


# API for Remote Control and JTAG Access

---

[TRACE32 Online Help](#)

[TRACE32 Directory](#)

[TRACE32 Index](#)

<b>Misc</b> .....	
<b>API for Remote Control and JTAG Access</b> .....	<b>1</b>
<b>Basic Concepts</b> .....	<b>3</b>
Release Information	3
Introduction	4
Interfaces	5
Operating of the API Requests	6
<b>Building an Application with API</b> .....	<b>7</b>
API Files	7
Connecting API and Application	7
<b>Communication Setup</b> .....	<b>8</b>
Preparing TRACE32 Software	8
Configuring the API	8
<b>API Functions</b> .....	<b>9</b>
Generic API Functions	9
T32_Config	Configure Driver 9
T32_Init	Initialize driver and connect 10
T32_Exit	Close connection 10
T32_Attach	Attach TRACE32 device 11
T32_Nop	Send Empty Message 12
T32_Ping	Send Ping Message 12
T32_Cmd	Execute PRACTICE Command 13
T32_CmdWin	Execute PRACTICE Command 13
T32_Stop	Stop PRACTICE program 14
T32_EvalGet	Get Evaluation Result 15
T32_GetMessage	Get Message Line Contents 16
T32_GetPracticeState	Check if a PRACTICE script is running 17
Functions for using the API with Multiple Debuggers	18
T32_GetChannelSize	Get size of channel structure 18
T32_GetChannelDefaults	Get default channel parameters 19
T32_SetChannel	Set active channel 20
ICD/ICE API Functions	21
T32_GetState	Get State of ICE/ICD 21
T32_GetCpuInfo	Get Information about used CPU 22

T32_GetRam	Get Memory Mapping	23
T32_ResetCPU	Prepare for Emulation	24
T32_ReadMemory	Read Target Memory	25
T32_WriteMemory	Write to Target Memory	27
T32_WriteMemoryPipe	Write to Target Memory pipelined	28
T32_ReadRegister	Read CPU Registers	29
T32_WriteRegister	Write CPU Registers	30
T32_ReadPP	Read Program Pointer	31
T32_ReadBreakpoint	Read Breakpoints	31
T32_WriteBreakpoint	Write Breakpoints	33
T32_Step	Single Step	34
T32_StepMode	Single Step with Mode Control	35
T32_Go	Start Realtime	36
T32_Break	Stop Realtime	36
T32_GetTriggerMessage	Get Trigger Message Contents	37
T32_GetSymbol	Get Symbol Information	38
T32_GetSource	Get Source Filename and Line	40
T32_GetSelectedSource	Get Source Filename and Line of Selection	41
T32_AnaStatusGet	Get State of State Analyzer	42
T32_AnaRecordGet	Get One Record of State Analyzer	43
T32_GetTraceState	Get State of Trace	46
T32_ReadTrace	Get One Record of Trace	47
T32_GetSocketHandle	Get the handle of the TRACE32 socket	51
T32_NotifyStateEnable	Register a function to be called at special event	52
T32_CheckStateNotify	Check message to receive for state notify	53
ICD TAP Access API Functions		55
T32_TAPAccessSetInfo	Configure JTAG Interface	56
T32_TAPAccessShiftIR	Shift Data to/from Instruction Register	58
T32_TAPAccessShiftDR	Shift Data to/from Data Register	59
T32_TAPAccessDirect	Direct JTAG Port Access	60
T32_TAPAccessShiftRaw	RAW JTAG Shifts	63
T32_TAPAccessAlloc	Retrieve a Handle for Bundled Access Mode	66
T32_TAPAccessFree	Release Handle for Bundled Access Mode	67
T32_TAPAccessExecute	Execute a Bundled TAP Access	67
T32_TAPAccessRelease	Unlock Debugger	68
<b>Version Control</b> .....		<b>69</b>

## Basic Concepts

---

## Release Information

---

Release 4.0, shipped from 01-September-2004 on, includes the ability to connect to several debuggers at once (multi-core debugging). It is backward compatible to release 3.

Release 3.0, shipped from 01-April-1998 on, is a compatible extended version. This document has changed.

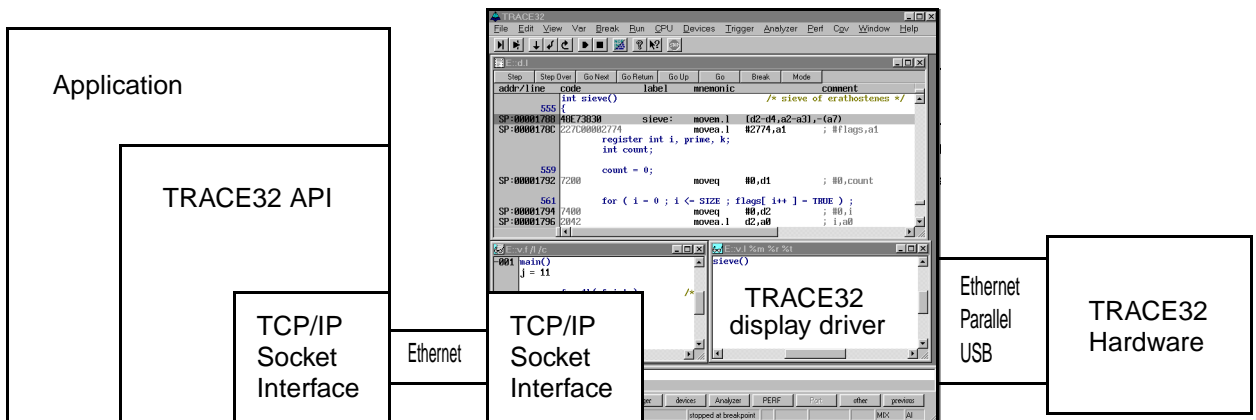
Release 2.0, shipped from 28-Oct.-1996 on, is incompatible to previous versions, regarding the socket communication. You need the 2.0 versions of hlinknet.c and hremote.c. Recompile your software with the new files.

# Introduction

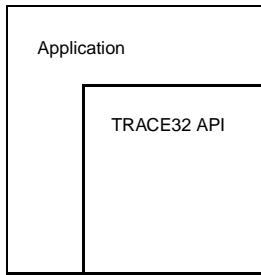
The TRACE32 Software contains an interface for external control of the TRACE32. The TRACE32 Application Programming Interface (further referred to as API) gives external applications the possibility to control the debugger and the program run by the debugger.

The API is built as a C library with a C function interface to the controlling application. The API communicates with the TRACE32 application (not with the TRACE32 itself!) using a socket interface. The command chain using TRACE32 API then looks like that:

Application ---> TRACE32 API ---> TRACE32 application --> TRACE32  
(C Functions) (sockets) (HW interface)

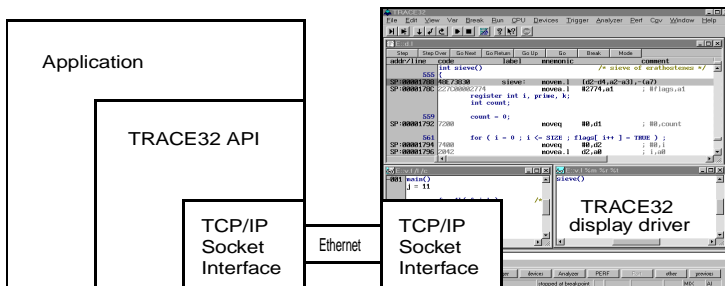


## Application --> TRACE32 API



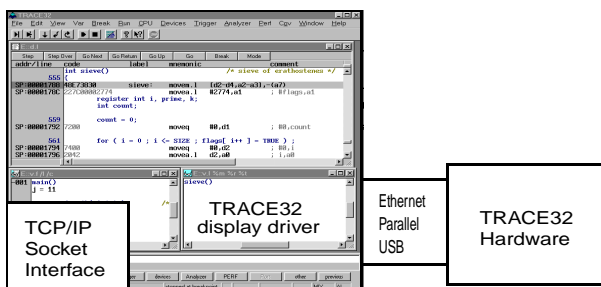
The application uses the API as ordinary C functions. The API is linked to the application at the usual linking stage.

