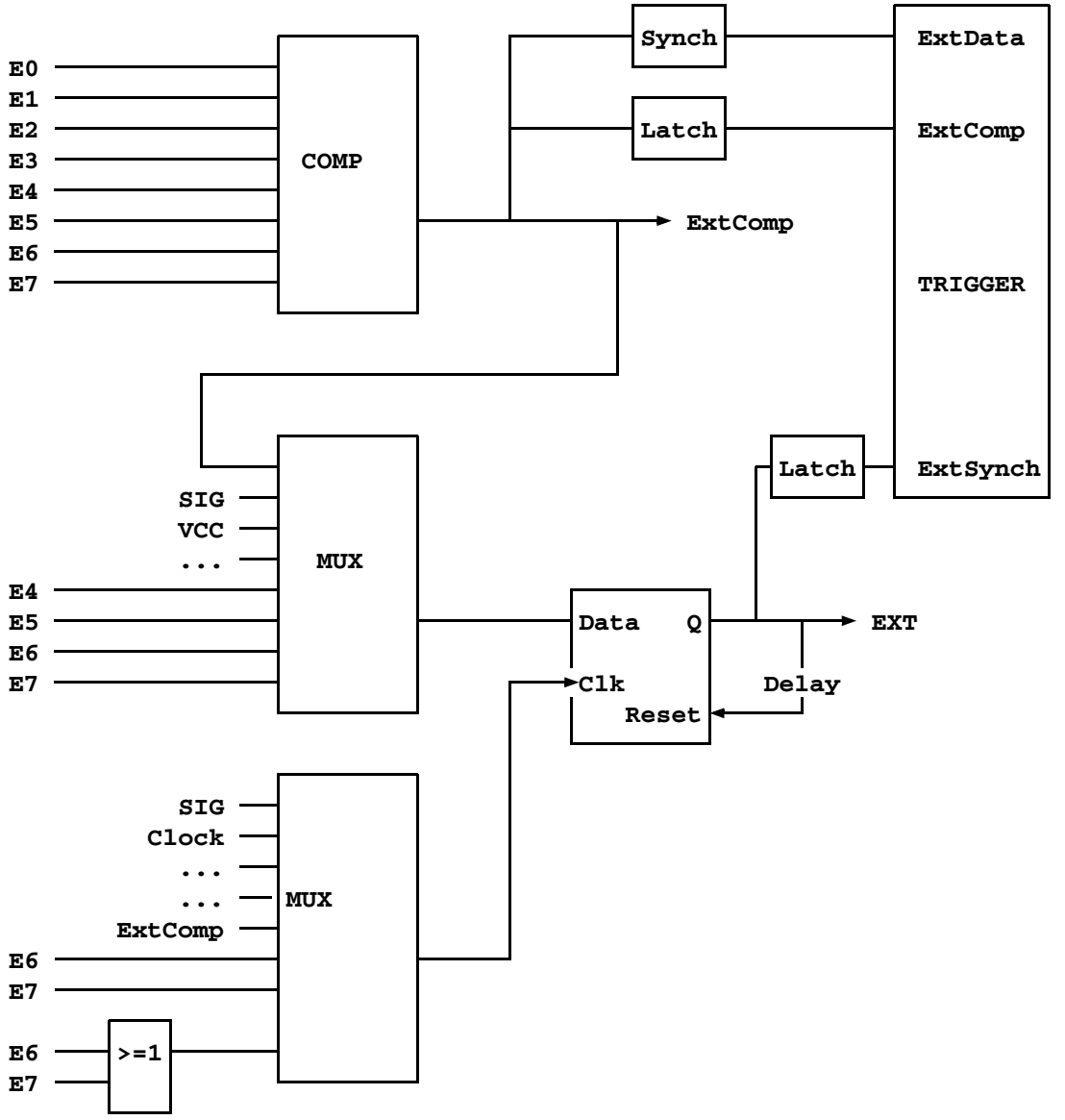



```
;--------------------------------------------------------------------
; Run for a fixed time

te.m alltime
te.delay 1000.s
te.on
go
...
```

TrEvent.view	Show state
TrEvent.RESet	Reset event trigger system
TrEvent.Init	Initialize event trigger system
TrEvent.OFF	Switch off
TrEvent.ON	Activate event trigger
TrEvent.Mode	Set trigger mode
TrEvent.Select	Select trigger signal
TrEvent.Enable	Define activation time
TrEvent.Delay	Define delay value

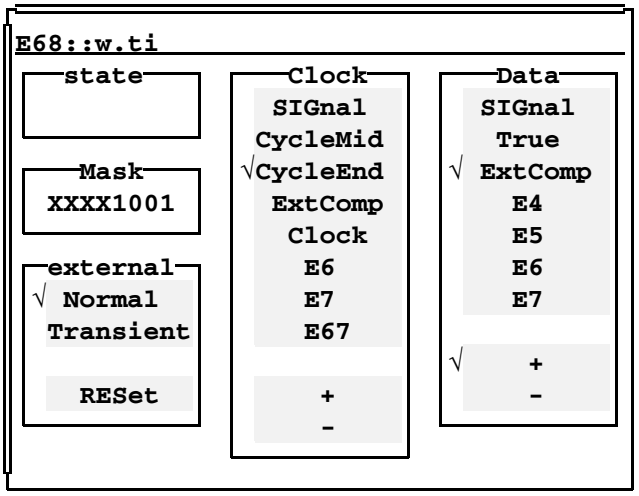
```
; Trigger if timer tick is missing

b.s timer_tick /a              ; set alpha breakpoint to system timer
t.m nottime                    ; set event trigger to timeout mode
t.d 1.2ms                      ; define elapsed time
t.e on                         ; activate event trigger

; Trigger after 1000 accesses to breakpoint ALPHA

te.s alpha                     ; select breakpoint ALPHA
te.m count                     ; count mode
te.d 1000.                     ; event counter
te.on                          ; switch-on

; Trigger after 1000 edges to probe EXTERNAL

te.s ext                       ; select breakpoint ALPHA
te.m count                     ; count mode
te.d 1000.                     ; event counter
te.on                          ; switch-on

; Trigger if breakpoint ALPHA does not respond at least every
; 10 ms

te.s alpha                     ; select breakpoint ALPHA
te.m nottime                   ; mode
te.d 10.ms                     ; set delay
te.on                          ; switch-on

; Trigger 10 ms after breakpoint ALPHA

te.s alpha                     ; select breakpoint ALPHA
te.m thentime                  ; mode
te.d 10.ms                     ; set delay
te.on                          ; switch-on
```

