API for Remote Control and JTAG Access in C

Release 09.2024

MANUAL

API for Remote Control and JTAG Access in C

API for Remote Control and JTAG Access in C

Version 05-Oct-2024

History

20-Jun-2023	Updated T32_APILock and T32_APIUnlock.
21-Dec-2022	Marked T32_ReadMemory, T32_WriteMemory, T32_ReadRegister, T32_WriteRegister, T32_ReadBreakpoint, and T32_WriteBreakpoint as deprecated functions.
20-May-2022	New parameter `T32_E_Go` for the function T32_NotifyStateEnable. New function T32_NotifyGoEnable.
14-Jul-2020	Remote API via TCP sockets.

Licensing Terms

The TRACE32 Application Programming Interface for Remote Control and JTAG Access (“Remote API”) contains source code for the client interface, which is copyright by Lauterbach. These licensing terms and conditions apply to all files referred to in this document.

You may:

- share the original C source code of the TRACE32 Remote API with others (e.g. in a public repository)
- use the original source code of the Remote API in your own software (commercial and non-commercial)
- modify the original source code if necessary for compilation or integration into your product or a library used by it
- port the Remote API source code to other computer languages

You may not:

- sell or sub-license the original source code of the TRACE32 Remote API
- modify the original source code, or any derived works, in a away that changes or extends the APDUs (Application Protocol Data Units) that it produces
- distribute any modified source code to others
- implement the host/server part of the Lauterbach TRACE32 Remote API in your own product (if you think you need this, please contact us to negotiate different licensing terms for this)

You have to:

- include these Licensing Terms in any derived works
- inform Lauterbach if you use the original source code, or any derived works, in a commercial product

Disclaimer: The API source code is designed to remote-control Lauterbach TRACE32 software. We provide this code “as is” without any implicit or explicit warranties, and without taking responsibility for its correctness or for its fitness for a specific purpose.

Introduction

Release Information

Release 4.0, shipped from 01-SEP-2004, includes the ability to connect to several debuggers at once (multi-core debugging).

The Remote API via TCP shipped from 01-SEP-2020 is always able to handle multiple clients.

Compatibility

Lauterbach ensures backward compatibility of the API.

Backward compatibility means, that application built with one release of the API will remain working on both, future versions of the API and future versions of the main TRACE32 software. Future releases of the API and/or the TRACE32 software will extend or replace some functionality, but will not break previous functionality.

The compatibility applies to:

- The C function interface.
  The functions listed in this manual will keep their calling conventions and the functionality described here.

- The socket stream.
  The binary data sent over the socket connection will keep functioning.

The compatibility does **not** apply to:

- The composition of the API functions in the source files.
  The coding of the function may change completely, keeping the above compatibilities.

- API internal data structures and representations.
  Variables and data structures, that are not exposed in the manual, may be changed without further notice. When accessing data structures of the API, use only the access functions mentioned herein.

Related Tutorials

For a video tutorial about the TRACE32 Remote API, visit:
support.lauterbach.com/kb/articles/trace32-remote-api

System Configuration Overview

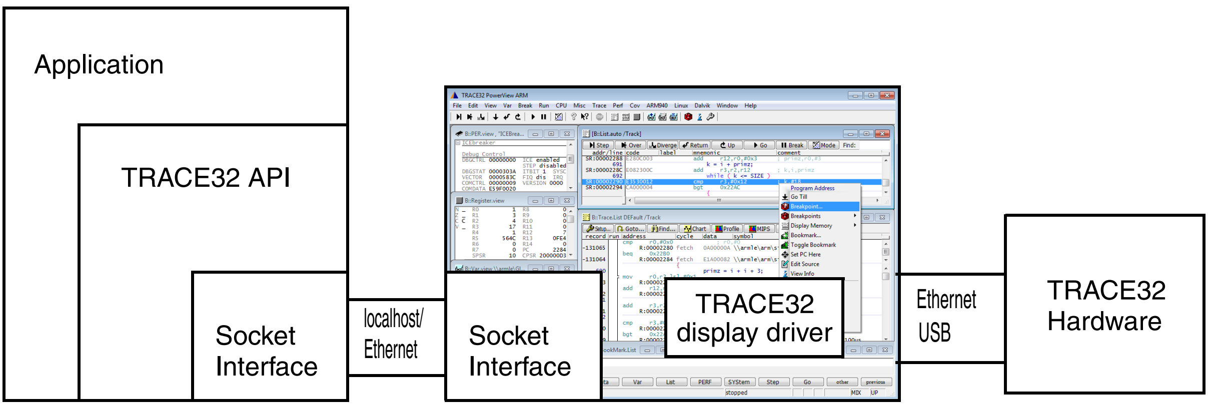
The TRACE32 PowerView software contains an external control interface. The TRACE32 Application Programming Interface (further referred to as API) gives external applications the possibility to control the debugger and the program run by the debugger.

The API is built as a plain C source library with a C function interface to the controlling application. Alternatively to the C source files, a prebuilt libraries for Linux and Windows, that export the same function set, are available.

The API communicates with the TRACE32 application (not with the TRACE32 debug interface itself!) using a socket interface. Starting from the TRACE32 release 09.2020, the API supports per default TCP socket streams. Previous TRACE32 versions only support a communication via UDP sockets.

This is the command chain using TRACE32 API:

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



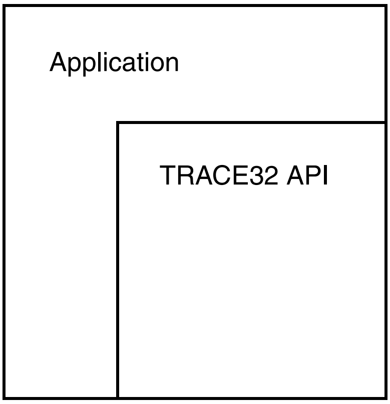
Restrictions in Demo Mode

The TRACE32 Remote API is blocked in “demo mode”, i.e. if you do not have a valid TRACE32 license. You will not be able to create successful connections between the API and TRACE32.

If you need to evaluate the API without having a full license, contact Lauterbach for an evaluation license of your TRACE32 system.