## TRACE32 API --> TRACE32 display driver



The communication to the TRACE32 software is implemented as a socket interface. This means, that the controlling application (including API) and the debugger software can reside on two different hosts, using network connections for communication. But be aware that this connection is not fault tolerant, because no network error detection is implemented in the API. It is recommended, that both parties run on the same host.

## TRACE32 display driver --> TRACE32



The debugger software just routes the API requests to the TRACE32 hardware. This interface is the one, you choosed for your debugger. E.g. it could be Ethernet, parallel or USB.

The answers for the request go exactly the opposite way, returning information to the application in passed buffers.

## Operating of the API Requests

---

The API requests are executed just in parallel with normal TRACE32 operation. You can use both, the TRACE32 user interface and the API simultaneously, although it is not recommended. The application will not be informed about changes that are done via the user interface. Also, unpredictable errors may occur, if e.g. an API request and a running PRACTICE file interfere.

## API Files

---

The API consists of two C source files and one C header file:

- **hlinknet.c**  
This file contains and handles the socket interface to the TRACE32 debugger software.
- **hremote.c**  
All API functions are coded in this source file
- **t32.h**  
This header file consists of some definitions and all function prototypes needed.

## Connecting API and Application

---

Whenever a part of the application uses the API, the header file "t32.h" must be included. The corresponding C source file must contain the line

```
#include "t32.h"
```

quite at the beginning of the source.

When compiling and linking the application, the API files must be handled as normal source components of the application. Compilation could look like this:

```
cc -c hlinknet.c  
cc -c hremote.c  
cc -c applic.c
```

assuming, that the application is coded in a file called "applic.c" and your C compiler is called "cc". The linker run is then invoked with:

```
cc -o applic hlinknet.o hremote.o applic.o
```

assuming the linker name "cc" and the object extension "o".

## Preparing TRACE32 Software

---

The TRACE32 Software has to be configured for use with a remote control, such as the API. To allow and configure remote control, add the following lines between two empty lines to the file "config.t32". If you are using MS-Windows and T32Start application to start TRAC32-SW you need to open the configuration at "advanced settings" where you can select "Use Port: yes" in the "API Port" folder. The automatically created config file (e.g. C:\temp\userT32\_1000123.t32) will have the necessary lines automatically.

```
RCL=NETASSIST
PACKLEN=1024
PORT=20000
```

PACKLEN specifies the maximum package length in bytes for the socket communication. It must not be bigger than 1024 and must fit to the value defined by T32\_Config().

The port number specifies the UDP port which is used to communicate with the API. The default is 20000. If this port is already in use, try one higher than 20000.

See also the TRACE32 online help ("help.ap RCL").

## Configuring the API

---

The API must be configured with the functions T32\_Config(), T32\_Init() and T32\_Attach(). T32\_Config() takes two string arguments, usually the node name and the port number. The function T32\_Init() then does a setup of the communication channel. T32\_Attach() attaches to the actual instrument. The T32\_Exit() function closes the connection and should always be called before terminating the application.

See chapter "Generic API functions" for a detailed description of these functions.



## Generic API Functions

---

### T32\_Config

### Configure Driver

---

#### Prototype:

```
int T32_Config ( char * string1, char * string2 );
```

#### Parameters:

```
string1, string2      ; commands for ethernet interface
```

#### Returns:

0 for ok, otherwise Error value

The two strings are concatenated and the resulting command is sent to the communication driver of the API. On UNIX/VMS systems this driver is the standard Ethernet interface driver of TRACE32. All commands described for this interface can be used here. Usually three commands will be used:

```
NODE=localhost  
PACKLEN=1024  
PORT=20000
```

`NODE` defines, on which host the TRACE32 display driver runs - normally local host.

`PACKLEN` specifies the maximum data package length and must not be bigger than 1024 and must fit to the value defined in the "config.t32" file (see chapter 3.1).

The `PORT` command defines the UDP port to use. If omitted, it defaults to 20000. Be sure that these settings fit to the RCL settings in the "config.t32" file.

#### Example:

```
error = T32_Config ( "NODE=", "myhost" );  
error = T32_Config ( "PACKLEN=", "1024" );  
error = T32_Config ( "PORT=", "20010" );
```

**Prototype:**

```
int T32_Init ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

This function initializes the driver and establishes the connection to the TRACE32 display driver. If zero is returned, the connection was set up successfully.

It is recommended to call `T32_Attach()` immediately after `T32_Init()` to have the full set of API functions available.

**Example:**

```
error = T32_Init ();
```

**Prototype:**

```
int T32_Exit ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

This function ends the connection to the TRACE32 display driver. This command should always be called before ending the application.

**Example:**

```
error = T32_Exit ();
```

**Prototype:**

```
int T32_Attach ( int dev );
```

**Parameters:**

dev	Device specifier
-----	------------------

**Returns:**

0 for ok, otherwise Error value

This command attaches the control to the specified TRACE32 device. It is recommended to attach to T32\_DEV\_ICE immediately after T32\_Init(), to have access to all API functions.

T32_DEV_OS	Basic operating system of the TRACE32 ("::"), disables all device specific commands (default)
T32_DEV_ICE	Debugger ("E::" or "B::"), including Basic OS commands
T32_DEV_ICD	same as T32_DEV_ICE

**Example:**

```
error = T32_Attach ( T32_DEV_ICE );
```

**Prototype:**

```
int T32_Nop ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

Send an empty message to the TRACE32 display driver and wait for it's answer.

**Example:**

```
error = T32_Nop ();
```

**Prototype:**

```
int T32_Ping ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

Sends a "ping" message to the TRACE32.

**Example:**

```
error = T32_Ping ();
```

**Prototype:**

```
int T32_Cmd ( char * command );
```

**Parameters:**

```
command          ; PRACTICE command to execute
```

**Returns:**

0 for ok, otherwise Error value

With this function a PRACTICE command is passed to TRACE32 for execution. Any valid PRACTICE command is allowed, including the start of a .cmm script via the "DO" command.

Currently the error values only indicate if there was a communication problem between debugger and API. Errors caused by executing the command are not reported (will be changed). For retrieving error information use the call `T32_GetMessage()` and consider the message type.

**Example:**

```
error = T32_Cmd ( "Data.Set %Long 12200 033FFC00" );
```

**Prototype:**

```
int T32_CmdWin ( dword, char * command );
```

Executing a PRACTICE command with given windows handler.

**Prototype:**

```
int T32_Stop ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

If a PRACTICE script is currently running, it is stopped. If an application is running in the ICE, it will not be affected by this command. For stopping the target program use `T32_Break()`.

**Example:**

```
error = T32_Stop ();
```

**Prototype:**

```
int T32_EvalGet ( unsigned long * peval );
```

**Parameters:**

```
peval           ; pointer to variable to catch the evaluation result
```

**Returns:**

0 for ok, otherwise Error value

Some of the PRACTICE commands and other functions set a global variable to store return values, evaluation results or error conditions. This value is always specific to the command used. The function T32\_EvalGet reads this value.

**Example:**

```
unsigned long result;  
error = T32_EvalGet ( &result );  
printf ("Result of last PRACTICE command: %d\n", result);
```

**NOTE:** Although this function belongs to the generic section, it is only available with the ICE device (See T32\_Attach).

**Prototype:**

```
int T32_GetMessage ( char message[128], word * ptype );
```

**Parameters:**

```
message          ; output parameter, set by API
ptype           ; output parameter, set by API
```

**Returns:**

0 for OK, otherwise Error value

Most PRACTICE commands write messages to the message line of TRACE32. This function reads the contents of the message line and the type of the message.

"message" must be a user allocated character array of at least 128 elements.

The message types are currently defined as following and can be combined:

Type	Meaning
1	General Information
2	Error
8	Status Information
16	Error Information
32	Temporary Display
64	Temporary Information

**Example:**