Function



The **TriggerIn** function can be used in conjunction with the EXTERNAL probe. It allows to combine external asynchronous or synchronous events and to synchronize these with the internal trigger logic. The trigger logic consists of a programmable comparator, combined with a D-flip-flop with delayed reset for generating a short pulse. Flip-flop data and clock input can be switched to any probe input, internal signals or comparator output. The output of the flip-flop (named EXT), as well as the output of the comparator (named ExtComp) can be used as a trigger event. These signals can also be routed directly to the counter and the event trigger unit for complex triggering. Synchronized to the CPU cycle the state analyzer can trigger on these events. In addition, a "transient" mode is available and can be triggered by status changes to individual input signals.

An EXTERNAL probe is not available on ECC8.



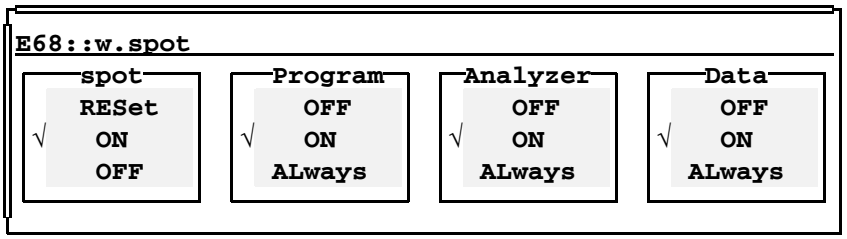
Setup

TrIn.state	Show state
TrIn.RESet	Select reset configuration
TrIn.Clock	Select synchronuous clock
TrIn.Data	Select data source
TrIn.Mask	Define comparator mask
TrIn.Transient	Set trigger mode

```
; ----------------------------------------
; trigger on rising edge of E6 input line

ti.res
ti.data true                    ; unqualified data
ti.clock E6 +                   ; select clock
t.s extsynch on                 ; activate trigger input

; ----------------------------------------
; trigger on high level of E6 input line

ti.res
ti.data E6                      ; select data bit
ti.clock cyclemid               ; select internal clock
t.s extsynch on                 ; activate trigger input

; ----------------------------------------
; trigger if E1 is low and E0 is high at
; the end of the cycle

ti.res
ti.mask 0x0xxxxxx01             ; define mask for E0 and E1
ti.data extcomp                 ; select comparator for data input
ti.clock cycleend               ; select internal clock
t.s extsynch on                 ; activate trigger

; ----------------------------------------
; trigger if E1 is low and E0 is high at
; the end of the cycle

ti.res
ti.mask 0x0xxxxxx01             ; set trigger mask
t.s extdata on                  ; route comparator to trigger
                                synchronously

; ----------------------------------------
; trigger if E1 and E0 are high for a short time

ti.res
ti.mask 0x0xxxxxx11             ; set comparator mask
t.s extcomp on                  ; activate asynchronous trigger

; ----------------------------------------
; trigger if E0 and E1 are low at the falling edge of E6

ti.res
ti.mask 0x0xxxxxx00             ; define mask for data
ti.data extcomp                 ; select comparator
ti.clock E6 -                   ; select clock line and polarity
t.s extsynch on                 ; activate standard trigger

; ----------------------------------------
; trigger on every transient on input E0 and E1
```

```
ti.res
ti.mask 0xxxxxx00              ; set mask for transient detection
ti.transient                   ; set mode
t.s extdata on                 ; set trigger synch. to CPU cycles

; ----------------------------------------
; trigger if no clock edge arrives within 1.ms on E6

ti.res
ti.data true                   ; no data qualifier
ti.clock E6 +                  ; set clock signal to E6
te.select ext                  ; select EXT for event trigger
te.mode nottime                ; set mode for event trigger
te.delay 1.ms                  ; set delay
te.on                          ; activate trigger

; ----------------------------------------
; count the clock edges on E6 while E0 and E1 are both low

ti.res
ti.mask 0xxxxxx00              ; set mask for data qualifier
ti.data extcomp                ; select comparator
ti.clock E6 +                  ; select clock
count.select ext               ; route signal to counter
count.mode eventhigh           ; select counter mode
...
count.go                       ; read-out counter value
print "Cycles   "
count.value()
...
```

Function



The spot system is used to temporarily halt the real-time program execution in order to get control over the CPU to display memory or internal CPU registers. Spot breakpoints can be defined in both program and data areas. The analyzer can also be used to trigger short breaks. Spot breakpoints are defined using the **Break.Set /Spot** command and are erased by means of the **Break.Delete /S** command. The three spotpoint sources, **Program**, **Data**, **Analyzer** can be enabled or disabled. If the switch is set to the **ALways** position the program will be halted each time a spot breakpoint is encountered and resumed after all windows are refreshed.

ON With the switch set to **ON**, the program will be interrupted only if there is a task for the emulation CPU. Emulation speed (performance) with the system switched to **ON** is about 1%-10% slower than without spotpoints. Each program stop takes aprox. 200 to 1000 microseconds. With slower CPU types or CPU types with background-debug interface (like MC68332) the performance impact is higher.

ALways In **ALways** mode the emulation is stopped until all windows are updated. This will normally take about 10 to 200 milliseconds, depending on the number and the complexity of the windows. The performance in this mode will go down by some orders of magnitude, depending on the time between two spotpoints.

The advantage of the SPOT function - compared to the general break function (**Break.SELect FORE**) - is that the break takes place at a specifically defined point within the program. For example, if a spot breakpoint is set to data area it will be assured that the program window shows this partition of the program where the data transfer is executed. Spotpoints set to program lines will allow displaying local variables in a running target program.