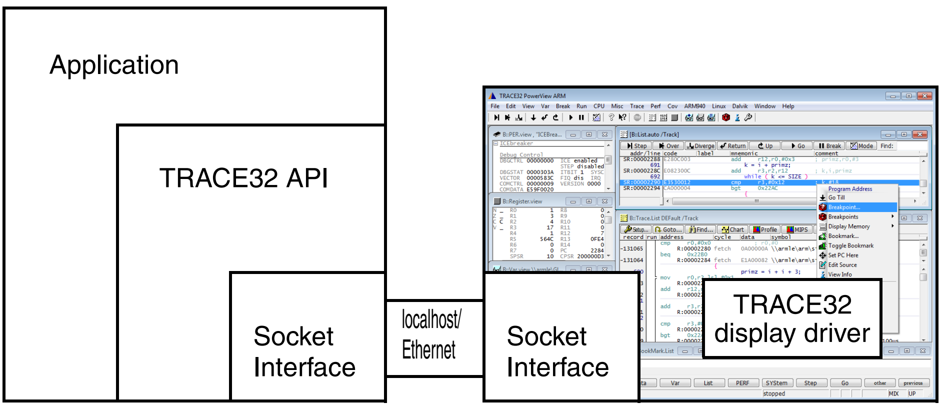
Interfaces

Application --> TRACE32 API

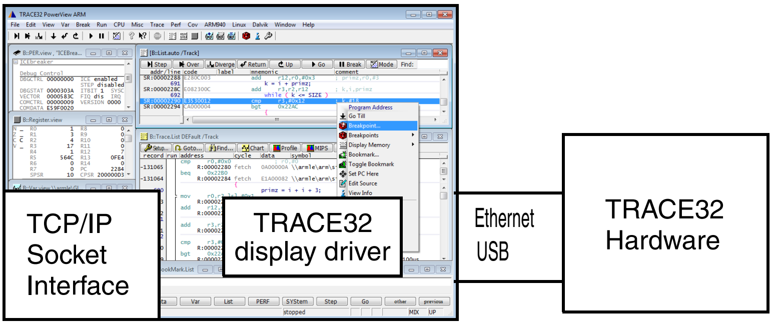


The application uses the API as ordinary C functions. The API is linked to the application at the usual linking stage. The API functions are not thread safe. If the application uses threads, it has to lock the functions against reentrancy.

TRACE32 API --> TRACE32 display driver



The communication to the TRACE32 software is implemented as a socket interface. The controlling application (compiled/linked with the API) and the debugger software can reside on two different hosts, using socket connections for communication.



The TRACE32 PowerView debugger software processes and routes the API requests to the TRACE32 hardware. This interface is the one, you chose for your debugger. E.g. it could be Ethernet or USB.

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

Operation of 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.

Conventions for Target Memory Access

When using Remote API functions to read and write *target memory* (e.g. `T32_ReadMemory()`, `T32_WriteMemory()`), it is necessary to follow the TRACE32 conventions given below.

Byte-Addresses

If not explicitly changed (see below), the address parameter for reading and writing target memory **always** is a “byte” (octet) address, independently of the target architecture’s native memory width. This implies that

- For machines that are byte-addressed (i.e. natively address single bytes like x86) the byte address corresponds to the native address. On these machines incrementing the address by 1 yields the *next byte in memory*.
- For machines using word-addresses (i.e. natively address *memory words* like many DSPs) *the byte address for use with the TRACE32 remote API* is calculated multiplying the word address with the native memory width (in bytes). On these machines incrementing the native address by 1 yields the *next word in memory.*
- Accessing peripheral registers or special purpose registers that are not byte addresses (e.g. Arm CP15 or PowerPC SPR) need an address correction that multiplies the register number by the byte width of the register access class. E.g. if `Data.dump SPR:0x10` shows 32bit for each register number (= SPR address), the corresponding API address is 0x10*4.

The address objects may be set to be used with other addressing modes, by setting the MAU (minimum addressable unit) with `T32_SetAddressObjSizeOfMau()`.

Byte-Size

The size parameter is always given in bytes, independently of the target architectures native memory width.

Examples:

```
// 1) read 16 bytes from address D:0x100 on a byte-addressed machine
//    (e.g. x86, MicroBlaze, PPC,...)
uint8_t buffer[16];
error = T32_ReadMemory (0x100, 0x0 /*D:*/, buffer, 16 /*bytes*/);

// 2) read 16 bytes from address D:0x100 on a word-addressed machine
//    using 16bit words (e.g. C2000)
uint8_t buffer[16];
error = T32_ReadMemory (0x100 * 2 /*16bit*/, 0x0, buffer, 16 /*bytes*/);

// 3) reading CP15 register number 0x101 on Arm32
uint8_t buffer[4];
error = T32_SetMemoryAccessClass("C15");
error = T32_ReadMemory (0x101 * 4 /*32bit*/, 0x0, buffer, 4 /*bytes*/);
```

Memory Access Class Specifiers:

The type of memory to access and the method to use are specified by so-called *memory class specifiers*. Among other memories these allow to address data and program memory (especially in DSPs), debugger virtual memory, bypass address translation ("absolute access") etc.

In the functions for memory access, the access *function parameter* is *only* used, if the access class is *not* set with **T32_SetMemoryAccessClass** (see there). Otherwise the access parameter is ignored and the access class set with **T32_SetMemoryAccessClass** is used.

Please refer to the following (non-exhaustive) list for the codes of various memory class specifiers. For additional information please contact Lauterbach support.

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

Additional memory access class values for Arm CPUs	
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:

Additional memory access class values for PowerPC CPUs:	
2	SPR
3	DCR
4	TLB
5	PMR
6	P: real mode address
7	P: virtual mode address

Additional memory access class values for ARC CPUs:	
2	AUX

Additional memory access class values for x86 CPUs:	
2	D: linear address
3	P: linear address
4	IO
5	MSR

Building an Application with API

The main root of the API are the C source files, which are available in the TRACE32 system directory under `~~/demo/api/capi/src`. Those are written to work with several compilers and operating systems, such as Windows, Linux, etc.

Alternatively libraries are available, which are just a prebuilt version of the source files, and export the same function sets. The libraries are located in the TRACE32 system directory under `~~/demo/api/capi/dll`.

Lauterbach recommends to use the source files. This chapter describes how to build an API based application using the sources.

Demo applications using the API can be found under `~~/demo/api/capi/test`. Please refer to the `readme.txt` file for more information.

API Files

The API consists of the following source and header files:

- **t32.h**

  This header file contains the necessary API definitions and function prototypes.

- **hremote.c**

  All API functions are coded in this source file

- **hlinknet.c**

  This file contains and handles the UDP socket interface to the TRACE32 debugger software.

- **tcpsimple2.c, tcpsimple2.h, t32nettcp.c, t32nettcp.h**

  These files contain and handle the TCP socket interface to the TRACE32 debugger software. These files are provided starting from TRACE32 release 09.2020.

tcpsimple2.[ch] and **t32nettcp.[ch]** are alternatives to **hlinknet.c** so either use **hlinknet.c**, so either use **tcpsimple2.[ch]**, **t32nettcp.[ch]** or **hlinknet.c**.

