LAUTERBACH
DEVELOPMENT TOOLS

Application Note for the LOGGER Trace

Release 09.2023

MANUAL

Application Note for the LOGGER Trace

Application Note for the LOGGER Trace

Version 09-Oct-2023

History

22-Jan-21 New manual.

Introduction

LOGGER is a software trace method which requires a modification of the target application in order to write specific trace information to the a reserved buffer on the target memory using a trace format provided by LAUTERBACH. TRACE32 loads then the trace information from the target memory for display and processing.

The trace method LOGGER is mainly used when no hardware based trace is available.

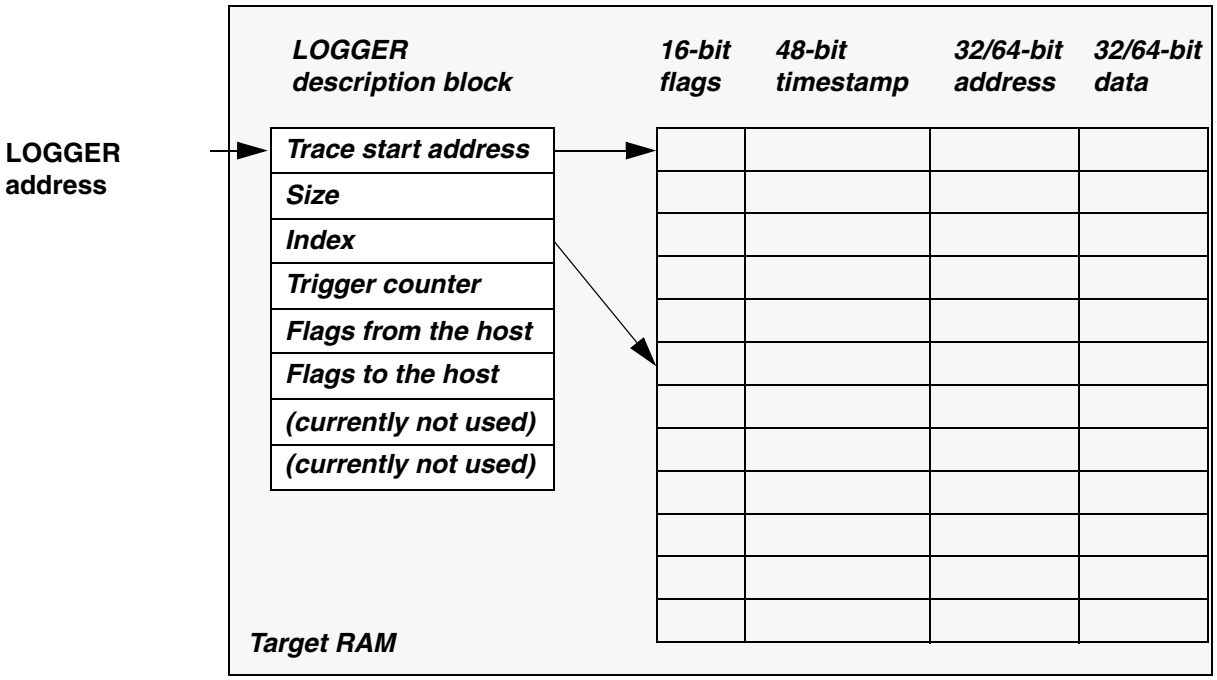
A description of the TRACE32 LOGGER commands can be found under **"LOGGER"** in General Commands Reference Guide L, page 44 (general_ref_l.pdf).

Related Tutorials

For video tutorials about the LOGGER, visit:
support.lauterbach.com/kb/articles/trace32-logger-trace

The LOGGER Trace Format

The LOGGER trace buffer includes a description block followed by the trace data.



LOGGER Description Block

The LOGGER description block has the following format:

Trace data start address (32- or 64-bit)	Start address of the trace data in the target RAM. This is the start of the LOGGER buffer plus the size of the logger header. The command **LOGGER.Mode 64Bit** controls the size of the start address: OFF (default): 32-bit ON : 64-bit
Size (32-bit)	Number of trace records (trace packets). The format of a trace record is described below.
Index (32-bit)	Index of the next record that should be written by the trace.
Trigger counter (32-bit)	Index of the trace record when the trigger was generated.