```
char message[128];
word mode;

error = T32_Cmd ("print"); /* delete previous outputs */
error = T32_Cmd ("print clock.date()");
error = T32_GetMessage (message, &mode);
printf ("Message: %s\nMode: %d\n", message, mode);
```



**Prototype:**

```
int T32_GetPracticeState( int* pstate );
```

**Parameters:**

```
pstate           ; output parameter, set by API  
                  ; 0 ... not running  
                  ; 1 ... running
```

**Return:**

0 for OK, otherwise Error value

Returns the run-state of PRACTICE. Use this command to poll for the end of a PRACTICE started via T32\_Cmd().

# Functions for using the API with Multiple Debuggers

---

A single API instance can be used with several TRACE32 debuggers (e.g. for Multi-Core debugging) by creating a communication channel to each of the debuggers. Instead of passing the channel as parameter to API calls, the whole API is switched to a specific channel via `T32_SetChannel()`.

A channel is created by allocating the required amount of memory (`T32_GetChannelSize()`), initializing this memory by `T32_GetChannelDefaults()`, activating it via `T32_SetChannel()` and then using `T32_Config()`, `T32_Init()` and `T32_Exit()` as would be done on the default channel.

Note that despite the channel concept, each debugger must be assigned a unique PORT address in its configuration file `config.t32` file.

## T32\_GetChannelSize

## Get size of channel structure

---

### Prototype:

```
int T32_GetChannelSize ( void );
```

### Parameters:

none

### Returns:

size\_of channel structure

Only necessary for multi-channel usage.

This function returns the size of a channel structure. Allocate memory with this size to be used for the channel switching.

### Example (see full example at `T32_SetChannel()`):

```
void* channel = malloc (T32_GetChannelSize());
```

**Prototype:**

```
void T32_GetChannelDefaults (void* channel);
```

**Parameters:**

pointer to channel receiving the defaults

**Returns:**

none

Only necessary for multi-channel usage.

This function fills the channel structure with default values. This is mandatory if using multiple channels.

**Example (see full example at T32\_SetChannel()):**

```
T32_GetChannelDefaults (channel);
```

**Prototype:**

```
void T32_SetChannel (void* channel);
```

**Parameters:**

pointer to activating channel

**Returns:**

none

Only necessary for multi-channel usage.

This function sets the active channel to be used for further T32\_\* calls.

**Example:**

```
void* channel_1 = malloc (T32_GetChannelSize());
void* channel_2 = malloc (T32_GetChannelSize());
T32_GetChannelDefaults (channel_1);
T32_GetChannelDefaults (channel_2);
T32_SetChannel (channel_1);
T32_Config ("PORT=", "20000");
T32_Init ();
T32_Attach (T32_DEV_ICE);
T32_SetChannel (channel_2);
T32_Config ("PORT=", "20002");
T32_Init ();
T32_Attach (T32_DEV_ICE);
...
```

This chapter describes all functions available with the ICE device of the TRACE32. See `T32_Attach()` for how to specify a device.

## T32\_GetState

## Get State of ICE/ICD

---

### Prototype:

```
int T32_GetState ( int * pstate );
```

### Parameters:

```
pstate           ; pointer to variable to catch the ICE state
```

### Returns:

0 for ok, otherwise Error value

Use this function to get the main state of the ICE. `*pstate` can have four different values:

0	General Information
1	System is halted, CPU makes no cycles (r.g. STOP instruction)
2	Emulation is stopped
3	Emulation is running

### Example:

```
int state;
error = T32_GetState ( &state );
printf ("System is ");
switch (state)
{
    case 0: printf ("down.\n");
    case 1: printf ("halted.\n");
    case 2: printf ("stopped.\n");
    case 3: printf ("running.\n");
}
```

**Prototype:**

```
int T32_GetCpuInfo ( char ** pstring, word * pfpu, word * pendian,
                    word * ptype );
```

**Parameters:**

<code>pstring</code>	; pointer to variable to catch a pointer to a string ; describing the cpu
<code>pfpu</code>	; pointer to variable to catch the fpu type
<code>pendian</code>	; pointer to variable to cache the byte order
<code>ptype</code>	; additional internal information

**Returns:**

0 for ok, otherwise Error value

This function gives information about the CPU type. `pstring` will contain an ASCII string with the CPU type and family. `pfpu` describes, whether a FPU is present or not. This is currently not used and always zero. `pendian` describes the byte order of the CPU: zero means big endian (12 34 becomes 1234), otherwise little endian (12 34 becomes 3412). `ptype` is for internal information and useless to the user.

**Example:**

```
char * cpustring = "";
unsigned short hasfpu, endian, tmp;
error = T32_GetCpuInfo ( &cpustring, &hasfpu, &endian, &tmp );
printf ("CPU is %s.\n", cpustring);
printf ("Endian type is %s.\n", endian?"little":"big");
```

**Prototype:**

```
int T32_GetRam ( dword * pstart, dword * pend, word * paccess );
```

**Parameters:**

```
pstart           ; pointer to variable with start address
pend             ; pointer to variable to catch the end address
paccess         ; pointer to variable with access type
```

**Returns:**

0 for ok, otherwise Error value

The memory mapping of the emulator can be get with this function. `pstart` specifies the first address to search for a memory block. A zero will force to search from beginning of the address space. After return, `pstart` contains the first address, at which the specified memory is mapped and `pend` contains the last address of the continuously mapped block To get all mapped blocks, call `T32_GetRam` repeatedly, until `paccess == 0`. `paccess` must contain the access mode. Currently there are two modes: 1 for Data RAM ("D:") and 2 for Program RAM ("P:"). If `paccess` contains zero after return, and no error occurred, then no (more) mapped memory was found. Otherwise `paccess` is not equal to zero (but changed!).

**Example:**

```
unsigned long start, end;
unsigned short access;
start = 0;                               /* search for first mamory block */
access = 1;                               /* search for Data RAM Block    */
error = T32_GetRam ( &start, &end, &access );
if (!access) printf ("No Dataram found.\n");
else printf ("Dataram found from %x to %x.\n", start, end);
```

**Prototype:**

```
int T32_ResetCPU ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

Prepares the ICE for emulation. This is done by executing the PRACTICE commands SYStem.UP and Register.RESet. This function can also be used to get control after the target software has crashed.

**Example:**

```
error = T32_ResetCPU ( );
```



**Prototype:**

```
int T32_ReadMemory (dword address, int access, byte * buffer, int size);
```

**Parameters:**

```
address          ; target memory address to start read
access           ; memory access specifier
buffer           ; output
size             ; number of bytes to read
```

**Returns:**

0 for ok, otherwise Error value

Reads data from target memory. The size of the data block is not limited.

The access parameter defines the memory access class and access method:

Bit 0...3                      encodes the memory class, values as defined below

Bit 6:                         Set for emulation memory access (E:, dual port access)

**Example:**

```
unsigned char buffer[16];
error = T32_ReadMemory ( 0x100, 0x40, buffer, 16 );     // ED: access
```

**Memory Classes:**

Generically used memory access class values (independent of CPU architecture):	
0	Data access, D:
1	Program access, P:
12	AD:
13	AP:
15	USR:

<b>Additional memory access class values for ARM CPUs</b>	
2	CP0
3	ICEbreaker
4	ETM
5	CP14
6	CP15
7	ARM logical
8	THUMB logical
9	ARM physical
10	THUMB physical
11	ETB
14	DAP:
<b>Additional memory access class values for PowerPC CPUs:</b>	
2	SPR
3	DCR
4	TLB
5	PMR
6	P: real mode address
7	P: virtual mode address

**Prototype:**

```
int T32_WriteMemory (dword address, int access, byte * buffer, int size);
```

**Parameters:**

```
address          ; target memory address to start write
access           ; memory access specifier
buffer           ; pointer to host buffer data area to write
size             ; number of bytes to write
```

**Returns:**

0 for ok, otherwise Error value

Writes data to target memory. The size of the data block is not limited. This function should be used to access variables and make other not time critical memory writes. The access flags define the memory access class and access method:

Bit 0...3	Memory Class, see <a href="#">T32_ReadMemory()</a>
Bit 6:	Set for emulation memory access (dual port access)
Bit 7:	Set to enable verify after write

**Example:**

```
unsigned char buffer[16];
...
error = T32_WriteMemory ( 0x100, 0xc0, buffer, 16 );
```

**Prototype:**

```
int T32_WriteMemoryPipe (dword address, int access, byte * buffer, int
size);
```

**Parameters:**

```
address          ; target memory address to start write
access           ; memory access flags
buffer           ; pointer to host buffer data area to write
size             ; number of bytes to write
```

**Returns:**

0 for ok, otherwise Error value

Writes data to target memory with pipelining. Pipelining means, that the memory write operation of the emulator is done in parallel to the downloading process. This speeds up the download. The return value of the function always refers to the previous Write command. The result of the last write command must be fetched by calling the function with size=0. The size of the data block is not limited. This function should be used to download an application program. The access flags define the memory access class and access method (see T32\_WriteMemory).