Setup

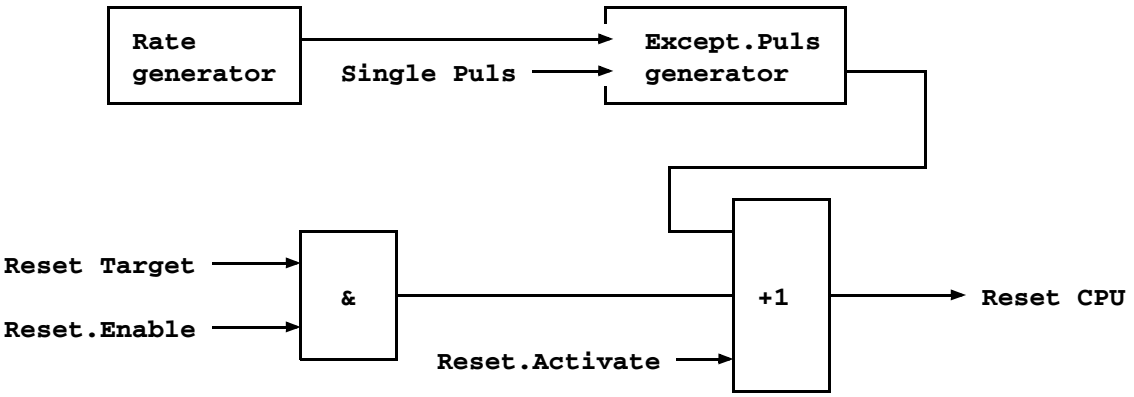
SPot.ON	Enable SPOT system
SPot.OFF	Disable SPOT system
SPot.RESet	Initialize SPOT system
SPot.Program	Program spot point
SPot.Data	Data spot point
SPot.Analyzer	Analyzer spot point
SPot.state	State window
SPot.Test	Set spotpoint

Examples

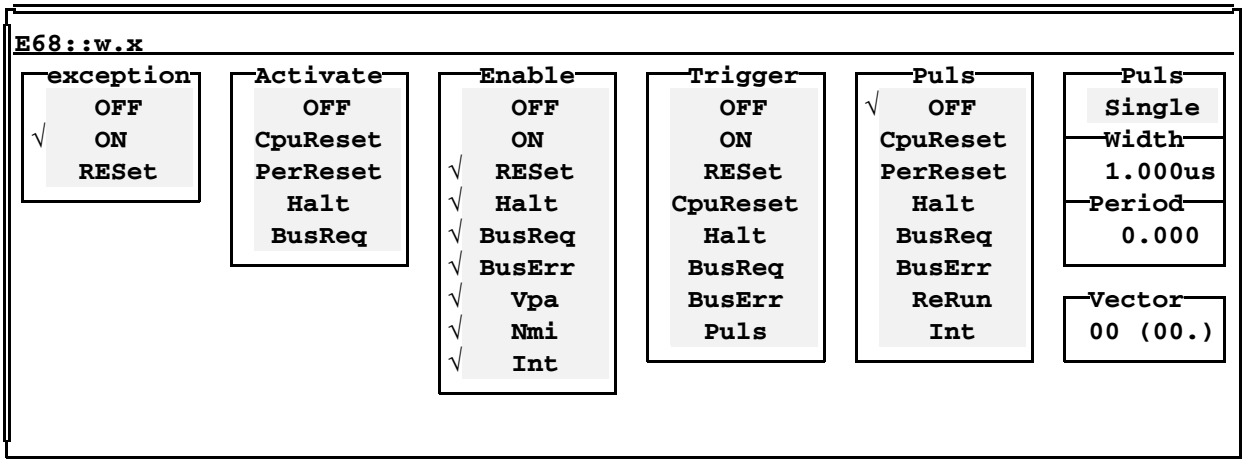
```
b.s sieve /s                    ; set spot point
w.r                             ; display registers on spot point

b.d sieve /s                    ; delete spot point

b.s 0x1000--0x10ff /s           ; set spot on data buffer
spot.always on                  ; update windows on every access
                                ; to buffer
```

Function

Exception commands are used to control and simulate all special CPU lines like RESET or interrupt inputs. This is especially useful during the development phase, allowing disabling of CPU input lines, or to simulate certain events. All signals are enabled during real-time emulation only. The exception generator works only while foreground emulation is running (i.e. the CPU must generate bus cycles). The exception system cannot control interrupts generated by internal peripherals in microcontrollers.



Structure of RESET control



eXception.state	Display state
eXception.Enable	Enable exception lines
eXception.Activate	Stimulate exception lines
eXception.Trigger	Trigger on exception lines
eXception.Puls	Route pulse generator to exception lines
eXception.Width	Define pulse width
eXception.PERiod	Define pulse period
eXception.Single	Force one exception pulse
eXception.Delay	Delayed interrupt enable
eXception.ON	Enable exception system
eXception.OFF	Disable exception system
eXception.RESet	Initialize exception system

```
; BUSREQ every 1 ms
x.e off                          ; disable all external lines
x.p busreq 10.us 1.ms
```

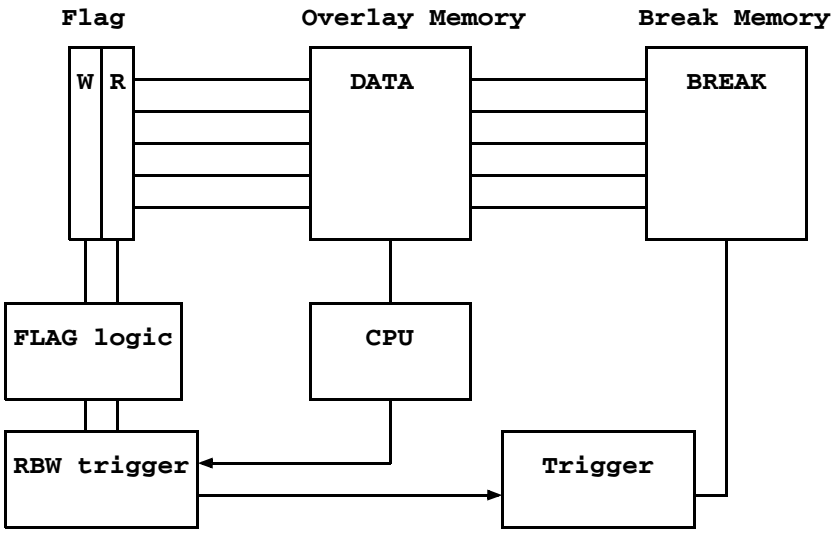
```
; Vector Interrupt (Single Mode)

x.v 20.                          ; vector = 20
x.p int 10.us                    ; single interrupt
g                                ; real-time emulation
x.single                         ; trigger interrupt simulation

;    Cyclical Reset

x.p cr 100.us  1.s               ; program generator
g                                ; start program
```

```
; Trigger on BUSREQ
x.t  br on

; Trigger on all interrupt vectors from 128 to 255
x.t  128.--255.  on

; Trigger on address error (vector 3)
x.t  3.  on

; Disable all vector associated interrupts
x.t 0.--255.  off

E::x.delay 50.ms       ; block all interrupts for the first 50 millisec.
```

For more detailed information on special probe functions refer to the **emulation probe manual**.