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"
```

before using any API definition or function. Please be aware, that the API calls are *neither* reentrant *nor* thread-safe. When using parallel threads in your application, please ensure locking the API calls against each other.

When compiling and linking the application, the API files must be handled as normal source components of the application. Assuming that the application is coded in a file called "`application.c`" and your C compiler is called "`cc`", compilation could look like this:

```
cc -c tcpsimple2.c
cc -c t32nettcp.c
cc -c hremote.c
cc -c application.c
```

The linker run is then invoked with:

```
ld -o application tcpsimple2.o tcpsimple2.o t32nettcp.o application.o
```

assuming the linker name is "`ld`" and the object extension is "`o`".

Logging the API Calls

The API contains a log mechanism that allows to log all API calls to a dedicated file. To use this logging, the API source code must be compiled with the preprocessor macro `ENABLE_APILOG`, e.g.:

```
cc -DENABLE_APILOG -c hremote.c
```

To activate the logging, set the environment variable `T32APILOGFILE` to the path and filename that should collect the log. E.g.:

```
set T32APILOGFILE=C:\temp\t32apilog.txt
```

The log file contains a timestamp, the API call with its parameters and the return of the API call. The format of the file is not fixed and may change slightly with different API versions.

Communication Setup

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 TRACE32 configuration file, e.g. "`config.t32`". If you are using Windows and T32Start application to start the TRAC32 software, you need to open the configuration at “advanced settings” where you can select “Use Port: yes” in the “API Port” folder. The automatically created configuration file (e.g. `C:\temp\userT32_1000123.t32`) will have the necessary lines automatically.

```
                  <- mandatory blank line
;T32 API Access (UDP)
RCL=NETASSIST
PORT=20000
PACKLEN=1024
                  <- mandatory blank line
;T32 API Access (TCP)
RCL=NETTCP
PORT=20000
                  <- mandatory blank line
```

`PACKLEN` specifies the maximum package length in bytes for the UDP socket communication. It must not be bigger than 1024 and must fit to the value defined by **T32_Config()**.

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

See also **“RCL Function”** (ide_func.pdf).

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.
- **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.

API Functions

Error Codes

If not otherwise specified, the TRACE32 Remote API functions return an error code. The error code is copied into a global variable called `T32_Errno`. A return value of 0 encodes “no error” (`T32_OK`).

The error codes are listed in the `t32.h` file, which can be found in the C source distribution of the API files. See “**Building an Application with API**”.

Generic API Functions

T32_Config — Configure Driver

Prototype:

```
int T32_Config ( const char *string1, const 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. The following settings are available:

`NODE=`	Defines on which host the TRACE32 display driver runs. Default is "localhost".
`PACKLEN=`	Specifies the maximum data package length used for UDP. The value must not be bigger than 1024 and must fit to the value defined in the "`config.t32`" file. No operation for TCP.

`PORT=`	Defines the TCP/UDP port for sending. Default is 20000. Be sure that these settings fit to the RCL settings in the "`config.t32`" file.
`TIMEOUT=`	Timeout for UDP. Defines the communication timeout of API functions in seconds. Default is 5s. If TRACE32 does not answer within this time, the API function returns with T32_COM_RECEIVE_FAIL. No operation for TCP.
`HOSTPORT=`	Defines the UDP port for receiving. By default, this is assigned automatically. Only use this setting if you really need to set a specific receive port. No operation for TCP.

Usually, the following commands will be used:

```
NODE=localhost
PORT=20000
```

Example:

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

T32_Init — Initialize driver and connect

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:

```
if ((error = T32_Init())!=0) {
  /* handle error */
}
if ((error = T32_Attach(T32_DEV_ICD) != 0) {
  /* handle error */
}
```

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 funtions.

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

Example:

```
error = T32_Attach ( T32_DEV_ICD );
```

T32_Nop — Send Empty Message

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 ();
```

T32_Ping — Send Ping Message

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 ( const char *command );
```

Parameters:

```
command              ; TRACE32 command to execute
```

Returns:

0 for ok, otherwise Error value

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

NOTE:	When executing a script via the "**DO**" command, the function will return immediately, not waiting for the end of the script. You may use **T32_GetPracticeState()** to actively wait for the script ending.

Negative error values indicate a communication problem between the debugger and the API.
An positive error value indicates that the command was not accepted.
Errors caused while executing the command are not reported, to retrieve further error information, please use the call **T32_GetMessage()** and check the message type.

Example:

```
error = T32_Cmd ("Data.Set %Long 0x12200 0x033FFC00");
```

Prototype:

```
int T32_Cmd_f ( const char *command, ... );
```

Parameters:

```
command              ; PRACTICE command to execute,
                     ; with format specifiers
```

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 PRACTICE script (*.cmm) via the **DO** command.

The command string can contain format specifiers that are allowed by the host's compiler for printf commands (e.g. "%d" or "%s"). The parameter list must contain appropriate arguments to fulfil the format specifiers requests.

NOTE:	When executing a script via the "**DO**" command, the function will return immediately, not waiting for the end of the script. You may use **T32_GetPracticeState()** to actively wait for the script ending.

Negative error values indicate a communication problem between the debugger and the API.
An positive error value indicates that the command was not accepted.
Errors caused while executing the command are not reported, to retrieve further error information, please use the call **T32_GetMessage()** and check the message type.

Example:

```
int     error;
int     address = 0x1234;
char*   ascii = "text";
error = T32_Cmd_f ("Data.Set 0x%x \"%s\" ", address, ascii);
```

This function is deprecated.

Prototype:

```
int T32_CmdWin ( uint32_t WindowHandle, const char *command );
```

Prototype:

```
int T32_ExecuteCommand ( const char *pCommand,
                         char* pBuffer,
                         uint32_t BufferSize );
```

Parameters:

```
pCommand           ; TRACE32 command to execute

pBuffer            ; pointer to error buffer

BufferSize         ; size of buffer
```

Returns:

0 for ok, otherwise Error value

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

NOTE:	When executing a script via the "**DO**" command, the function will return immediately, not waiting for the end of the script. You may use **T32_GetPracticeState()** to actively wait for the script ending.

Negative error values indicate a communication problem between the debugger and the API.
An positive error value indicates that the command was not accepted.
In contrast to **T32_Cmd()** errors are reporting using the buffer provided by the caller.

Example:

```
char buffer[4096];
const char* cmd = "Data.Set %Long D:0x1234 0x033FFC00";
error = T32_ExecuteCommand (cmd, buffer, 4096);
if (error)
    printf ("Error: %s", 4096, buffer);
```

Prototype:

```
int T32_ExecuteFunction ( const char *pExpression,
                          char       *pBuffer,
                          uint32_t    BufferSize
                          uint32_t   *pResultType);
```

Parameters:

```
pExpression        ; TRACE32 function to execute

pBuffer            ; pointer to error/result buffer

BufferSize         ; size of buffer

pResultType        ; result type
```

Returns:

0 for ok, otherwise Error value

With this function a TRACE32 expression is passed to TRACE32 for execution. Any valid TRACE32 expression is allowed, see **EVAL**.