**Example:**

```
unsigned char buffer[1024];
...
error = T32_WriteMemoryPipe ( 0x400, 0xc0, buffer, 1024 );
```

**Prototype:**

```
int T32_ReadRegister (dword mask1, dword mask2, long *buffer);
```

**Parameters:**

```
mask1, mask2      ; register addressing mask
buffer            ; pointer to host buffer to catch register data
```

**Returns:**

0 for ok, otherwise Error value

The two 32-bit values `mask1` and `mask2` form a 64-bit bitmask. Each bit corresponds with one CPU register. Bit 0 of `mask1` is register #0, bit 31 of `mask2` is register #63. Registers are only read from the emulator, if their corresponding bit is set. The values of the registers are written in an array. Array element 0 is register 0, element 63 is register 63.

**Example:**

```
long buffer[64];

                        /* define register array */
error = T32_ReadRegister ( 0x3ff, 0x0, buffer );

                        /* read the first 10 registers */
```

**Prototype:**

```
int T32_WriteRegister(dword mask1, dword mask2, long *buffer);
```

**Parameters:**

```
mask1, mask2      ; register addressing mask  
buffer            ; pointer to host buffer containing the register data
```

**Returns:**

0 for ok, otherwise Error value

The two 32-bit values `mask1` and `mask2` form a 64-bit bitmask. Each bit corresponds with one CPU register. Bit 0 of `mask1` is register #0, bit 31 of `mask2` is register #63. Registers are only written, if their corresponding bit is set. The values of the registers are passed as an array. Array element 0 is register 0, element 63 is register 63.

**Example:**

```
long buffer[64];  
  
                /* define register array */  
buffer[1] = buffer [3] = 0x30f0;  
  
                /* write register 1 and 3 */
```

**Prototype:**

```
int T32_ReadPP(dword * pp);
```

**Parameters:**

```
pp                ; pointer to variable to catch the program pointer  
value
```

**Returns:**

0 for ok, otherwise Error value

This function reads the current value of the program pointer. It is only valid, if the application is stopped, i.e. the state of the ICE is "Emulation stopped" (see T32\_GetState). The program pointer is a logical pointer to the address of the next executed assembler line. In contrary to T32\_ReadRegister, this function is completely processor independent.

**Example:**

```
dword pp;  
error = T32_ReadPP ( &pp );  
printf ("Current Program Pointer: %x\n", pp);
```

**T32\_ReadBreakpoint****Read Breakpoints****Prototype:**

```
int T32_ReadBreakpoint ( dword address, int access, word * buffer,  
                        int size );
```

**Parameters:**

```
address           ; address to begin reading breakpoints  
access            ; memory access flags  
buffer            ; pointer to host buffer to catch breakpoint data  
size              ; number of addresses to read
```

**Returns:**

0 for ok, otherwise Error value

Read breakpoint and flag information from emulator. The `access` variable defines the memory class and access method (see `T32_ReadMemory`). The size of the range is not limited. The buffer contains 16-bit words in the following format:

Bit 0	execution breakpoint (Program)
Bit 1	HLL stepping breakpoint (Hll)
Bit 2	spot breakpoint (Spot)
Bit 3	read access breakpoint (Read)
Bit 4	write access breakpoint (Write)
Bit 5	universal marker a (Alpha)
Bit 6	universal marker b (Beta)
Bit 7	universal marker c (Charly)
Bit 8	read flag (ICE), if mapped or marker d (FIRE,ICD)
Bit 9	write flag (ICE), if mapped or marker e (FIRE,ICD)
Bit 10	implemented as ONCHIP (FIRE,ICD)
Bit 11	implemented as SOFT (FIRE,ICD)
Bit 12	implemented as HARD (FIRE,ICD)

**Example:**

```
unsigned short buffer[16];
error = T32_ReadBreakpoint ( 0x100, 0x40, buffer, 16 );
```



**Prototype:**

```
int T32_WriteBreakpoint ( dword address, int access, int breakpoint,
                          int size );
```

**Parameters:**

```
address          ; address to begin writing breakpoints
access          ; memory access flags
breakpoint      ; breakpoints to set or clear in area
size            ; number of addresses to write
```

**Returns:**

0 for ok, otherwise Error value

Set or clear breakpoints. The `access` variable defines the memory class and access method (see `T32_ReadMemory`). The size of the range is not limited. The breakpoint argument defines, which breakpoints to set or clear over the memory area:

Bit 0	execution breakpoint (Program)
Bit 1	HLL stepping breakpoint (Hll)
Bit 2	spot breakpoint (Spot)
Bit 3	read access breakpoint (Read)
Bit 4	write access breakpoint (Write)
Bit 5	universal marker a (Alpha)
Bit 6	universal marker b (Beta)
Bit 7	universal marker c (Charly)
Bit 8	Set to clear breakpoints

**Example:**

```
error = T32_WriteBreakpoint ( 0x100, 0x40, 0x19, 16 );
```

**Prototype:**

```
int T32_Step ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

Executes one single step on the emulator.

**Example:**

```
error = T32_Step ();
```

**Prototype:**

```
int T32_Step ( int mode );
```

**Parameters:**

```
mode                ; stepping mode
```

**Returns:**

0 for ok, otherwise Error value

Executes one step on the emulator. The mode parameter controls the stepping mode:

0	assembler step
1	HLL step
2	mixed = assembler step with HLL display

Bit 7 of mode defines step into or step over a function call

**Example:**

```
error = T32_StepMode (0x81);
```

Steps over a function call, halting on the next HLL line.

**Prototype:**

```
int T32_Go ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

Start realtime emulation. The function will return immediately after the emulation has been started. The T32\_GetState function can be used to wait for the next breakpoint. All other commands are allowed while the emulation is running.

**Example:**

```
error = T32_Go ();
```

**Prototype:**

```
int T32_Break ( void );
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

Stops the realtime emulation asynchronously.

**Example:**

```
error = T32_Break ();
```

**Prototype:**

```
int T32_GetTriggerMessage ( char message[128] );
```

**Parameters:**

```
message      ; pointer to an array of 128 characters to catch the message
```

**Returns:**

0 for ok, otherwise communication error value

When stopping on a read or write breakpoint (or equivalent), the trigger system generates an appropriate message. This message (as shown in the "Trigger" window), can be read with this function.

"message" must be a user allocated character array of at least 128 elements.

**Example:**

```
char message[128];  
  
error = T32_GetTriggerMessage (message);  
printf ("Trigger system reports: %s\n", message);
```

**Prototype:**

```
int T32_GetSymbol (char* symbol, dword* address, dword* size, dword*
access);
```

**Parameters:**

```
symbol           ; pointer to symbol name
address          ; pointer to variable to catch the symbol address
size            ; pointer to variable to catch the symbol size (if any)
access          ; pointer to variable to catch the symbol access class
```

**Returns:**

0 for ok, otherwise communication error value.

This function returns the symbol information for a specified symbol name. If the specified symbol was not found, *address*, *size* and *access* contains -1. Note, that there is not possible to get the information of non-static local variables (as they have no address).

This function can also be used to get the address of a source line.

## Example:

```
dword address, size, access;

char* symname = "variable";