Function

The flag system is an excellent tool for the analysis of complex software systems. FLAG memory is divided into 4 KByte blocks, and can be mapped into the CPU's entire external bus address range. A read-flag, as well as a write-flag is available for each address. The read-flag is set at memory read and OP-FETCH cycles (prefetch too), whereas the write-flag is set for all write-to memory operations. After power-up, or execution of the **MAP.RESet** command, no flag memory is mapped. FLAG memory is activated by means of the **MAP.Flag** or **MAP.RAM** command. It can, however, also be mapped-to using the **FLAG.Set** command if the **Map** option is selected.



FLAG system and READ-BEFORE-WRITE trigger

Before restarting a program all flag-memory cells should be reset (**FLAG.Delete** or **FLAG.Init** commands). Evaluating which cells were accessed can be done by means of the **FLAG.List** command. However, the flag memory state is

displayed whenever any **DATA command** is executed.

If no flag memory cells are changed, the flag system will be enabled or disabled by using the **FLAG.ON** or **FLAG.OFF** options.

The flag memory is dual-ported so that analysis can also be carried out during real-time emulation.

Applications

1. Stack depth
2. Code coverage analysis
3. Checking variables (READ/WRITE)
4. Unused variables (WRITE-ONLY)
5. Uninitialized variable triggering (READ ONLY/ READ-BEFORE-WRITE)

Problems

The flag system is not able to decide if the prefetch is executed or not. Therefore short skips are not detected as not executed code and prefetches at function end set some flag bits in the next function. Data flags are not correctly set, if the CPU cache function is activated (disable for data analysis).

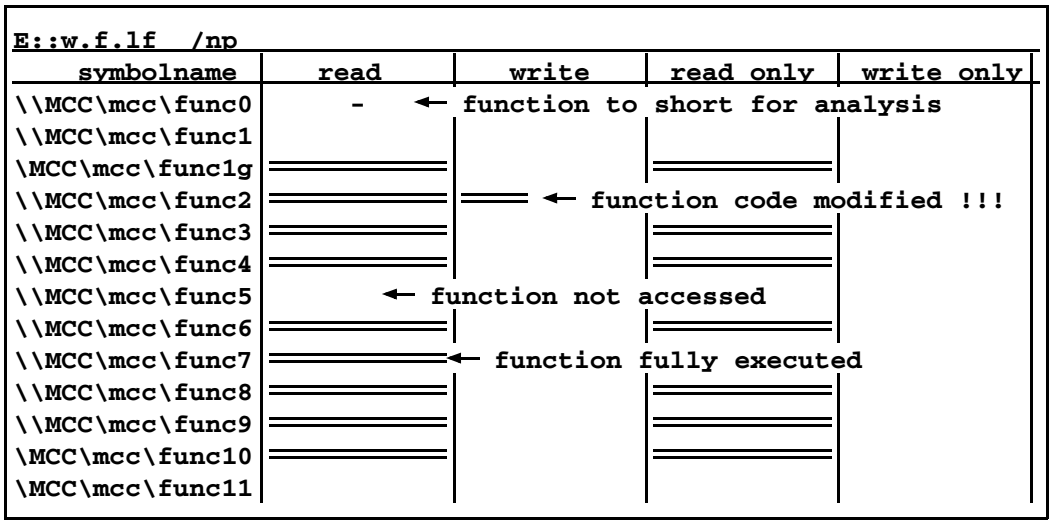
FLAG Control

FLAG.state	Display state
FLAG.ON	Enable flag system
FLAG.OFF	Disable flag system
FLAG.Set	Set flags
FLAG.Delete	Delete flags

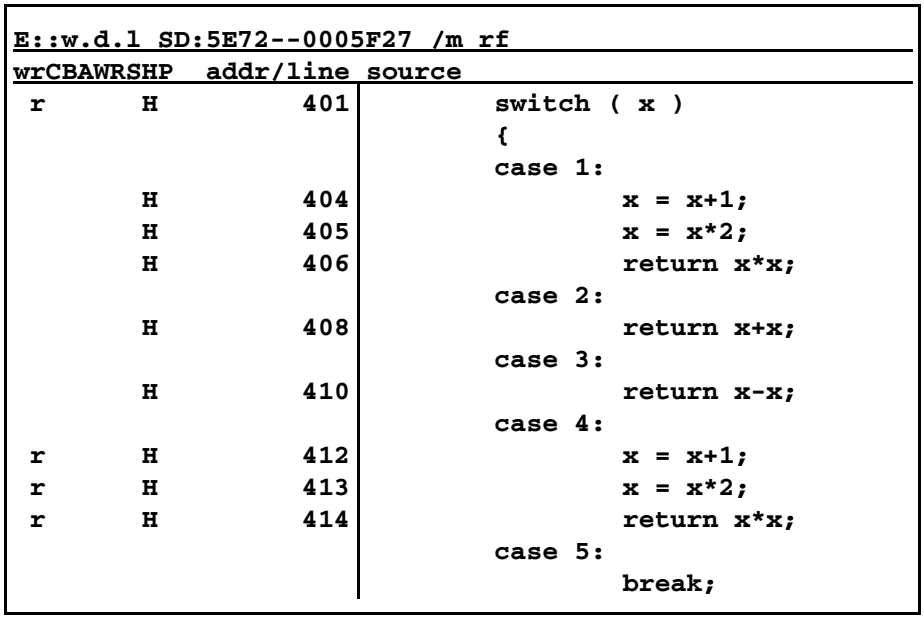
FLAG.List	List flags
FLAG.ListFunc	List flags on function level
FLAG.ListVar	List flags on variable level
FLAG.ListModul	List flags on module level