Negative error values indicate a communication problem between the debugger and the API.
An positive error value indicates that the expression was not accepted and the buffer holds the error string.

If no error occurred, the buffer holds the result in a string representation.

The meaning of the result type values is described in the documentation of function **EVAL.TYPE()**.

Example:

```
const char* expr = "Data.Long(D:0x1234)";
char buffer[4096];
uint32_t restype;
error = T32_ExecuteFunction (expr, buffer, 4096, &restype);
if (error)
    printf ("Error: %s",  4096, buffer);
else
    printf ("Result: %s", 4096, buffer);
```

Prototype:

```
int T32_ExecuteFunction_Double ( const char *pExpression,
                                 char        *pBuffer,
                                 uint32_t     BufferSize
                                 double      *pResult);
```

Parameters:

```
pExpression        ; TRACE32 function to execute

pBuffer            ; pointer to error buffer

BufferSize         ; size of buffer

pResult            ; pointer to result
```

Returns:

0 for ok, otherwise Error value

Similar to **T32_ExecuteFunction()** but the result is returned as a double-precision floating-point value.

Examples:

```
char buffer[4096];
double result;
error = T32_ExecuteFunction_Double ("FPU(F0)", buffer, 4096, result);
if (error)
    printf ("Error: %s", 4096, buffer);
else
    printf ("Result: %f", result);
```

```
const char* expr = "Data.Byte(SD:0x300)*1.0+Data.Byte(SD:0x300)*0.01"
char buffer[4096];
double result;
error = T32_ExecuteFunction_Double (expr, buffer, 4096, result);
if (error)
    printf ("Error: %s", 4096, buffer);
else
    printf ("Result: %f Volt", result);
```

Prototype:

```
int T32_ExecuteFunction_UInt64 ( const char *pExpression,
                                 char       *pBuffer,
                                 uint32_t    BufferSize
                                 uint64_t   *pResult);
```

Parameters:

```
pExpression        ; TRACE32 function to execute

pBuffer            ; pointer to error buffer

BufferSize         ; size of buffer

pResult            ; pointer to result
```

Returns:

0 for ok, otherwise Error value

Similar to **T32_ExecuteFunction()** but the result is returned as a uint64_t and not a string.

Example:

```
const char* expr = "Register(R5)";
char buffer[4096];
uint64_t result;
error = T32_ExecuteFunction_UInt64 (expr, buffer, 4096, result);
if (error)
    printf ("Error: %s", 4096, buffer);
else
    printf("Result: %i", result);
```

Prototype:

```
int T32_Printf ( const char *string, ... );
```

Parameters:

```
string              ; text to print to TRACE32 AREA window,
                    ; with format specifiers
```

Returns:

0 for ok, otherwise Error value

This function prints the given string onto the message line of TRACE32 and into the active AREA window.

The string can contain format specifiers that are allowed by the host's compiler for printf commands (e.g. "%d" or "%s"). The parameter list must contain appropriate arguments to fulfil the format specifiers requests.

Example:

```
int    error;
int    result = 0;
error = T32_Printf ("Last result was %d.\n", result);
```

Prototype:

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

Parameters:

none

Returns:

0 or 1 for ok, otherwise Error value

If a PRACTICE script is currently running, it is stopped. For stopping the target program use **T32_Break()**.

Example:

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

Prototype:

```
int T32_EvalGet ( uint32_t *pEvalResult );
```

Parameters:

```
pEvalResult        ; pointer to variable receiving the evaluation result
```

Returns:

0 for ok, otherwise Error value

Some PRACTICE commands (e.g. **Eval**) 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:

```
int       error;
uint32_t  result;
T32_Cmd ("EVAL VERSION.BUILD()");
error = T32_EvalGet (&result);
if (error == T32_OK)
    printf ("Attached to TRACE32 build version %d.\n", result);
else
    printf ("Error getting evaluation result: %d!\n", error);
```

NOTE: This function is only available when attached to a device (see **T32_Attach**).

Prototype:

```
int T32_EvalGetString ( char* EvalString );
```

Parameters:

```
EvalString          ; pointer to character array receiving the evaluation
                    result. The array must be at least 4096 bytes.
```

Returns:

0 for ok, otherwise Error value

Some PRACTICE commands (e.g. **Eval**) 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_EvalGetString reads the last evaluation result that returned a string.

Example:

```
int   error;
char  evalString[4096];
T32_Cmd ("EVAL \"hello\"+conv.char(0x20)+\"world\"");
error = T32_EvalGetString (evalString);
if (error == T32_OK)
    printf ("EVAL returned string \"%s\".\n", evalString);
else
    printf ("Error getting evaluation result: %d!\n", error);
```

NOTE: This function is only available when attached to a device (see **T32_Attach**).

Prototype:

```
int T32_GetMessage ( char AreaMessage[256], uint16_t *pMessageType );
```

Parameters:

```
pMessageType        OUT      pointer to a variable getting the status
                             information (see below).

AreaMessage         OUT      pointer to array of at least 256 characters.
                             Contents set by API, but is only valid, if
                             *pMessageType != 0
```

Returns:

0 for OK, otherwise Error value

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

NOTE:	This call truncates the message length to 255 characters. If you need longer messages, use **T32_GetMessageString** instead.

NOTE:	The message length depends on the actual AREA size.

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

Type	**Meaning**
0	OK : the call was successful. **The returned message has to be ignored**
1	General Information
2	Error
8	Status Information
16	Error Information

32	Temporary Display
64	Temporary Information

Example:

```
char      message[256];
uint16_t  type;

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

Prototype:

```
int T32_GetMessageString ( char* AreaMessage,
                           uint16_t ArraySize,
                           uint16_t *pMessageType,
                           uint16_t *pMessageLen );
```

Parameters:

```
AreaMessage      OUT     pointer to an character array getting the
                         message. Only valid, if *pMessageType != 0

ArraySize        IN      size of the AreaMessage character array

pMessageType     OUT     pointer to a variable getting the status
                         information (see below).

pMessageLen      OUT     pointer to a variable getting the original
                         length of the message.
```

Returns:

0 for OK, otherwise Error value

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

NOTE:	The message length depends on the actual AREA size.

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

Type	Meaning
0	OK : the call was successful. **The returned message has to be ignored**
1	General Information
2	Error
8	Status Information
16	Error Information

32	Temporary Display
64	Temporary Information

Example:

```
char      message[256];
uint16_t  type;
uint16_t  length;

error = T32_Cmd ("print");                    /* delete previous outputs */
error = T32_Cmd ("print clock.date()");
error = T32_GetMessageString (message, 256, &type, &length);
printf ("Message: %s\nType: %d\nLength: %d\n",
    message, type, length);
```

Prototype:

```
int T32_Terminate ( int retval );
```

Parameters:

```
retval              ; TRACE32 instance returns this value
                    ; to the command shell when terminating
```

Returns:

0 for OK, otherwise Error value

Use this command to terminate the connected TRACE32 instance.

T32_GetPracticeState — Check if a PRACTICE script is running

Prototype:

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

Parameters:

```
pstate              ; output parameter, set by API
                    ; 0: not running
                    ; 1: running
                    ; 2: dialog window open
```

Returns:

0 for OK, otherwise Error value

Returns the run-state of PRACTICE. Use this command to poll for the end of a PRACTICE script started via **T32_Cmd()**.

T32_SetMode — Set Data.List display mode

Prototype:

```
int T32_SetMode ( int mode );
```

Parameters:

```
mode                ; display mode for Data.List windows:
                    ; 0=ASM, 1=HLL, 2=MIX
```

Returns:

0 for OK, otherwise Error value

Sets the display mode for **List** windows.

Prototype:

```
int T32_GetWindowContent ( const char* command,
                           char      * buffer,
                           uint32_t    requested,
                           uint32_t    offset,
                           uint32_t    print_code );
```

Parameters:

```
command           ; PRACTICE command to open the TRACE32 window

buffer            ; output

requested         ; number of bytes to read

offset            ; offset to start read from

print_code        ; print format
```

Returns:

-1 in case of error, otherwise the number of bytes received.

Get the content of a TRACE32 window in the selected print format specified by the `print_code` parameter. Possible print code values are:

```
T32_PRINTCODE_ASCII
T32_PRINTCODE_ASCIIE
T32_PRINTCODE_ASCIIP
T32_PRINTCODE_CSV
T32_PRINTCODE_XML
```

Example:

```
char buf[1024];
uint32_t offset = 0, len;
uint32_t code = T32_PRINTCODE_ASCII;
const char * cmd = "List"; // get the content of the List window
do {
  len = T32_GetWindowContent(cmd, buf, sizeof(buffer), offset, code);
  if (len < 0)
    break;
  printf("%s", buf);
  offset += len;
} while (len > 0);
```

Prototype:

```
int T32_GetApiRevision ( uint32_t *pRevNum );
```

Parameters:

```
pRevNum             ; pointer to variable receiving the revision number
```

Returns:

0 for OK, otherwise Error value

Returns the revision number of the Remote API (source files or library) at the application side. It does *not* report the revision number of the TRACE32 software.

Prototype:

```
int T32_GetSocketHandle ( int *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.

```
int t32soc;

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

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 each single API call, 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:	Each debugger must be assigned a unique PORT address in its configuration file (e.g. config.t32).

T32_GetChannelSize — Get size of channel structure (deprecated)

This function is deprecated. Please use one of these functions instead:

- **T32_RequestChannelNetAssist**
- **T32_RequestChannelNetTcp**

Prototype:

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

Parameters:

none

Returns:

size_of channel structure

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

Example:

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

This function is deprecated. Please use one of these functions instead:

- T32_RequestChannelNetAssist
- T32_RequestChannelNetTcp

Prototype:

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

Parameters:

pointer to channel structure 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:

```
void* channel = malloc(T32_GetChannelSize());
T32_GetChannelDefaults(channel);
T32_SetChannel(channel);
...
```

Prototype:

```
int T32_RequestChannelNetAssist(void **channel);
```

Parameters:

Pointer to channel structure receiving the defaults.

Returns:

0 for ok, otherwise Error value

Creates a NETASSIST (UDP) channel. Can be configured later with **T32_Config()**. Requires `RCL=NETASSIST` line in `config.t32`. For more details, please refer to **"Preparing TRACE32 Software"**, page 17.

Example:

```
void *channel;
T32_RequestChannelNetAssist(&channel);
T32_SetChannel(channel);
...
```

Full example see at **T32_SetChannel()**.

Prototype:

```
int T32_RequestChannelNetTcp(void **channel);
```

Parameters:

Pointer to channel structure receiving the defaults.

Returns:

0 for ok, otherwise Error value

Creates a NETTCP (TCP) channel. Can be configured later with **T32_Config()**. Requires `RCL=NETTCP` line in `config.t32`. For more details, please refer to **“Preparing TRACE32 Software”**, page 17.

Example:

```
void *channel;
T32_RequestChannelNetTcp(&channel);
T32_SetChannel(channel);
...
```

Full example see at **T32_SetChannel()**.

Prototype:

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

Parameters:

Pointer to activating channel.

Returns:

none

This function sets the active channel to be used for further T32_* calls.

Example:

```
void *channel_1;
void *channel_2;
T32_RequestChannelNetAssist(&channel_1);
T32_RequestChannelNetTcp(&channel_2);
// switch to channel 1
T32_SetChannel(channel_1);
T32_Config("PORT=", "20000");
T32_Init();
T32_Attach(T32_DEV_ICE);
// switch to channel 2
T32_SetChannel(channel_2);
T32_Config("PORT=", "20002");
T32_Init();
T32_Attach(T32_DEV_ICE);
…
```

Prototype:

```
int T32_ReleaseChannel(void **channel);
```

Parameters:

Pointer to created channel.

Returns:

0 for ok, otherwise Error value

Example:

```
void *channel
T32_ReleaseChannel ( &channel );
```

Full example see at **T32_SetChannel()**.

API Functions

This chapter describes all available API functions. See **T32_Attach()** for how to specify a device.

T32_GetState — Get State of Debugger

Prototype:

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

Parameters:

```
pstate              ; pointer to variable receiving the debugger state
```

Returns:

0 for ok, otherwise Error value. Note that `pstate` is not modified if an error has occurred.

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

0	Debug system is down
1	This value is returned in one situation: • (Intel x86/x64 debugger only) Target is in bootstall
2	Target execution is stopped (**Break**)
3	Target execution is running (**Go**)

Example:

```
int state = -1;

error = T32_GetState ( &state );
/* no error handling, but state preset to detect problems */
printf ("System is ");
switch (state)
{
     case 0:  printf ("down.\n");    break;
     case 1:  printf ("halted.\n");  break;
     case 2:  printf ("stopped.\n"); break;
     case 3:  printf ("running.\n"); break;
      default: printf ("Error!\n");
}
```

Prototype:

```
int T32_GetCpuInfo ( char      ** pstring,
                     uint16_t   * pfpu,
                     uint16_t   * pendian,
                     uint16_t   * ptype );
```

Parameters:

```
pstring              ; pointer to variable receiving a pointer
                     ; to a string describing the CPU

pfpu                 ; pointer to variable receiving the FPU type

pendian              ; pointer to variable receiving the byte order

ptype                ; 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 an 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 = "";
uint16_t  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_ResetCPU ( void );
```

Parameters:

none

Returns:

0 for ok, otherwise Error value

Tries to reset the target CPU. 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 ();
```

This function is deprecated, please use T32_ReadMemoryObj instead.