Flags from the host (32-bit)	Bit 0: Arm (high active) Bit 8: FIFO mode (low active), Stack mode (high active)
Flags to the host (32-bit)	Bit 0: Overrun (high active) Bit 8: Trigger (high active) Bit 9: Break (high active)

LOGGER Trace Records

The software trace can work in 2 operation modes:

- **Address/data trace**

  Address and data information is sampled.

- **Flow trace**

  A flow trace is available on architectures which provide a 'branch trace' capability like all PowerPC families. For a flow trace all changes in the program flow are sampled. The TRACE32 software reconstructs and displays the complete program flow out of this information. This mode is not documented in this manual. Refer to **"MPC5xx/8xx Debugger and Trace"** (debugger_ppc.pdf) for more information.

Address and Data Trace

Software Trace Record Description	
Flags (16-bit)	**Bits 12 .. 15:** trace packet type: • Fetch (0x1) • Data Read (0x2) • Data Write (0x3) **Bits 8 .. 11:** data width (1, 2, 4 or 8 bytes) **Bits 0 .. 7**: core number for SMP trace.
Timestamp (48-bit)	Timestamp from a timer, counter etc. from the target
Address (32- or 64-bit)	Fetch or data address. This field is 32- or 64-bit depending on **LOGGER.Mode 64Bit**.
Data (32- or 64-bit)	Data value. This field is 32- or 64-bit depending on **LOGGER.Mode 64Bit**.

Software Trace Record Description FlowTrace (e.g. PowerPC, SH4)	
Flags (16-bit)	0xF00x: Flow trace record
Timestamp (48-bit)	Timestamp from a timer, counter etc. from the target
Address (32-bit)	Address 1
Address (32-bit)	Address 2

LOGGER Target Application

C and C++ source and header files for using the TRACE32 LOGGER are available in TRACE32 system directory under `~~/demo/etc/logger`:

- **logger.h / logger.hpp**

  This header file contains the necessary type and macro definitions as well as function prototypes.

- **logger.c / logger.cpp**

  Contains the LOGGER functions.

Whenever a part of the application uses the LOGGER, the header file "`logger.h`"/"`logger.hpp`" must be included.

If the LOGGER is used in 64-bit mode, the macro `LOGGER_64BIT` has to be defined.

If the LOGGER is used in SMP mode, the macro `LOGGER_SMP` has to be defined.

LOGGER Functions

T32_LoggerInit

Prototype:

```
void T32_LoggerInit ();
void T32_LoggerC::T32_LoggerInit();
```

Initializes the LOGGER internal data structures and calls **T32_TimerInit()**. This routine must be called before using any other LOGGER related routines.

When using the C++ LOGGER files, this function is called in the constructor of the `T32_LoggerC` class.

The **LOGGER.Init** command has to be executed after calling this function in order to read the LOGGER buffer size.

T32_TimerInit

Prototype:

```
void T32_TimerInit ();
void T32_LoggerC::T32_TimerInit();
```

Initializes the architecture specific timer. This routine is architecture specific and must be implemented by the user.

T32_TimerGet

Prototype:

```
unsigned long long T32_TimerGet ();
unsigned long long T32_LoggerC::T32_TimerGet()
```

Returns current timestamp of architecture specific timer (48-bit width). This routine is architecture specific and must be implemented by the user.

Example for PowerPC using the TBL register:

```
unsigned long long T32_TimerGet ()
{
     unsigned long tb;
     asm volatile ("mftb %0": "=r" (tb));
     return tb;
}
```

T32_LoggerData

Prototype:

Single core mode (`LOGGER_SMP` undefined):

```
void T32_LoggerData (int cycletype, void* address, data_t data);
```

SMP mode (`LOGGER_SMP` defined):

```
void T32_LoggerData (int cycletype, void* address, data_t data,int core);
```

C++:

```
void T32_LoggerC::T32_LoggerData(int cycletype, void* address,
                                                 data_t data, int core=0)
```

Adds a new event to the LOGGER.