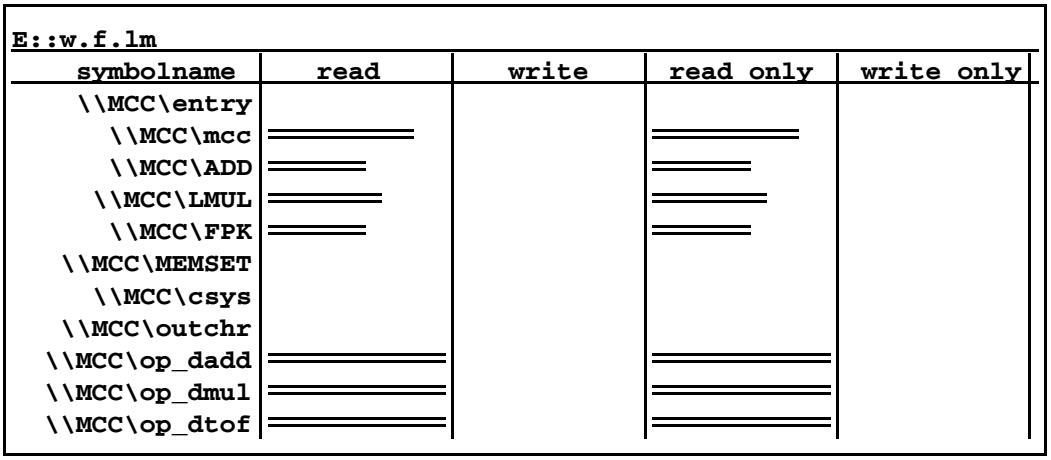
```
E68::w.f.l
                                  C
 C:0000645E--00006461                    \\MWC\splimit+4E24
 C:00006462--00006469              w     \\MWC\splimit+4E28
 C:0000646A--00006471             rw     \\MWC\splimit+4E30
 C:00006472--0000648D                    \\MWC\environ_
 C:0000648E--00006495             rw     \\MWC\I_a_arena_
 C:00006496--00006771                    \\MWC\I_a_block_+4
 C:00006772--0000FF03              w
 C:0000FF04--0000FF0B             rw
 C:0000FF0C--0001FFFF              w
 C:00020000--00413FFF          noRam
```

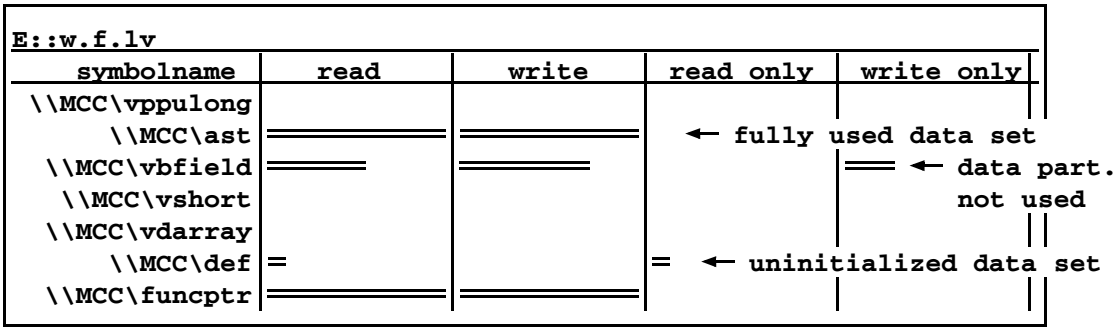
```
E::w.f.lf
    symbolname      read        write       read only     write only
\\MCC\mcc\func0
\\MCC\mcc\func1
\MCC\mcc\func1g     ========    ========    ========
\\MCC\mcc\func2     ========    ========    ========
\\MCC\mcc\func3     ========    ========    ========
\\MCC\mcc\func4     ========    ========    ========
\\MCC\mcc\func5     =  <- function prefetched
\\MCC\mcc\func6     ========    <- function fully executed
\\MCC\mcc\func7     ========    ========    ========
\\MCC\mcc\func8     ========    ========    ========
\\MCC\mcc\func9     ========    ========    ========
\MCC\mcc\func10     ========    ========    ========
\MCC\mcc\func11     ======      <- function partionally executed
MCC\mcc\func11a                 <- function not accessed
```



More detailed information is available with a short click in the data section of the window.