        /* search for a variable called "variable" */
char* srcline = "\\file\\12";

        /* search for line 12 in file "file.c" */
error = T32_GetSymbol ( symname, &address, &size, &access );

        /* get information about a variable */
printf ("Symbol %s is located at 0x%x,\n", symname, address);
printf ("the variable with this symbol has a size of %d bytes.", size);
error = T32_GetSymbol ( srcline, &address, &size, &access );

        /* get information about a source line */
printf ("Source line 12 of file 'file.c' is located at 0x%x,\n",
address);
printf ("the line is compiled occupying %d bytes of code.", size);
```

**Prototype:**

```
int T32_GetSource (dword address, char filename[256], dword *line);
```

**Parameters:**

address	; address for which file and line are requested
filename	; output parameter, is set by the API function
line	; output parameter, is set by the API function

**Returns:**

0 for ok, otherwise Error value

With a given target address, this function calculates and gets the corresponding source filename and source line. filename **must** be an array of characters with at least 256 elements.

**Example:**

```
char filename[128];
dword line, curr_addr;

error = T32_ReadPP ( &curr_addr );           /* get program pointer */
error = T32_GetSource ( curr_addr, filename, &line );
printf ("Current Source: %s at line %d\n", filename, line);
```



**Prototype:**

```
int T32_GetSelectedSource (char filename[256], dword *line);
```

**Parameters:**

```
filename      ; pointer to an array of characters, output parameter  
line         ; pointer to source line, output parameter
```

**Returns:**

0 for ok, otherwise Error value

This function requests the source filename and line number of a selected source line in TRACE32 PowerView. The source line can be selected in any TRACE32 PowerView window containing source (e.g. "a.l" or "d.l"). If no previous selection was done, or if no source line is selected, the function returns with `filename` pointing to a NULL string.

`filename` **must** be an array of characters with at least 256 elements.

**Example:**

```
char filename[256];  
dword line;  
  
error = T32_GetSelectedSource ( filename, &line );  
if ( strlen (filename) )  
    printf ("Selected Source: %s at line %d\n", filename, line);  
else  
    printf ("No source line selected.\n");
```

**Prototype:**

```
int T32_AnaStatusGet (byte* state, long* size, long* min, long* max);
```

**Parameters:**

```
state           ; pointer to variable to catch the current analyzer
state size     ; pointer to variable to catch the trace buffer size
min number     ; pointer to variable to catch the minimum record
max number     ; pointer to variable to catch the maximum record
```

**Returns:**

0 for ok, otherwise communication error value

This function requests the state of the TRACE32 State Analyzer. This function is obsolete. New software should use the T32\_GetTraceState function.

“state” contains the current analyzer state:

0	analyzer is switched off
1	analyzer is armed
2	analyzer is triggered
3	analyzer recording broken

“size” contains the trace buffer size. It specifies the amount of records, which can be recorded, **not** the amount of records, which are actually stored in the buffer.

“min”, “max” contain the minimum and the maximum record number stored in the trace buffer. Note that the record numbers can be negative or positive.

**Example:**

```
byte state;
long size, min, max;

error = T32_AnaStatusGet (&state, &size, &min, &max);
printf ("State: %s\n", !state ? "off" : ((state == 1) ? "armed" :
    (( state == 3) ? "brokeed" : "unknown")));
printf ("Buffer size = %d records\n", size);
printf ("Minimum/Maximum record number: %d/%d\n", min, max);
```

**Prototype:**

```
int T32_AnaRecordGet (long recordnr, byte* buffer, int length);
```

**Parameters:**

```
recordnr      ; record number of record to read
buffer        ; byte array to catch the record information
length       ; number of bytes to read from record
```

**Returns:**

0 for ok, otherwise communication error value

This function reads the record information of one record of the Analyzer trace buffer. This function is obsolete. New software should use the T32\_ReadTrace function.

“recordnr” specifies the record number to read.

“buffer” contains the read record information (see below).

“length” specifies the number of bytes to read from the information into the buffer. This can be used to limit the amount of bytes transmitted and written into the buffer. If you specify “0”, all information will be transmitted; in this case allocate an array with 256 bytes at least.

The buffer will then contain the following data:

index	content		
0	return value:	0 = Ok -1 = no analyzer present -2 = invalid record number	
1	reserved		
2	physical access class:	lower 4 bits: higher 4bits:	1=Data 2=Program 3=First Cycle 4=res. 5=Breakpoint Cycle 6=res. 7=Write Cycle 8=Opfetch1 Cycle
3	reserved		
4-7	physical address (little endian)		

8-15	bus data (max. 8 bytes, depending on bus data width)
16	bus data width
17	bus access cycle (read/write/fetch, processor dependant)
18-19	status lines, processor dependant
20-27	time stamp (one bit equals 20/256 ns)
28/29	external trigger A/B inputs
30	logical access class: 1=Data 2=Program
31	reserved
32-35	logical address
rest	reserved

## Example:

```
int      i;
long     recordnr = 100;
long long time;
byte     buffer[256];

error = T32_AnaRecordGet (recordnr, buffer, 0);
if (!error && !buffer[0])          /* no error          */
{
    printf ("Address = 0x%02x%02x%02x%02x\n",
            buffer[7], buffer[6], buffer[5], buffer[4]);
    printf ("Data      = 0x");
    for (i = 0; i < buffer[16]; i++)
        printf ("%02x", buffer[8+i]);
    printf ("\n");
    printf ("Time      = 0x");
    time = 0;
    for (i = 7; i >= 0; i--)
    {
        printf ("%02x", buffer[20+i]);
        time += (long long) buffer[20+i] << i*8;
    }
    printf ("\n");
    time = time * 625 / 8000;          /* calculate nanoseconds */
    printf ("          = %d s, %d ms, %d us, %d ns\n",
            (long) (time / 1000000000L),
            (long) (time % 1000000000L / 1000000L),
            (long) (time % 1000000L / 1000L),
            (long) (time % 1000L));
}
```

**Prototype:**

```
int T32_GetTraceState (int tracetype, int* state, long* size, long* min,
long* max);
```

**Parameters:**

```
tracetype      ; type of trace and interpretation
state          ; pointer to variable to catch the current trace state
size          ; pointer to variable to catch the trace buffer size
min number    ; pointer to variable to catch the minimum record
max number    ; pointer to variable to catch the maximum record
```

**Returns:**

0 for ok, otherwise communication error value

This function requests the state of the selected Trace.

“tracetype” contains the trace method selection.

- |   |  |
|---|--|
| 0 | Trace (the Trace selected with Trace.METHOD command)                               |
| 1 | PowerIntegrator  |
| 2 | Trace raw data (same as 0, but no interpretation of trace data)                    |
| 3 | Trace funneled data (same as 0, but only decoding of funneled data for one source) |

“state” contains the current trace state:

- |   |                            |
|---|----------------------------|
| 0 | analyzer is switched off   |
| 1 | analyzer is armed          |
| 2 | analyzer triggered         |
| 3 | analyzer recording brokeed |

“size” contains the trace buffer size. It specifies the amount of records, which can be recorded, **not** the amount of records, which are actually stored in the buffer.

“min”, “max” contain the minimum and the maximum record number stored in the trace buffer. Note that the record numbers can be negative or positive.

### Example:

```
byte state;
long size, min, max;

error = T32_GetTraceState (0, &state, &size, &min, &max);
printf ("State: %s\n", !state ? "off" : ((state == 1) ? "armed" :
    (( state == 3) ? "broken" : "unknown")));
printf ("Buffer size = %d records\n", size);
printf ("Minimum/Maximum record number: %d/%d\n", min, max);
```

## T32\_ReadTrace

## Get One Record of Trace

### Prototype:

```
int T32_ReadTrace (int tracetype, long record, int n, unsigned long mask,
byte* buffer);
```

### Parameters:

tracetype	; type of trace and interpretation
record n	; record number of record to start reading from number of
mask	; records to read
buffer	; type of data to extract from the trace
	; byte array to catch the record information

### Returns:

0 for ok, otherwise communication error value

This function reads the information of one or more records from the trace buffer.

“tracetype” contains the trace method selection. See T32GetTraceState for the encoding.

“record” specifies the record number to read.

“n” is the number of records to read.

“mask” defines which information should be extracted. Each bit is related to a four byte chunk of data.

“buffer” contains the read record information. All data is stored in little endian format.

The buffer will then contain the following data:

bit group	byte	content
0	0	return value: 0=Ok -1=no analyzer present -2=invalid record number
0	1	reserved
0	2	reserved
0	3	reserved
1	0	external trace data 0 or flow trace data byte (only ETM V3, only row or funnel trace source)
1	1	external trace data 1 or flow trace control byte (only ETM V3, only row or funnel trace source) bit 2: TCNTL
1	2	trigger level
1	3	trigger flags
2	0...3	timestamp lower 32 bits (little endian) 0x40 -> 5ns 0x80 -> 10ns 0x100 -> 20ns 0x500 -> 100ns
3	0...3	timestamp upper 32 bits (little endian)
4	0...3	physical address (little endian)
5	0...3	physical address upper 32 bits (little endian)
6	0...3	physical access class and segment
7	0...3	reserved
8	0...3	logical address (little endian)
9	0...3	logical address upper 32 bits (little endian)
10	0...3	logical access class and segment
11	0...3	reserved



12	0...3	data 0...3	
13	0...3	data 4...7	
14	0	data bus mask (byte enables)	
14	1	cycle type information:	bit 1=Data bit 2=Program bit 3=First Cycle bit 4=res. bit 5=Breakpoint Cycle bit 6=res. bit 7=Write Cycle
14	2	data bus width	
14	3	reserved	
15	0...3	reserved	
16...31	0...3	logical analyzer or port channel data	

## Example:

```
int      i;
long     recordnr = 100;
long long time;
byte     buffer[256];

error = T32_ReadTrace (0, recordnr, 1, 0x710c, buffer);
if (!error && !buffer[0])      /* no error */
{
    printf ("Address = 0x%02x%02x%02x%02x\n", buffer[11], buffer[10],
            buffer[9], buffer[8]);
    printf ("Data      = 0x");
    for (i = 0; i < buffer[22]; i++)
        printf ("%02x", buffer[12+i]);
    printf ("\n");
    printf ("Time      = 0x");
    time = 0;
    for (i = 7; i >= 0; i--)
    {
        printf ("%02x", buffer[0+i]);
        time += (long long) buffer[0+i] << i*8;
    }
    printf ("\n");
    time = time * 625 / 8000;      /* calculate nanoseconds */
    printf ("          = %d s, %d ms, %d us, %d ns\n",
            (long) (time / 1000000000L),
            (long) (time % 1000000000L / 1000000L),
            (long) (time % 1000000L / 1000L),
            (long) (time % 1000L));
}
```

**Prototype:**

```
int T32_GetSocketHandle (SOCKET *soc);
```

**Parameters:**

```
soc           ; pointer to the handle of the socket created by the API  
              ; to communicate with TRACE32
```

**Returns:**

0 for ok, otherwise communication error value

This function returns a pointer to the handle of the socket created by the API to communicate with TRACE32. It could be used for example to register asynchronous notification for sending or receiving data on this socket.

**Example:**

Register the TRACE32 socket for asynchronous notification then a message is received on the socket.

```
SOCKET t32soc;  
T32_GetSocketHandle(&t32soc);  
if ( nr )  
    WSAAsyncSelect(t32soc, myHwnd, WM_ASYNC_SELECT, FD_READ);  
else  
    WSAAsyncSelect(t32soc, myHwnd, WM_ASYNC_SELECT, 0);
```

**Prototype:**

```
int T32_NotifyStateEnable (int event,void (*func)());
```

**Parameters:**

```
event          ; number of the event; to communicate with TRACE32  
func           ; pointer to a function
```

**Returns:**

0 for ok, otherwise communication error value

This function registers a callback function with the API that will be called by the API when the specified event occurs. For this mechanism to work, the user must ensure that the function T32\_CheckStateNotify is called periodically (e.g. in the windows main loop) because that will make the API reevaluate accumulated events.

“event” specifies the number of the event. Currently only the following event is specified through a constant:

T32\_E\_BREAK        Emulator break

“func” points to a function that is called when the event takes place.

**Example:**

Register the function targetHalted to be called whenever the emulator goes into state “break” (stopped).

```
if ( T32_NotifyStateEnable(T32_E_BREAK,targetHalted) )  
    printf ("Notify Break: Could not initialize! \n");  
else  
    printf ( "Notify Break Enable.\n");
```

**Prototype:**

```
int T32_CheckStateNotify (unsigned param1);
```

**Parameters:**

```
param1      ; parameter 1 of registered func at T32_NotifyStateEnable
```

**Returns:**

0 for OK, otherwise communication error value

This function makes the API reevaluate events accumulated since the last call to T32\_CheckStateNotify. If a callback function for any of these events was registered with T32\_NotifyStateEnable, the callback function is executed as callback(param1). The parameter is used independently of the event type and is intended for passing generic parameters like application handles etc.

**As the CAPI does not have an own thread, it is the application program's responsibility to periodically call this function.**

Example:

The typical Windows callback routine for an application which also handles the asynchronous notification of a socket.

```
long CALLBACK MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;                /* window handle */
UINT message;             /* type of message */
WPARAM wParam;           /* additional information */
LPARAM lParam;           /* additional information */
{
    switch (message)
    {
        case WM_COMMAND:    /* message: command from application menu */
            break;
        case WM_ASYNC_SELECT:
            if ( WSAGETSELECTERROR(lParam) != 0 )
                break;// error receiving select notification
            switch ( WSAGETSELEVENT(lParam) )
            {
                case FD_READ:
                    T32_CheckStateNotify(&apphandle);
                    break;
            }
            break;
        case WM_DESTROY:    /* message: window being destroyed */
            break;
        default:             /* Passes it on if unprocessed */
            return ( DefWindowProc(hWnd, message, wParam, lParam) );
    }
    return (0);
}
```

This chapter describes all functions available for direct access to the JTAG TAP controller. There are two possible modes to access the TAP controller, the Single Access Mode and the Bundled Access Mode. For a sequence of TAP accesses (e.g. to read memory), the Bundled Access Mode is recommended.

The functions `T32_TAPAccessShiftIR`, `T32_TAPAccessShiftDR` and `T32_TAPAccessDirect` are provided for JTAG access. These functions need a handle to access the TAP controller. For Single Access Mode, two predefined Handles are available, which control the behavior of the debugger after the API access:

Handle	Effect
<code>T32_TAPACCESS_HOLD</code>	All debugger actions concerning the TAP controller will be suspended. The API has exclusive access to the JTAG port.
<code>T32_TAPACCESS_RELEASE</code>	Allows the debugger to access the TAP controller after this API access

For Bundled Access Mode, the TAP access handle must be acquired by calling `T32_TAPAccessAlloc`. All IR, DR and direct accesses will be stored, instead of being executed immediately. Those bundled accesses are executed with a call to `T32_TAPAccessExecute` in the given order. While a bundled access is executed, the API holds exclusive access to the TAP controller.

**Prototype:**

```
int T32_TAPAccessSetInfo(int irpre, int irpost, int drpre, int drpost,
    int tristate, int tapstate, int tcklevel, int slave);
```

**Parameters:**