Parameters:

```
cycletype               ; Type of the event e.g. T32_FETCH or
                        ; (T32_DATA_READ|T32_LONG).

address                 ; Address of the event. This is an instruction
                        ; address in case of T32_FETCH and T32_EXECUTE,
                        ; otherwise a data address.

data                    ; Data related to the event.

core                    ; Core number for SMP systems.
```

Examples (single core):

```
// add a write cycle of the 32-bit variable mcount to the LOGGER trace
T32_LoggerData (T32_DATA_WRITE|T32_LONG, &mcount, mcount);

// add a read cycle of the 8-bit variable vchar to the LOGGER trace
T32_LoggerData (T32_DATA_WRITE|T32_BYTE, &vchar, vchar);

// add a fetch cycle of the function func2 to the LOGGER trace
T32_LoggerData (T32_FETCH, func2, 0 /* unused */);
```

Examples (SMP):

```
// add a write cycle of the 16-bit variable vshort to the LOGGER trace
// for core 1
T32_LoggerData (T32_DATA_WRITE|T32_WORD, &vshort, vshort, 1);

// add a fetch cycle of the function func1 to the LOGGER trace for
// core 0
T32_LoggerData (T32_FETCH, func1, 0 /* unused */, 0);
```

T32_LoggerDataFast

Prototype:

Single core mode (`LOGGER_SMP` undefined):

```
void T32_LoggerDataFast (int cycletype, void* address, data_t data);
```

SMP mode (`LOGGER_SMP` defined):

```
void T32_LoggerDataFast (int cycletype, void* address, data_t data,
                                                              int core);
```

C++:

```
void T32_LoggerC::T32_LoggerDataFast(int cycletype, void* address,
                                                  data_t data, int core=0);
```

Adds a new event to the LOGGER. In comparison with **T32_LoggerData()**, this function

- does not write a timestamp
- does not check if the LOGGER is armed (**LOGGER.Arm**)
- does not support **Stack** mode (**LOGGER.Mode Stack**)

Refer to **T32_LoggerData()** for a description of the parameters.

T32_LoggerTrigger

Prototype:

```
void T32_LoggerTrigger();
void T32_LoggerC::T32_LoggerTrigger()
```

Generates a LOGGER trigger.

Data Cycles

Used together with `T32_DATA_READ` and `T32_DATA_WRITE`.

T32_BYTE	8-bit access
T32_WORD	16-bit access
T32_LONG	32-bit access
T32_QUAD	64-bit access

Cycle Types

T32_FETCH	Adds a trace record for a program fetch cycle.
T32_EXECUTE	Adds a trace record for a program execute cycle, data holds number of executed bytes.
T32_DATA_READ	Adds a trace record with a read transaction (load).
T32_DATA_WRITE	Adds a trace record with a write transaction (store).

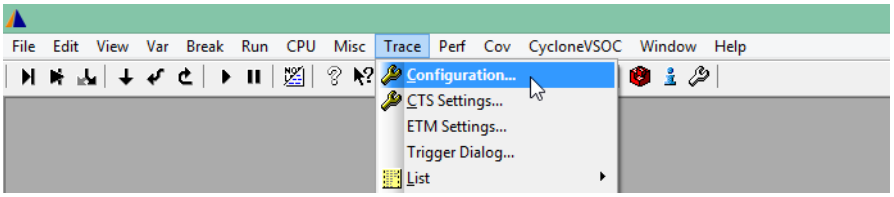
LOGGER Size

T32_LOGGER_SIZE	Size of the LOGGER ring buffer, must be a power of 2.