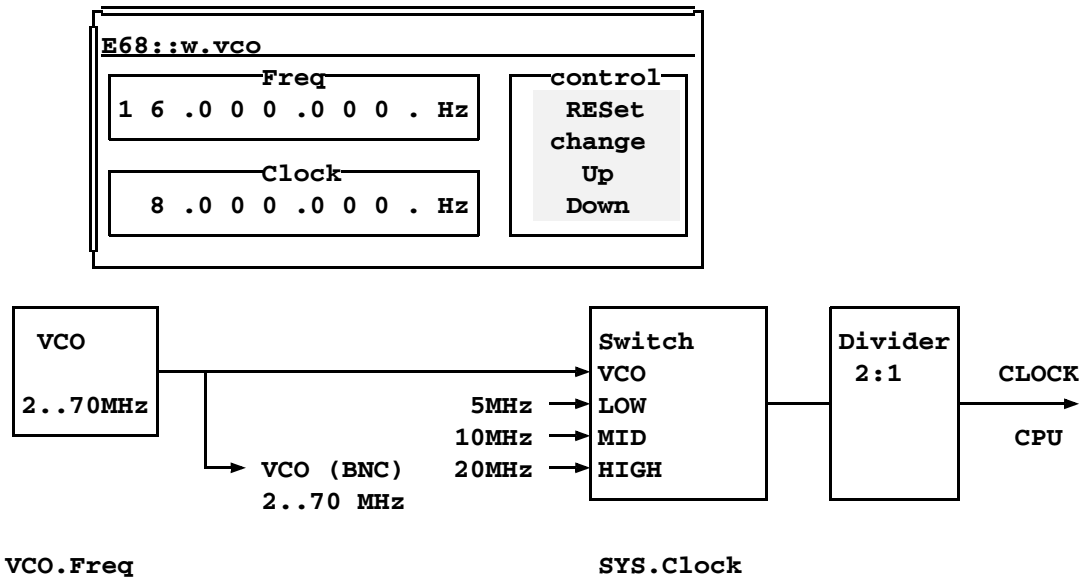


More detailed information is generated when a short click in the data section of the window was done:

```
E::w.d.p %var v.range(vbfield) /m rf
wrCBAWRSHP     address  data          value              symbol
wr             SD:002554 FF           vbfield.a / .b / .c / .d \\MCC\vbf
wr             SD:002555 FF           vbfield.d / .e     \\MCC\vbfield+1
wr             SD:002556 F8           vbfield.e          \\MCC\vbfield+2
wr             SD:002557 00           vbfield            \\MCC\vbfield+3
wr             SD:002558 FF           vbfield.f          \\MCC\vbfield+4
wr             SD:002559 FF           vbfield.f / .g     \\MCC\vbfield+5
wr             SD:00255A FF           vbfield.h          \\MCC\vbfield+6
wr             SD:00255B FF           vbfield.h / .i     \\MCC\vbfield+7
w              SD:00255C FF           vbfield.j          \\MCC\vbfield+8
w              SD:00255D FF           vbfield.j          \\MCC\vbfield+9
wr             SD:00255E E0           vbfield.k / .l     \\MCC\vbfield+0A
               SD:00255F 00           vbfield            \\MCC\vbfield+0B
w              SD:002560 FF           vbfield.m          \\MCC\vbfield+0C
w              SD:002561 FF           vbfield.m          \\MCC\vbfield+0D
               SD:002562 00                              \\MCC\vbfield+0E
               SD:002563 00                              \\MCC\vbfield+0F
w              SD:002564 FF           vbfield.n = -1     \\MCC\vbfield+10
```

Function

The VCO may be used as a clock generator to support the emulation CPU, or may be used separately from the emulation system. The frequency range is from 2 to 50 (70) MHz. In response to the emulation adapter this frequency is divided by a fixed rate generating the CPU clock. The frequency control is made either on VCO or CPU clock frequency level. The step rate is 50 kHz on the VCO level. The VCO clock is ready on the BNC connector in rear of the ECU32 or ECC8 module.

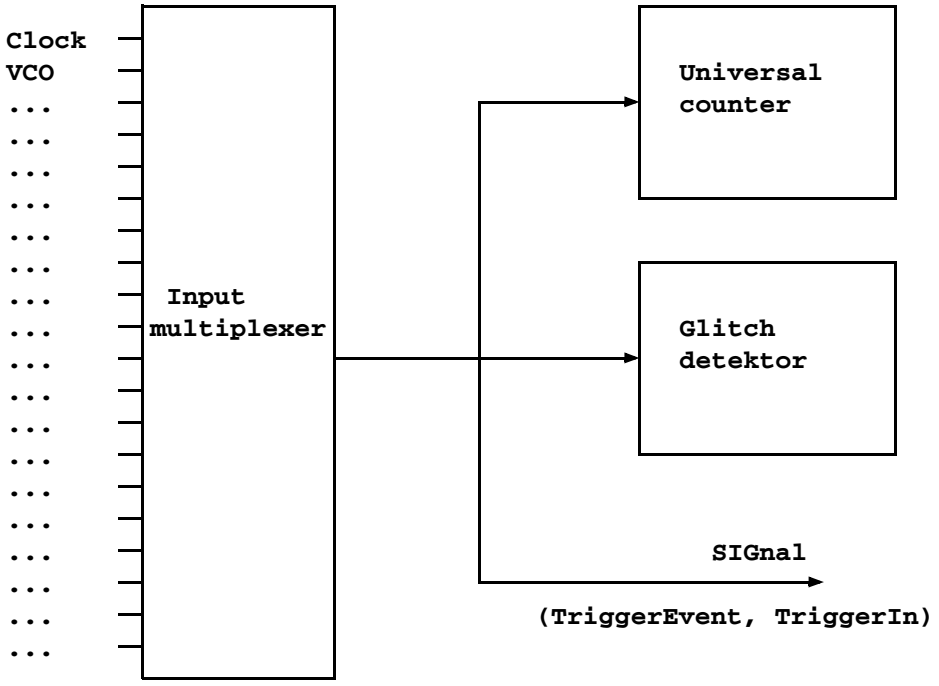


Schematic of clock generation, when using internal clock

VCO.state	Display frequency and mode
VCO.Frequency	Set VCO frequency
VCO.Clock	Set CPU clock frequency
VCO.Up	Increment frequency
VCO.Down	Decrement frequency
VCO.RESet	Initialize VCO

Function

The universal counter is the logic measurement system for sampling of pulses and frequencies. The input multiplexer enables the counter to measure all important CPU lines and all external probe inputs. Therefore the counter input normally need not be hard wired to the signal. Together with the port analyzer all peripheral pins of microcontroller chips can be attached.



Principle of Universal Counter and Glitch Detector

The count ranges are:

frequency:	0 .. 20 MHz
CPU clock:	0 .. 80 MHz
VCO:	0 .. 80 MHz
Puls width:	100 ns .. 300 Days
Period:	100 ns .. 300 Days
Events:	2.8 * 10E+14 max. rate 10 MHz

The input signal is selected with the function **Count.Select**. The function **Count.Mode** is used to change the counter mode and the **Count.Gate** function defines the gate time. Frequency and event analyzing may be qualified by the foreground running signal.

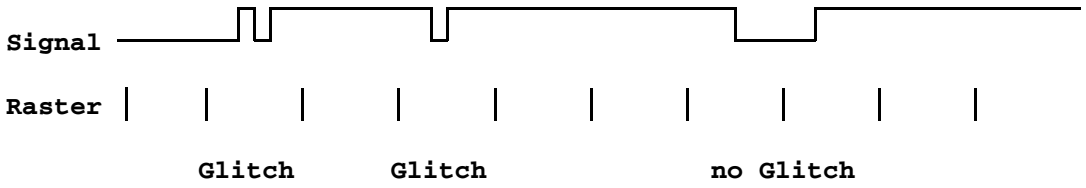
If there is no event counting, it will be possible to activate more than one count window. Every window represents a separate counter. For example it is possible to check the clock frequency and the puls width on some probe inputs simultaneously.