```
irpre      ; Number of instruction register bits of all cores in the
            ; JTAG chain between the dedicated core and the TDO signal
irpost     pin
            ; Number of instruction register bits of all cores in the
drpre      JTAG
            ; chain between TDI signal and the dedicated core
drpost     ; Number of cores in the JTAG chain between the dedicated
            core
            ; and the TDO signal (one data register bit per core which is
            ; in BYPASS mode)
tristate   ; Number of cores in the JTAG chain between the TDI signal
            and
            ; the dedicated core (one data register bit per core which is
tapstate   ; in BYPASS mode)
            ; TRUE, if more than one debugger is connected to JTAG port.
            ; With this option, the debugger switches to tristate mode
            ; after each access.
            ; In multi-debugger mode, this parameter specifies the state
            ; of the TAP controller, which is expected when the debugger
tcklevel   ; takes control and set before the debugger switches to
            ; tristate mode. This value has to be identical for all
slave      ; debuggers connected to this JTAG port. See table below for
            ; a list of possible states.
            ; In multi-debugger mode, this is the level of the TCK signal
            ; when all debuggers are tristated.
            ; In multi-debugger mode, only one debugger is allowed to
            ; control nTRST, nRESET. All others have to set this value to
            ; TRUE.
```



## Returns:

0 for ok, otherwise Error value

Values for tapstate:

0	Exit2-DR	8	Exit2-IR
1	Exit1-DR	9	Exit1-IR
2	Shift-DR	10	Shift-IR
3	Pause-DR	11	Pause-IR
4	Select-IR-Scan	12	Run-Test/Idle
5	Update-DR	13	Update-IR
6	Capture-DR	14	Capture-IR
7	Select-DR-Scan	15	Test-Logic-Reset

## Example:

```
TDI ----> TAP_A ----> TAP_B ----> MyTAP ----> TAP_C ----> TDO

IRLEN(TAP_A) = 3 bits
IRLEN(TAP_B) = 5 bits
IRLEN(TAP_C) = 6 bits

IRPRE = IRLEN(TAP_C) = 6
IRPOST = IRLEN (TAP_A) + IRLEN (TAP_B) = 8
```

**Prototype:**

```
int T32_TAPAccessShiftIR(T32_TAPACCESS_HANDLE connection,
    int numberofbits, unsigned char* poutbits, unsigned char* pinbits);
```

**Parameters:**

```
connection      ; TAP access handle (see 4.3)
numberofbits    ; amount of bits to scan
poutbits        ; buffer containing data scanned into the TAP
                 controller,
pinbits         ; or NULL to scan in Zeros
                 ; buffer for data to be scanned out of the TAP
                 ; controller, or NULL to discard the received data
```

**Returns:**

0 for ok, otherwise Error value

Use this function to scan data through the Instruction Register

**Example:**

```
unsigned char status;
unsigned char tap_instr = TAP_STATUS;

T32_TAPAccessShiftIR (T32_TAPACCESS_RELEASE, 8, &tap_instr, &status);
```

**Prototype:**

```
int T32_TAPAccessShiftDR(T32_TAPACCESS_HANDLE connection,
    int numberofbits, unsigned char* poutbits, unsigned char* pinbits);
```

**Parameters:**

```
connection      ; TAP access handle (see 4.3)
numberofbits    ; amount of bits to scan
poutbits        ; buffer containing data scanned into the TAP
                 controller,
pinbits         ; or NULL to scan in Zeros
                 ; buffer for data to be scanned out of the TAP
                 ; controller, or NULL to discard the received data
```

**Returns:**

0 for ok, otherwise Error value

Use this function to scan data through the Data Register

**Example:**

```
// Retrieve the PVR value (PowerPC)
unsigned char status;
unsigned char pvrnr[4];
unsigned char tap_instr = TAP_COP_PVR;

T32_TAPAccessShiftIR (T32_TAPACCESS_HOLD, 8, &tap_instr, &status);
T32_TAPAccessShiftDR (T32_TAPACCESS_RELEASE, 32, NULL, pvrnr);
// Write Zeros
```

**Prototype:**

```
int T32_TAPAccessDirect(T32_TAPACCESS_HANDLE connection, int nbytes,
    byte * poutbytes, byte * pinbytes);
```

**Parameters:**

```
connection      ; TAP access handle (see 4.3)
nbytes          ; size in bytes of the array psignals
poutbytes       ; array containing direct access commands
pinbytes        ; array receiving the results of the direct access
                 ; commands
```

**Returns:**

0 for ok, otherwise Error value

The primary use of this function is to directly access the JTAG port, such as toggling HRESET or reading TDO, via a variety of commands.

The poutbytes buffer can also contain multiple commands. Any command consists of one or more bytes. The size of the return value is always identical with the command size.

For a direct access to the JTAG port pins, commands can be generically generated. All commands for read accesses are predefined:

**JTAG signals:**

T32_TAPACCESS_TDO	T32_TAPACCESS_TDI
T32_TAPACCESS_TMS	T32_TAPACCESS_TCK
T32_TAPACCESS_nTRST	

**System signals:**

T32_TAPACCESS_nRESET	T32_TAPACCESS_nRESET_LATCH
T32_TAPACCESS_VTREF	T32_TAPACCESS_VTREF_LATCH

**Debugger related signals:**

T32\_TAPACCESS\_nENOUT

The two latches display any occurrence of RESET/VTREF fail since the last check. The functionality of read accesses depends on the used debugger and target.

nENOUT enables the output driver of the debug cable (negative logic).

Write accesses are generated by OR-ing the corresponding read command with one of the following values:

T32_TAPACCESS_SET_0 T32_TAPACCESS_SET_LOW	Sets Signal to logical LOW
T32_TAPACCESS_SET_1 T32_TAPACCESS_SET_HIGH	Sets Signal to logical HIGH
T32_TAPACCESS_SET(x)	Sets Signal to value x

The returned result of a write command is identical with that of the corresponding read command.

Additional Commands:

Command (Byte 0)	Cmd. Size in Bytes	Byte1
T32_TAPACCESS_SLEEP_MS	2	Time in msec
T32_TAPACCESS_SLEEP_US	2	Time in usec
T32_TAPACCESS_SLEEP_HALF_CLOCK	1	No parameter. The debugger waits for an half JTAG clock cycle. <b>NOTE:</b> This command does not work with return clock from target (RTCK). Clock accurate arbitrary shifts should be done by <a href="#">"T32_TAPAccessShiftRaw RAW JTAG Shifts"</a> (api_remote.pdf).

**NOTE:** The existence and functionality if direct access commands may vary depending on the used debugger and/or target hardware.

## Example:

```
// reset target
unsigned char commands[8];
unsigned char result[8];
unsigned char hreset_state;

commands[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
commands[1] = T32_TAPACCESS_nRESET | T32_TAPACCESS_SET_0;
commands[2] = T32_TAPACCESS_SLEEP_MS;
commands[3] = 50; // Wait 50 ms
commands[4] = T32_TAPACCESS_nRESET | T32_TAPACCESS_SET_1;
commands[5] = T32_TAPACCESS_SLEEP_MS;
commands[6] = 50; // Wait 50 ms
commands[7] = T32_TAPACCESS_nRESET;

T32_TAPAccessDirect (T32_TAPACCESS_RELEASE, 8, commands, result);

hreset_state = result[7];
```

**Prototype:**

```
int T32_TAPAccessShiftRaw(T32_TAPACCESS_HANDLE connection,
    int numberofbits, byte * pTMSBits, byte * pTDIBits,
    byte * pTDOBits, int options);
```

**Parameters:**

connection	; TAP access handle (see 4.3)
numberofbits	; defines how many TCK clock cycles the shift is long
pTMSBits	; TMS bit pattern. May be NULL in case no specific pattern shall be shifted.
pTDIBits	; TDI bit pattern. May be NULL in case no specific pattern shall be shifted.
pTDOBits	; array to store TDO answer. May be NULL if the result shall not be recorded
options	; shift option bit mask (see below)

**Returns:**

0 for ok, otherwise Error value

This function should be used to send/receive arbitrary TDI/TMS/TDO pattern. The buffers are considered bit wise beginning with the first byte e.g. pTDIBits = 0x03 0x04 will shift out 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 for TDI.

It is possible to pass a NULL pointer for any of the parameters. The advantage of this method is that less data needs to be transferred between debug box and API. By setting all communication arrays to NULL the amount of shifted bits is not limited. The receive/send data pattern size are limited to a size of (T32\_TAPACCESS\_MAXBITS - 64) bits. If TMS and TDI are transferred both the maximum pattern size is limited to 1/2 \* (T32\_TAPACCESS\_MAXBITS - 64). If TDI or TMS are left out the pattern can be defined by the options parameter:

For a direct access to the JTAG port pins, commands can be generically generated. All commands for read accesses are predefined:

**Pattern Options TMS:**

SHIFTRAW_OPTION_TMS_ZERO	Shifts TMS = 0
SHIFTRAW_OPTION_TMS_ONE	Shifts TMS = 1
SHIFTRAW_OPTION_LASTTMS_ONE	Shifts TMS = 0, except for the last cycle where TMS = 1

## Pattern Options TDI:

SHIFTRAW_OPTION_TDI_ZERO	Shifts TDI = 0
SHIFTRAW_OPTION_TDI_ONE	Shifts TDI = 1
SHIFTRAW_OPTION_TDI_LASTTDO	Shifts TDI pattern that equals last read back TDO (where pTDOBits where defined). Please ask LAUTERBACH support if that feature shall be extended.

### Example 1:

```
int TAPAccessShiftRaw_Test_Hold()
{
    unsigned char pTDI[1];
    unsigned char pTMS[1];
    unsigned char pTDO[1];
    int err = 0;

    /*Drive from Run/Test Idle to Shift/IR ( 1 1 0 0 )*/
    pTMS[0] = 0x3;
    if (err = T32_TAPAccessShiftRaw(T32_TAPACCESS_HOLD , 4 , pTMS, 0 , 0,
        SHIFTRAW_OPTION_NONE))
        goto error;

    /*Shift 0x5 / 5-Bit TAP and read back response - Drive to Exit1-IR*/
    pTDI[0] = 0x6;
    if (err = T32_TAPAccessShiftRaw(T32_TAPACCESS_HOLD , 5 , 0, pTDI ,
        pTDO,
        SHIFTRAW_OPTION_LASTTMS_ONE))
        goto error;

    /*Drive From Exit1-IR to RUN-Test/Idle ( 1 0 )*/
    pTMS[0] = 0x1;
    if (err = T32_TAPAccessShiftRaw(T32_TAPACCESS_HOLD , 2 , pTMS, 0 , 0,
        SHIFTRAW_OPTION_NONE))
        goto error;

error:
    T32_TAPAccessRelease();
    return err;
}
```

The T32\_TAPAccessShiftRaw function can be combined with the T32\_TAPAccessExecute mechanism to speed up multiple pattern calls. Make sure that the pTDOBits pointer is valid until T32\_TAPAccessExecute is called.



## Example 2:

```
int TAPAccessShiftRaw_Test_Execute()
{
    unsigned char pTDI[1];
    unsigned char pTMS[1];
    unsigned char pTDO[1];
    int err = 0;

    T32_TAPACCESS_HANDLE handle = T32_TAPAccessAlloc ();
    /*Drive from Run/Test Idle to Shift/IR ( 1 1 0 0 )*/
    pTMS[0] = 0x3;
    if (err = T32_TAPAccessShiftRaw(handle , 4 , pTMS, 0 , 0,
        SHIFTRAW_OPTION_NONE))
        goto error;
    /*Shift 0x5 / 5-Bit Tap and read back response - Drive to Exit1-IR*/
    pTDI[0] = 0x6;
    if (err = T32_TAPAccessShiftRaw(handle , 5 , 0, pTDI , pTDO,
        SHIFTRAW_OPTION_LASTTMS_ONE))
        goto error;
    /*Drive From Exit1-IR to RUN-Test/Idle ( 1 0 )*/
    pTMS[0] = 0x1;
    if (err = T32_TAPAccessShiftRaw(handle , 2 , pTMS, 0 , 0,
        SHIFTRAW_OPTION_NONE))
        goto error;

    if (err = T32_TAPAccessExecute(handle, T32_TAPACCESS_HOLD))
        goto error;
error:
    T32_TAPAccessRelease();
    T32_TAPAccessFree(handle);
    return err;
}
```

**Prototype:**

```
T32_TAPACCESS_HANDLE T32_TAPAccessAlloc();
```

**Parameters:**

none

**Returns:**

Handle for bundled TAP accesses

Use this function to retrieve a handle for bundled TAP accesses. The execution sequence associated with a handle can be used multiple times.

**Example:**

```
unsigned char status;
unsigned char pvrnr[4];
unsigned char tap_instr = TAP_COP_PVR;

T32_TAPACCESS_HANDLE handle = T32_TAPAccessAlloc ();

T32_TAPAccessShiftIR (handle, 8, &tap_instr, &status);
T32_TAPAccessShiftDR (handle, 32, NULL, pvrnr);
T32_TAPAccessExecute (handle, T32_TAPACCESS_RELEASE);

T32_TAPAccessFree (handle);
```

**Prototype:**

```
int T32_TAPAccessFree(T32_TAPACCESS_HANDLE connection);
```

**Parameters:**

```
connection ; TAP access handle (see 4.3)
```

**Returns:**

0 for ok, otherwise Error value

Use this function to release the handle returned by T32\_TAPAccessAlloc when it is no longer needed.

**Example:**

see 4.3.5 T32\_TAPAccessAlloc for an example

**T32\_TAPAccessExecute****Execute a Bundled TAP Access****Prototype:**

```
int T32_TAPAccessExecute(T32_TAPACCESS_HANDLE connection,  
                        T32_TAPACCESS_HANDLE connectionhold);
```

**Parameters:**

```
connection ; Handle for a bundled TAP access  
connectionhold ; TAP access handle (see 4.3)
```

**Returns:**

0 for ok, otherwise Error value

Use this function, to execute all DR, IR and direct TAP accesses associated with given handle.

**Example:**

see 4.3.5 T32\_TAPAccessAlloc for an example

**Prototype:**

```
int T32_TAPAccessRelease();
```

**Parameters:**

none

**Returns:**

0 for ok, otherwise Error value

If debugger accesses are suspended due to a IR, DR, direct access or the T32\_TAPAccessExecute call with the access handle T32\_TAPACCESS\_HOLD, use this function to resume debugger accesses.

**Example:**

```
// Retrieve the PVR value (PowerPC)
unsigned char status;
unsigned char pvrnr[4];
unsigned char tap_instr = TAP_COP_PVR;

T32_TAPAccessShiftIR (T32_TAPACCESS_HOLD, 8, &tap_instr, &status);
T32_TAPAccessShiftDR (T32_TAPACCESS_HOLD, 32, NULL, pvrnr);

// At this point, the debugger is still locked

T32_TAPAccessRelease ();
```

Document version control:

<b>Version</b>	<b>Date</b>	<b>Change</b>
4.1		new commands T32_TAPACCESS_SLEEP_MS, T32_TAPACCESS_SLEEP_US, T32_TAPACCESS_SLEEP_HALF_CLOCK
4.0SP1	15.03.05	Corrected endianness documentation in T32_GetCpuInfo()
4.0	20.08.04	Added Multi-Debugger access: T32_GetChannel*(), T32_SetChannel()
3.6	30.09.03	T32_TAPACCESS_nENOUT added.
3.5	15.03.03	New Section 4.3 ICD TAP Access API Functions:
3.4	26.10.99	Changes in handling of big messages Correction of error at T32_GetMessage without T32_Attach T32_Cmd() can now handle long command names e.g. data.load instead of d.l
3.3	12.04.99	New Functions: T32_GetSocketHandle(SOCKET *soc) T32_NotifyStateEnable(int event,void (*func>()) T32_CheckStateNotify(unsigned param1, unsigned param2)
3.2	01.12.98	Format of logical address class of T32_AnaRecordGet changed
3.1	13.03.98	T32_GetSymbol description extended T32_GetTriggerMessage, T32AnaStatusGet, T32AnaRecordGet added
3.0	03.03.98	Document format changed (LWP to FM)
2.1	18.12.96	T32_GetSymbol added
2.0	28.10.96	Incompatibility to previous versions in socket communications T32_StepMode added
1.4	23.09.96	PACKLEN definitions added Chapter numbering added
1.3	15.08.96	T32_GetSelectedSource added Table of Versions added
1.2	29.05.96	Overall revision