Prototype:

```
int T32_ReadMemory ( uint32_t  byteAddress,
                     int       access,
                     uint8_t  *buffer,
                     int       byteSize );
```

Parameters:

```
byteAddress          ; target memory address to start read

access               ; memory access specifier

buffer               ; output

byteSize             ; 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…4	Memory class, see "**Conventions for Target Memory Access**"
Bit 6	Set for emulation memory access (E:, dual port access)

For a more advanced version of the function to read memory, including 64bit addresses and several access options, see **T32_ReadMemoryObj**.

NOTE:	See the section "**Conventions for Target Memory Access**" for important conventions regarding the byteAddress, byteSize, and access parameters.

NOTE:	The access parameter is *only* used, if the access class is *not* set with **T32_SetMemoryAccessClass** (see there). Otherwise the access parameter is ignored and the access class set with T32_SetMemoryAccessClass is used.

Example:

```
// Read 16 bytes from address D:0x100

// 1) byte-addressed machine (e.g. x86, MicroBlaze, PPC)
uint8_t buffer[16];
error = T32_ReadMemory (0x100, 0x0 /*D:*/, buffer, 16 /*bytes*/);

// 2) word-addressed machine using 16bit words (e.g. C2000)
uint8_t buffer[16];
error = T32_ReadMemory (0x100 * 2 /*16bit*/, 0x0, buffer, 16 /*bytes*/);
```

Prototype:

```
int T32_ReadMemoryObj (T32_BufferHandle bufferHandle,
    const T32_AddressHandle addressHandle, const T32_Length length);
```

Parameters:

```
bufferHandle          ; handle to buffer object where the read data
                        will be stored

addressHandle         ; handle to address object containing the address
                        and access method where to read from

length                ; number of bytes to read
```

Returns:

0 for ok, otherwise Error value

Reads data from target memory.

A “buffer handle” must be declared and requested by the application as shown in the example and description below.

An “address handle” must be declared, requested and set by the application as shown in the example and description below.

Example to read a buffer from a 32bit address:

```
uint32_t myAddress = 0x12345678L;
uint8_t  LocalBuffer[32];

T32_BufferHandle  myBufferHandle;
T32_AddressHandle myAddressHandle32;

T32_RequestBufferObj(&myBufferHandle, 0);
T32_RequestAddressObjA32(&myAddressHandle32, myAddress);

T32_ReadMemoryObj (myBufferHandle, myAddressHandle32, 32);

T32_CopyDataFromBufferObj (LocalBuffer, 32, myBufferHandle);
T32_ReleaseBufferObj (&myBufferHandle); // release single object

/* read data is now stored in "LocalBuffer" */;
```

Address Object handling: For a description of the address object, see chapter "**Address Object**".

Buffer Object handling: For a description of the buffer object, see chapter "**Buffer Object**".

T32_WriteMemory — Write to Target Memory (deprecated)

This function is deprecated, please use T32_WriteMemoryObj instead.

Prototype:

```
int T32_WriteMemory ( uint32_t  byteAddress,
                      int       access,
                      uint8_t  *buffer,
                      int       byteSize );
```

Parameters:

```
byteAddress          ; target memory address to start write

access               ; memory access specifier

buffer               ; output

byteSize             ; 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…4	Memory class, see "**Conventions for Target Memory Access**"
Bit 6	Set for emulation memory access (dual port access)
Bit 7	Set to enable verify after write

For a more advanced version of the function to write memory, including 64bit addresses and several access options, see **T32_WriteMemoryObj**.

NOTE:	See the section "**Conventions for Target Memory Access**" for important conventions regarding the byteAddress, byteSize, and access parameters.

NOTE:	The access parameter is *only* used, if the access class is *not* set with **T32_SetMemoryAccessClass** (see there). Otherwise the access parameter is ignored and the access class set with T32_SetMemoryAccessClass is used.

Example:

```
uint8_t buffer[16];

error = T32_WriteMemory ( 0x100, 0xc0, buffer, 16 );
```

This function is deprecated. Please use one of these functions instead:

- **T32_WriteMemory**
- **T32_WriteMemoryObj**

Prototype:

```
int T32_WriteMemoryPipe ( uint32_t  byteAddress,
                          int       access,
                          uint8_t  *buffer,
                          int       byteSize );
```

Parameters:

```
byteAddress          ; target memory address to start write

access               ; memory access specifier

buffer               ; output

byteSize             ; number of bytes to write
```

NOTE:	See the section “Conventions for Target Memory Access” for important conventions regarding the byteAddress, byteSize, and access parameters.

Returns:

0 for ok, otherwise Error value

Writes data to target memory with pipelining. Pipelinig means that the memory write operation 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 byteSize=0. The size of the data block is not limited.

NOTE:	• No other API calls are allowed between consecutive calls to **T32_WriteMemoryPipe()**. • A sequence of **T32_WriteMemoryPipe()** calls must end with a call with byteSize = 0.

The access flags define the memory access class and access method (see **T32_WriteMemory**).

Example:

```
uint8_t buffer[1024];

error = T32_WriteMemoryPipe ( 0x400, 0xc0, buffer, 1024 );
```

Prototype:

```
int T32_WriteMemoryObj (T32_BufferHandle bufferHandle,
    const T32_AddressHandle addressHandle, const T32_Length length);
```

Parameters:

```
bufferHandle          ; handle to buffer object containing the data
                        to write

addressHandle         ; handle to address object containing the address
                        and access method where to write to

length                ; number of bytes to write
```

Returns:

0 for ok, otherwise Error value

Writes data to target memory.

A “buffer handle” must be declared, requested and set by the application as shown in the example and description below.

An “address handle” must be declared, requested and set by the application as shown in the example and description below.

Example to write a buffer to a 64bit address:

```
uint8_t  LocalBuffer[32];

T32_BufferHandle  myBufferHandle;
T32_AddressHandle myAddressHandle64;

T32_RequestBufferObj(&myBufferHandle, 0);
T32_RequestAddressObjA64(&myAddressHandle64, 0x2000200020002000LL);

/* copy data to write into the buffer */
T32_CopyDataToBufferObj (myBufferHandle, 8, "abcdefgh");

T32_WriteMemoryObj (myBufferHandle, myAddressHandle64, 8);

T32_ReleaseAllObjects ();       // release all T32 objects
```

Buffer Object handling: For a description of the buffer object, see chapter “**Buffer Object**”.