Level Display

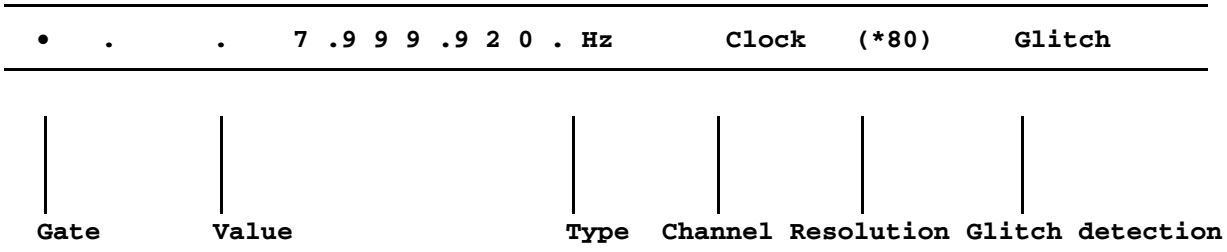
If there is no signal (frequency is zero), the level of the input signal will be displayed (high or low level). The threshold level is defined by the input probes (fixed to 1.4 V or variable).

Glitch Detection

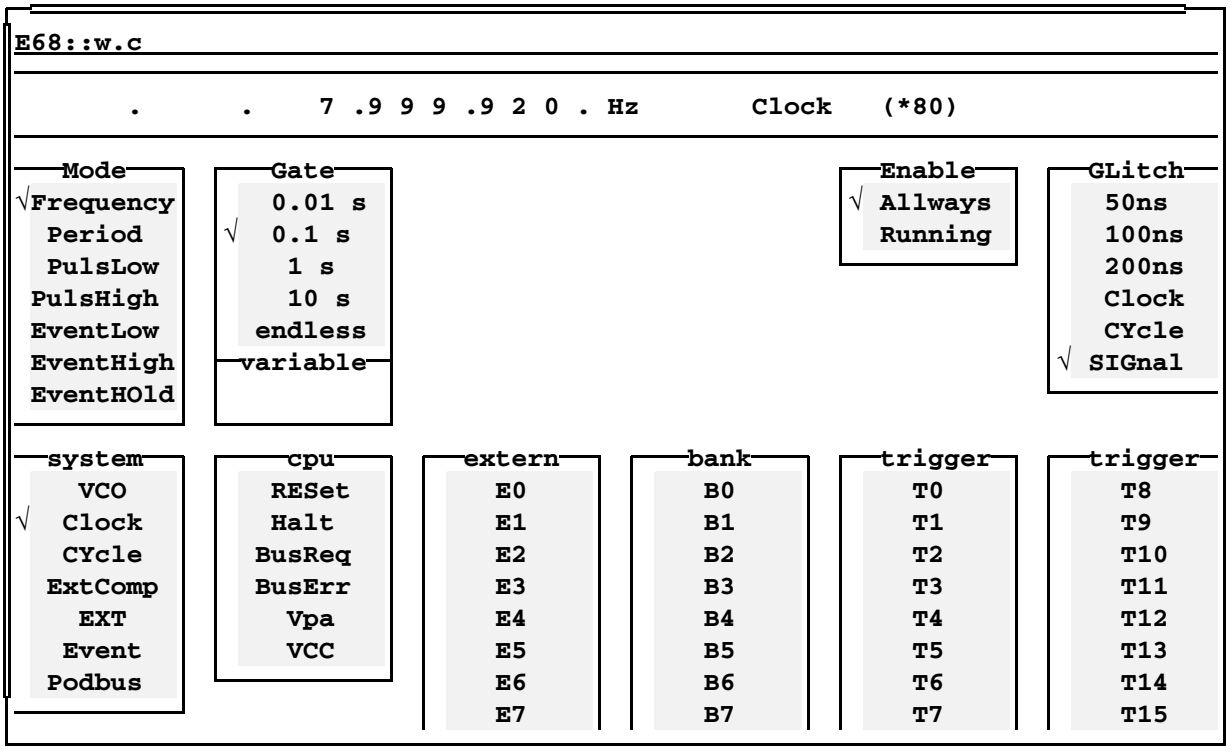
A special glitch detection circuit will check the measurement signal, if there is more than one edge between two measurement points. In a microprocessor based system some signals are only allowed to change one time within a CPU cycle. If there are more edges, this usually is defined as a hardware or design problem. The time raster for glitch detection may be fixed or generated by the signal. The glitch detection is stored and reset on programming the counter again. The glitch detector can be used to trigger the emulator (command **TrMain.Set Glitch**).



Display Window



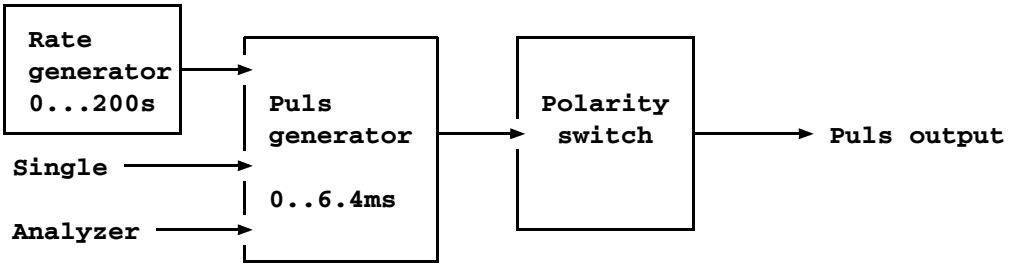
Count.state	Display value and setup
Count.RESet	Initialize counter setup
Count.Init	Initialize counter
Count.Mode	Select counter mode
Count.Select	Select input signal
Count.Gate	Select gate time
Count.Enable	Select run time
Count.Glitch	Set mode of glitch detector
Count.PROfile	Display profile