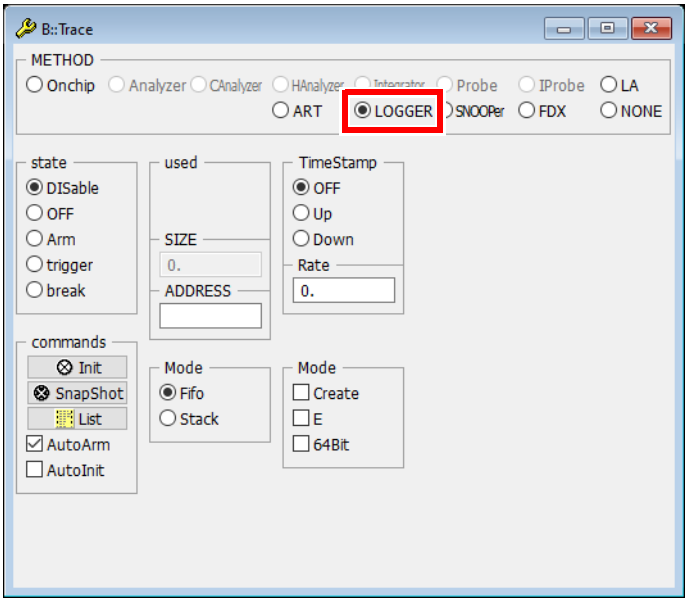
NOTE:	In this chapter, we assume that the target application has already been instrumented to include the LOGGER functionality.

The LOGGER trace is part of the TRACE32 trace framework. To configure the LOGGER trace:

1. On the TRACE32 main menu bar, choose **Trace** menu > **Configuration**:



2. Under **METHOD**, click the radio option **LOGGER**.



Or execute the following commands on the TRACE32 command line:

```
Trace.state
Trace.METHOD LOGGER
```

Alternatively, execute the **LOGGER.state** command:

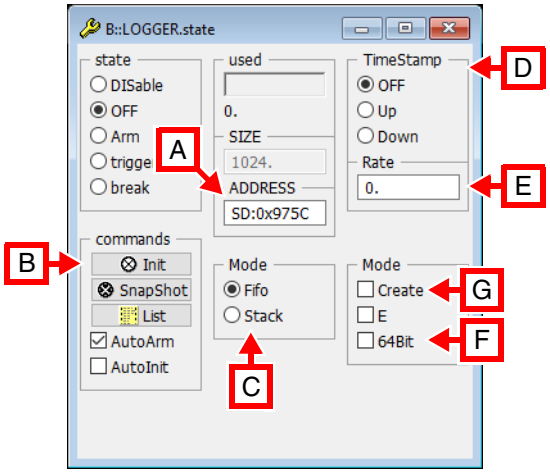
```
LOGGER.state
```

All commands relative to the LOGGER trace can be executed using the **Trace** command group (e.g. **Trace.List**) after selecting the LOGGER method in the **Trace.state** window, or using the LOGGER command group (e.g. **LOGGER.List**). The second form is especially useful if the LOGGER trace should be used together with a different trace method. In this application note, the **LOGGER** command group will be used.

The following steps are needed to configure the LOGGER trace:

1. Reset the LOGGER trace to its default settings using the command **LOGGER.RESet**.

2. Define the address of the LOGGER trace control block in memory using the **ADDRESS** field **[A]** of the **LOGGER.state** window or using the command **LOGGER.ADDRESS**




```
LOGGER.ADDRESS T32_LoggerStruct
```

3. Select the recording mode **[C]**. In **Fifo** mode, if the LOGGER trace buffer is full, new trace records will overwrite older records. Therefore the LOGGER trace memory always contains the last cycles before stopping the trace. In **Stack** mode however, if the LOGGER trace buffer is full the recording will be stopped so that the trace buffer always contains the first records.

   The recording mode can also be set using the commands **LOGGER.Mode Fifo** or **LOGGER.Mode Stack**. The LOGGER trace recording mode is set per default to **Fifo**.

4. Initialize the LOGGER by pressing the **Init** button **[B]** or using the **LOGGER.Init** command. The function **T32_LoggerInit()** should have been already executed before initializing the LOGGER, for example:

```
Go sYmbol.EXIT(T32_LoggerInit)
LOGGER.Init
```

   After the initialization, the **SIZE** field contains the size of the LOGGER trace.