Address Object handling: For a description of the address object, see chapter “**Address Object**”.

T32_TransferMemoryBundleObj Read/Write Target Memory Bundles

Prototype:

```
int T32_TransferMemoryBundleObj (T32_MemoryBundleHandle bundles);
```

Parameters:

```
bundles                ; handle to bundle object containing the list of
                         buffers to be read or written
```

Returns:

0 for ok, otherwise Error value

Reads/writes a list of target memory buffers back-to-back. The purpose of this function is to allow for fast transfer of many memory read and write operations.

A “buffer handle” must be declared, requested and set by the application as shown in the example below.

Example showing the transfer of a bundle with one read buffer and one write buffer:

```
T32_Size bundleSize, i;
T32_AddressHandle addressHandle;
T32_BufferSynchStatus syncStatus;
T32_MemoryBundleHandle bundleHandle;

T32_RequestMemoryBundleObj(&bundleHandle, 0);

T32_RequestAddressObjA32(&addressHandle, 0x10000);
T32_AddToBundleObjAddrLength(bundleHandle, addressHandle, 8);
T32_ReleaseAddressObj(&addressHandle);

T32_RequestAddressObjA32(&addressHandle, 0x20000);
T32_AddToBundleObjAddrLengthByteArray(bundleHandle, addressHandle,
                                      8, "abcdefgh");
T32_ReleaseAddressObj(&addressHandle);

T32_TransferMemoryBundleObj(bundleHandle);

T32_GetBundleObjSize(bundleHandle, &bundleSize);
for (i = 0; i < bundleSize; i++) {
  T32_GetBundleObjSyncStatusByIndex(bundleHandle, &syncStatus, i);
  if (syncStatus == T32_BUFFER_READ) {
    uint8_t buf[8];
    T32_CopyDataFromBundleObjByIndex(buf, 8, bundleHandle, i);
  }
  else if (syncStatus != T32_BUFFER_WRITTEN)
    printf("ERROR: Bundle buffer read/write error");
}

T32_ReleaseMemoryBundleObj(&bundleHandle);
```

Bundle Object handling: For a description of the bundle object, see chapter “**Bundle Object**”.

Address Object handling: For a description of the address object, see chapter “**Address Object**”.

Prototype:

```
int T32_SetMemoryAccessClass ( const char* access );
```

Parameters:

```
access                ; memory access class specifier as string
```

Returns:

0 for ok, otherwise Error value.

Sets the memory access class for all further memory accesses, e.g. with **T32_ReadMemory** or **T32_WriteMemory**. The “access” parameter of those calls will then be ignored.

The memory access class must be given in a null-terminated string containing the access class specifier as listed in the Processor Architecture Manuals **without the colon**.

Note: the access class is *not* validated. Wrong access classes will be accepted, but will give errors in the subsequent memory accesses.

An empty string or NULL as parameter will disable this access class and re-enables the “access” parameter of the memory read/write calls.

Example:

```
; read CP15 register of an Arm architecture:
error = T32_SetMemoryAccessClass ("C15");
error = T32_ReadMemory (0x4, 0, buffer, 4);

; read supervisor data
error = T32_SetMemoryAccessClass ("SD");
error = T32_ReadMemory (0x4, 0, buffer, 4);

; switch back and use access parameter of T32_ReadMemory
error = T32_SetMemoryAccessClass ("");
error = T32_ReadMemory (0x4, 0x40, buffer, 4);
```

This function is deprecated, please use T32_ReadRegisterObj instead.

Prototype:

```
int T32_ReadRegister ( uint32_t  mask1,
                       uint32_t  mask2,
                       uint32_t *buffer );
```

Parameters:

```
mask1, mask2          ; register addressing mask
buffer                ; pointer to host buffer receiving 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 debugger, 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. Contact Lauterbach to get a register map of a specific CPU.

For a more advanced version of the function to read registers, including 64bit accesses and core specification, see **T32_ReadRegisterObj**.

Example:

```
uint32_t buffer[64];

                      /* define register array */

error = T32_ReadRegister ( 0x3ff, 0x0, buffer );

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

Prototype:

```
int T32_ReadRegisterByName ( const char  *regname,
                             uint32_t    *value,
                             uint32_t    *hvalue )
```

Parameters:

```
regname          ; pointer to register name
value            ; pointer to variable receiving the value
hvalue           ; pointer to variable receiving the upper 32bit
```

Returns:

0 for ok,
>0 for access error (e.g. wrong register name)
<0 for communication error.

This function provides the value for a specified register.
If the size of the register is smaller or equal to 32bit, the value is stored in “value”.
If the size of the register is 64bit, the upper 32bit are stored in “hvalue”.

For a more advanced version of the function to read registers, including 64bit accesses and core specification, see **T32_ReadRegisterObj**.

Example:

```
uint32_t  value, hvalue;
int       state

state = T32_ReadRegisterByName ("R1", &value, &hvalue);

printf ("R1 = 0x%x, state = %d, %s\n",
        value, state, state==0 ? "OK" : "NOK");
```

Prototype:

```
int T32_ReadRegisterObj (T32_RegisterHandle registerHandle);
```

Parameters:

```
registerHandle       ; handle to register object containing the register
                       where to read from with options
```

Returns:

0 for ok, otherwise Error value

Reads one register from the target CPU.

A “register handle” must be declared, requested and set by the application as shown in the example and description below.

Example to read a 32bit register with a given name:

```
uint32_t regValue;

T32_RegisterHandle myRegisterHandle32;

T32_RequestRegisterObjR32Name(&myRegisterHandle32, "PC");

T32_ReadRegisterObj (myRegisterHandle32);

T32_GetRegisterObjValue32(myRegisterHandle32, &regValue);

T32_ReleaseRegisterObj (&myRegisterHandle); // release single object

/* read register value is now stored in "regValue" */;
```

Register Object handling: For a description of the register object, see chapter “**Register Object**”.

Prototype:

```
int T32_ReadRegisterSetObj (T32_RegisterSetHandle registerSetHandle);
```

Parameters:

```
registerSetHandle    ; handle to register set object containing the
                       registers to read
```

Returns:

0 for ok, otherwise Error value

Reads a predefined register set from the target CPU.

A “register set handle” must be declared, requested and set by the application as shown in the example and description below.