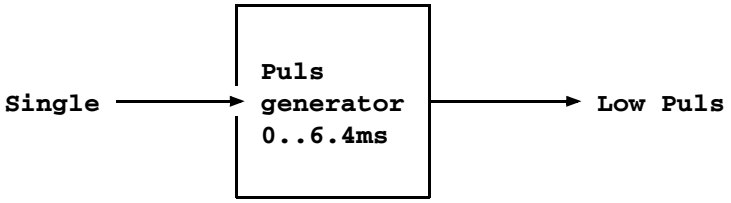
```
; test xtal frequency
c.mode f
c.s clock
c.gate 0.1s
c.go
if count.value()<1000000.
     print "Oscillator Error"

; test reset line
c.mode f
c.s reset
c.go
if count.value()!=0.
     print "RESET floating"
if count.level()==0x0
     print "RESET activ"

; test puls width on trigger probe
name.t0 test-
c.m pulslow
c.s test-
c.go
if count.value()<10.us
     stop "Puls too short"
if count.value()>11.us
     stop "Puls too long"
```

Function

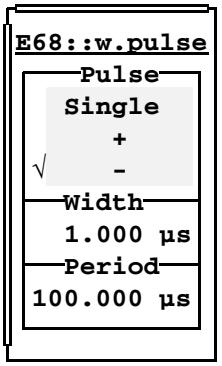
The pulse generator is an independent system for generating short pulses or static signals, and can be used for stimulation in the target system or to reset hardware. The output pin of the generator is placed on the output probe of the ECU module. The triggering can be done periodically, manually from the keyboard or by the trigger unit of the analyzer.



Puls Generator on ECU32



Puls Generator on ECC8



Bottom for single pulse
Polarity

Pulse width

Cycle duration for periodic release

```
E68::
width:  1.000 µs period:100.000 µs pol: -
```

Setup

PULSE.RESet	Reset pulse generator
PULSE.Width	Define pulse width
PULSE.Puls	Define pulse width and polarity
PULSE.PERiod	Define period
PULSE.Single	Execute single or multiple pulse
PULSE.state	Display setup

Examples

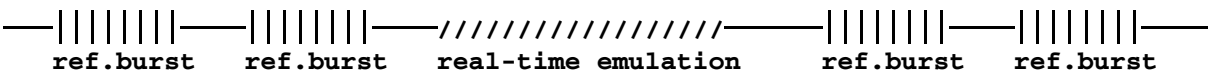
```
pulse.pulse  100.us 1.ms -     ; Pulse active low, 100 µs, 1 kHz

pulse.pulse  100.us +          ; Single pulse 100 µs, active high

pulse -                        ; Active low pulse

pulse off                      ; Switch off
pulse -                        ; Set output to high level

pulse +                        ; Set output to low level

pulse.width  20.u              ; Set pulse width to 20 µs

pulse.w  5.ms                  ; Set pulse width to 5 ms
pulse.per  off                 ; Set pulse generator to single pulse mode

pulse.per  on                  ; Set pulse generator to periodic pulse
                               ; mode

pulse.per  1.ms                ; Activate periodic mode, cycle duration
                               ; is 1 ms  (1 kHz)
```

Function

The refresh function has to be activated when a dynamic RAM is on the target system and is refreshed by the CPU. The refresh function uses burst read cycles for refreshing the target memory. Address range and storage class can be defined.

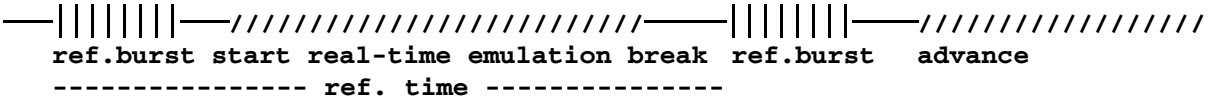
StandBy

On **StandBy** mode the refresh burst is forced while no emulation is running.

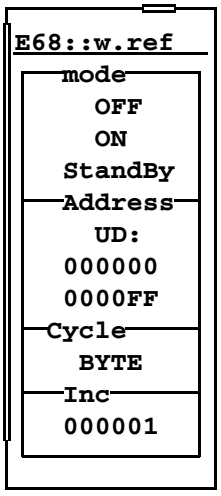


On

When using the **ON** mode a cyclic refresh burst will be made, if real-time emulation is active.



NOTE: This command is for refreshing target DRAM's only. When dynamic emulation memory is present, it is automatically refreshed by dual-port access.



Setup

REFresh.state	Display state
REFresh.RESet	Reset refresh function
REFresh.OFF	Disable refresh function
REFresh.StandBy	Standby refresh
REFresh.Address	Define address range
REFresh.Inc	Define increment
REFresh.CYcle	Define bus cycle
REFresh.Time	Define repetition rate

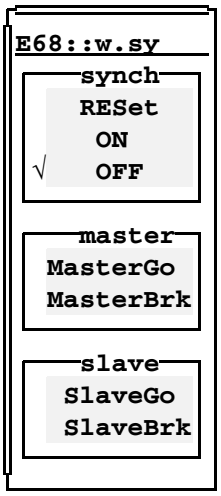
Examples

```
; force refresh with 256 cycles

ref.a  ud:0x0--0x03ff           ; address range 0 to 03ffh
ref.c  long                     ; long word
ref.i  0x4                      ; increment 4
ref.sb                          ; switch on refresh operation
```

Function

If simultaneously more than one emulator has to be started or halted, synchronization will be necessary. For this, one emulator is the master, and at least one other emulator has to be defined as the slave. The synchronization of **Go** and **Break** can be done separately. The master emulator, starting all other emulators, can be stopped by one of these via the **MasterBreak** function.

The SYNCH function synchronizes the emulation function only, analyzer operations remain unaffected.



Setup

SYnch.state	Display state
SYnch.RESet	Initialize system
SYnch.ON	Master-Slave synchronisation ON
SYnch.OFF	Master-Slave synchronisation OFF
SYnch.MasterGo	Define system as master for start
SYnch.MasterBrk	Define system as master for break
SYnch.SlaveGo	Define system as slave for start
SYnch.SlaveBrk	Define system as slave for break

Examples

```
sy.sg on                          ; Activate slave mode
sy.sb on
sy.mg off                         ; Deactivate master mode
sy.mb off
sy.on                             ; Switch-on synchronization
```