Please note that it is also possible to use the debugger in order to initialize the LOGGER control block instead of calling the **T32_LoggerInit()** function. This can be enabled by setting the LOGGER mode **Create [G]** or executing the command **LOGGER.Mode Create ON**. It will be then possible to specify the LOGGER size in TRACE32. You should however make sure in this case that the selected size is reserved by the target application for the LOGGER buffer.

Example:

```
LOGGER.Mode Create ON
LOGGER.SIZE 1024
LOGGER.Init
```

5. Configure the timestamp usage of the LOGGER trace **[D]**. Per default, timestamps are disabled (**OFF**). This setting should be used if the LOGGER target code does not generate timestamp information in the LOGGER trace records. Otherwise, **Up** should be selected if generated timestamps are counting upwards and **Down** if generated timestamps are counting downwards.

   If timestamps are used, their frequency (in ticks per second) has additionally to be specified using the **Rate** field **[E]** or the **LOGGER.TimeStamp.Rate** command.

6. If the LOGGER operates in 64-bit mode, the **64Bit** check box **[F]** has to be selected. Alternatively, use the command **LOGGER.Mode 64Bit**.

The settings done in the **LOGGER.state** window can be saved in the format of a PRACTICE script to an external file using the **STOre** command or to the clipboard using the **ClipSTOre** command.

STOre *<file>* **LOGGER**	Create a PRACTICE script to restore the LOGGER trace settings
ClipSTOre LOGGER	Provide the commands to restore the LOGGER trace settings in the cliptext

Per default, the LOGGER automatically starts recording when the program execution is started and stops recording when it is stopped. This behavior can be controlled using the command **LOGGER.AutoArm** or the **AutoArm** check box from the **LOGGER.state** window. The trace recording can also be controlled manually using the commands **LOGGER.Arm** and **LOGGER.OFF**, or the **Arm** and **OFF** radio buttons.

Display of LOGGER Trace Contents

The LOGGER trace contents can only be displayed after the recording has been stopped (state **OFF** or **break**). A display of the trace contents while recording is not possible. Moreover, in order to read the trace data, the debugger needs to access the memory. This means that either the program execution has to be stopped or, if supported by the target processor, memory access on run-time needs to me enabled. Please refer to the description of the command **SYStem.MemAccess** in your **Processor Architecture Manual** for more information. The LOGGER dual port mode (**LOGGER.Mode E ON**) needs to be enabled if the LOGGER should access the memory on run-time.

Example 1: display the LOGGER result after stopping the program execution

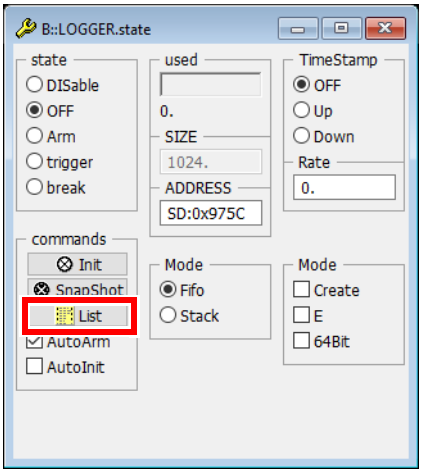
```
LOGGER.AutoArm ON
; stop the program execution, trace recording will also be stopped:
Break
; display the trace recording
LOGGER.List
```

Example 2: display the LOGGER result on run-time without stopping the target processor:

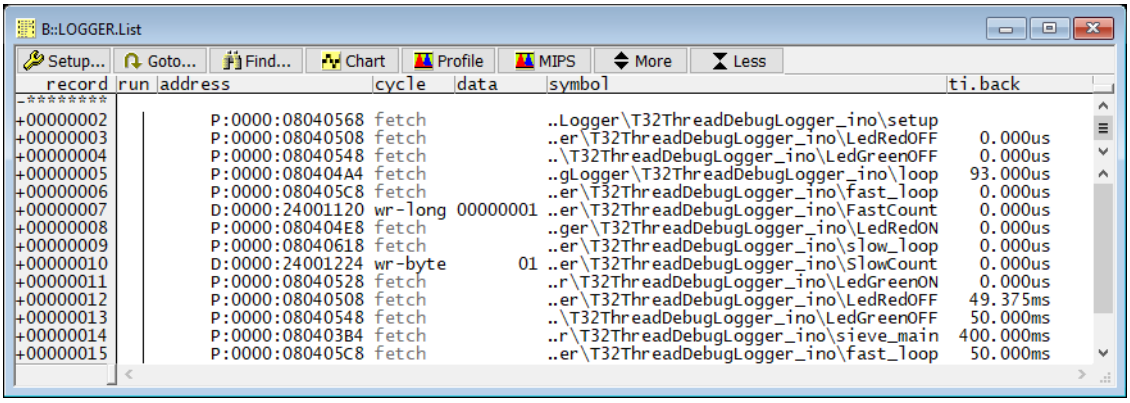
```
; enable memory access on run-time (if supported by the target processor)
; the command may differ depending on the target architecture
SYStem.MemAccess Enable

; enable LOGGER dual port mode
LOGGER.Mode E ON

; stop the trace recording
LOGGER.OFF

; display the trace recording
LOGGER.List
```

List of Recorded Samples

The LOGGER trace contents can be displayed using the **List** button from the **LOGGER.state** window or using the command **LOGGER.List**.




```
LOGGER.List
```



The **LOGGER.List** window displays per default for each recorded trace packet the following information:

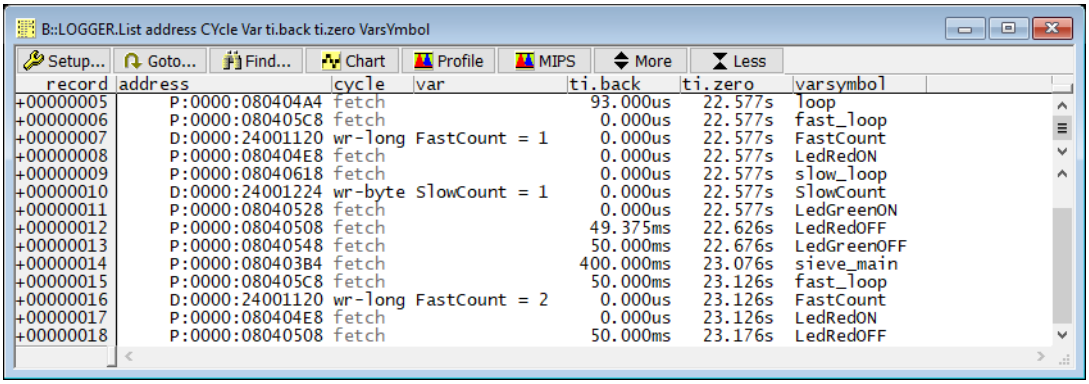
run	displays the core number for SMP systems if the LOGGER is used in SMP mode. This column is empty otherwise.
address	instruction address for fetch cycles or data load/store address for data cycles.
cycle	cycle type: • **fetch**: program fetch cycle • **execute**: program execute cycle • **wr-*<width>***: read transaction. *<width>*: byte, word, long or quad. • **rd-*<width>***: read transaction. *<width>*: byte, word, long or quad.

symbol	symbolic information.
ti.back	time relative to previous record. This column is empty if timestamps are not enabled.

The different columns in the window can be rearranged by changing the order of the **LOGGER.List** parameters. Moreover, other columns can be added to the window. You can use for example the keyword **Var** to display the recorded variable in its HLL representation or **TIme.Zero** to display the time relative to the start of the recording. Please refer to the documentation of the **LOGGER.List** command for a complete list of the different possible parameters.

Example:

```
LOGGER.List address CYcle Var ti.back ti.zero VarSymbol
```



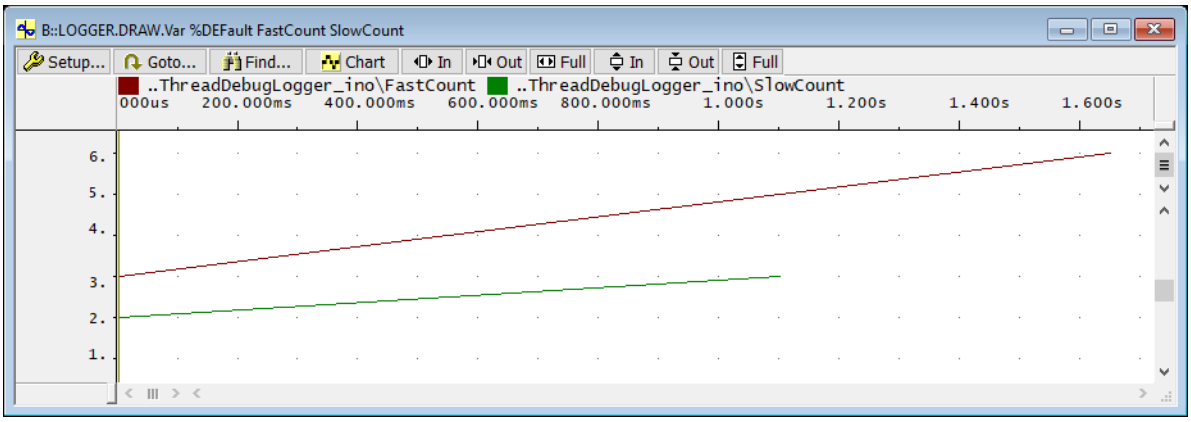
record	address	cycle	var	ti.back	ti.zero	varsymbol
+00000005	P:0000:080404A4	fetch		93.000us	22.577s	loop
+00000006	P:0000:080405C8	fetch		0.000us	22.577s	fast_loop
+00000007	D:0000:24001120	wr-long	FastCount = 1	0.000us	22.577s	FastCount
+00000008	P:0000:080404E8	fetch		0.000us	22.577s	LedRedON
+00000009	P:0000:08040618	fetch		0.000us	22.577s	slow_loop
+00000010	D:0000:24001224	wr-byte	SlowCount = 1	0.000us	22.577s	SlowCount
+00000011	P:0000:08040528	fetch		0.000us	22.577s	LedGreenON
+00000012	P:0000:08040508	fetch		49.375ms	22.626s	LedRedOFF
+00000013	P:0000:08040548	fetch		50.000ms	22.676s	LedGreenOFF
+00000014	P:0000:080403B4	fetch		400.000ms	23.076s	sieve_main
+00000015	P:0000:080405C8	fetch		50.000ms	23.126s	fast_loop
+00000016	D:0000:24001120	wr-long	FastCount = 2	0.000us	23.126s	FastCount
+00000017	P:0000:080404E8	fetch		0.000us	23.126s	LedRedON
+00000018	P:0000:08040508	fetch		50.000ms	23.176s	LedRedOFF

Graphical Display of LOGGER Trace Results

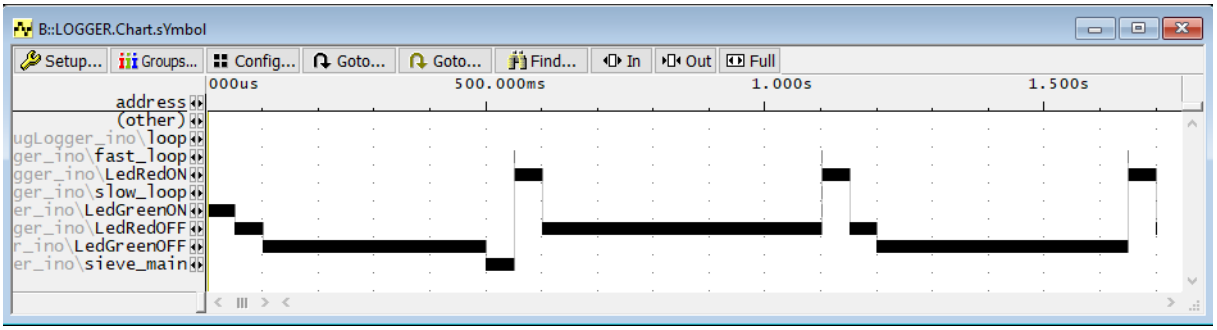
You can use the command **LOGGER.DRAW** to display the recorded data values graphically. Please refer to the documentation of the **<trace>.DRAW** command group for more information.

Example:

```
LOGGER.DRAW.Var %DEFault FastCount SlowCount
```

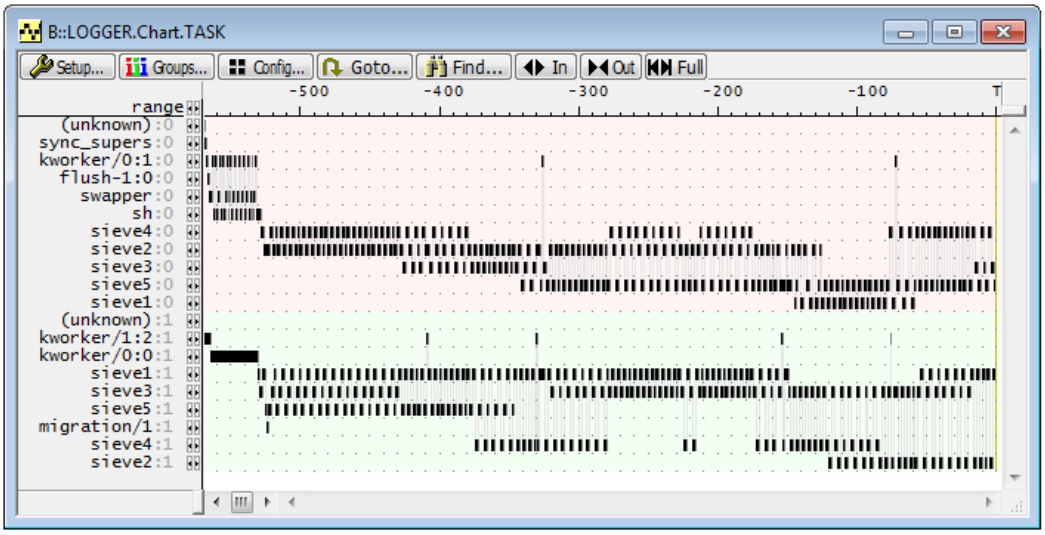


If fetch accesses are recorded for multiple function, then you can display the function activity chart using the command **LOGGER.Chart.sYmbol**.



Using the LOGGER for Task Switch Trace

If the used target processor does not provide an on-chip trace support for recording task switches, the operating system can be patched with the LOGGER functions in order to write the task switch information to a reserved buffer in the memory. The task switch information can be then displayed with the command **LOGGER.Chart.TASK**



An example for the Linux kernel based on a kernel module is described under **"Using the LOGGER for Task Switch Trace"** in Training Linux Debugging, page 51 (training_rtos_linux.pdf).

LOGGER Trace Trigger

The TRACE32 LOGGER offers a basic triggering functionality. The target application can trigger the LOGGER to stop recording on a specific event. The trigger can be generated by calling the function **T32_LoggerTrigger()**.

In order to react on the trigger, the debugger needs to access the LOGGER control data on run-time. The LOGGER trigger can thus only work if memory access on run-time is possible and enabled (**SYStem.MemAccess**). Moreover, the LOGGER dual port access needs to be enabled using the command **LOGGER.Mode E ON**.

Depending on the speed of the memory access on run-time, there could be a delay before the debugger reacts on the trigger.

Example: generate a trigger in order to stop the trace recording when the variable `mcount` gets the value 5

```
...
T32_LoggerData(T32_DATA_WRITE | T32_LONG, &mcount, mcount, 0);
mcount++;
if (mcount == 5) {
  T32_LoggerTrigger();
  T32_LoggerStruct.oflags |= T32LOGGERDATA_OFLAG_BREAK;
}
...
```

You can see in the following screen shot that the trigger was generated after 5 records (`T00000000`) but the debugger reacted after 12 additional records.