Example to read a 32bit register set:

```
const char *regNames[5] = {"R0", "R1", "R2", "R3", "PC"};
uint32_t    regValues[5];
int         i;

T32_RegisterSetHandle regSetHandle;

T32_RequestRegisterSetObjR32 (&regSetHandle, 5);
T32_SetRegisterSetObjNames (regSetHandle, regNames, 5);

T32_ReadRegisterSetObj (regSetHandle);

T32_GetRegisterSetObjValues32(regSetHandle, regValues, 5);
T32_ReleaseRegisterSetObj (&regSetHandle);

/* read register values are now stored in "regValues" array*/;
for (i = 0; i < 5; i++)
    printf ("Register %s = 0x%08x\n", regNames[i], regValues[i]);
```

RegisterSet Object handling: For a description of the register set object, see chapter “**RegisterSet Object**”.

This function is deprecated, please use T32_WriteRegisterObj instead.

Prototype:

```
int T32_WriteRegister ( uint32_t   mask1,
                        uint32_t   mask2,
                        uint32_t  *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. Contact Lauterbach to get a register map of a specific CPU.

For a more advanced version of the function to write registers, including 64bit accesses and core specification, see **T32_WriteRegisterObj**.

Example:

```
uint32_t buffer[64];

                      /* define register array */

buffer[1] = buffer [3] = 0x30f0;
error = T32_WriteRegister ( 0x0c, 0x0, buffer );

                      /* write register 1 and 3 */
```

Prototype:

```
int T32_WriteRegisterByName ( const char  *regname,
                             uint32_t    value,
                             uint32_t    hvalue )
```

Parameters:

```
regname          ; pointer to register name
value            ; value to write to the register
hvalue           ; upper 32bit in case of 64bit register
```

Returns:

0 for ok,
>0 for access error (e.g. wrong register name)
<0 for communication error.

This function sets the value of the specified register.
If the size of the register is smaller or equal to 32bit, it is set to the parameter "value".
If the size of the register is 64bit, specify the upper 32bit in "hvalue".

For a more advanced version of the function to write registers, including 64bit accesses and core specification, see **T32_WriteRegisterObj**.

Example:

```
uint32_t  value, hvalue;
int       state

value  = 0x1234;
hvalue = 0;
state  = T32_WriteRegisterByName ("R1", value, hvalue);

printf ("R1 is set to 0x%x, state = %d, %s\n",
        value, state, state==0 ? "OK" : "NOK");
```

Prototype:

```
int T32_WritRegisterObj (T32_RegisterHandle registerHandle);
```

Parameters:

```
registerHandle        ; handle to register object containing the register
                        where to write to with options
```

Returns:

0 for ok, otherwise Error value

Writes to one register of the target CPU.

A “register handle” must be declared, requested and set by the application as shown in the example and description below.

Example to write a 64bit value to a 64bit register:

```
/* write 0x2000200020002000 to the program counter */

T32_RegisterHandle myRegisterHandle64;

T32_RequestRegisterObjR64(&myRegisterHandle64);
T32_SetRegisterObjName(myRegisterHandle64, "PC");
T32_SetRegisterObjValue64(myRegisterHandle64, 0x2000200020002000LL);

T32_WriteRegisterObj (myRegisterHandle64);

T32_ReleaseAllObjects ();       // release all T32 objects
```

Register Object handling: For a description of the register object, see chapter “**Register Object**”.

Prototype:

```
int T32_WriteRegisterSetObj (T32_RegisterSetHandle registerSetHandle);
```

Parameters:

```
registerSetHandle    ; handle to register set object containing the
                       registers to write
```

Returns:

0 for ok, otherwise Error value

Writes to a predefined register set from the target CPU.

A “register set handle” must be declared, requested and set by the application as shown in the example and description below.

Example to write a 32bit register set:

```
const char *regNames[5]  = {"R0", "R1", "R2", "R3", "PC"};
uint32_t    regValues[5] = {0x10, 0x11, 0x12, 0x13, 0x20};

T32_RegisterSetHandle regSetHandle;

T32_RequestRegisterSetObjR32 (&regSetHandle, 5);
T32_SetRegisterSetObjNames (regSetHandle, regNames, 5);
T32_GSetRegisterSetObjValues32 (regSetHandle, regValues, 5);

T32_WriteRegisterSetObj (regSetHandle);

T32_ReleaseRegisterSetObj (&regSetHandle);
```

RegisterSet Object handling: For a description of the register set object, see chapter “**RegisterSet Object**”.

Prototype:

```
int T32_ReadPP ( uint32_t *pp );
```

Parameters:

```
pp                   ; pointer to variable receiving 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 (see `T32_GetState`). The program pointer is a logical pointer to the address of the next executed assembler line. Unlike `T32_ReadRegister`, this function is completely processor independent.

Example:

```
uint32_t pp;
error = T32_ReadPP ( &pp );
printf ("Current Program Pointer: %x\n", pp);
```

This function is deprecated, please use T32_ReadBreakpointObj instead.

Prototype:

```
int T32_ReadBreakpoint ( uint32_t   address,
                         int        access,
                         uint16_t  *buffer,
                         int        size );
```

Parameters:

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

Returns:

0 for ok, otherwise Error value

Read breakpoint information from debugger.

The `access` variable defines the memory class and access method. See **T32_ReadMemory** for definitions and other methods of specifying the access class.

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	marker d

Bit 9	marker e
Bit 10	implemented as ONCHIP
Bit 11	implemented as SOFT
Bit 12	implemented as HARD

For a more advanced version of the function to read breakpoints, including 64bit accesses and other options, see **T32_ReadBreakpointObj**.

Example:

```
uint16_t buffer[16];
error = T32_ReadBreakpoint ( 0x100, 0x40, buffer, 16 );
```

This function is deprecated, please use T32_WriteBreakpointObj instead.

Prototype:

```
int T32_WriteBreakpoint ( uint32_t  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** for definitions and other methods of specifying the access class.

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

For a more advanced version of the function to write breakpoints, including 64bit accesses and other options, see **T32_WriteBreakpointObj**.

Example:

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

Prototype:

```
int T32_GetBreakpointList ( int              *numbps,
                            T32_Breakpoint   *bps,
                            int               max );
```

Parameters:

```
numbps              ; pointer to variable receiving number of breakpoints
bps                 ; structure array receiving the breakpoint list
max                 ; maximum number of array elements
```

Returns:

0 for ok, otherwise Error value

Read the breakpoint list of the debugger. The T32_Breakpoint structure contains the address, status and type of the breakpoint:

address	start address of the breakpoint
enabled	1 if breakpoint is enabled, 0 if disabled
type	breakpoint type
auxtype	auxilary breakpoints (e.g. temporary)

For a more advanced version of the function to get a breakpoint list, including 64bit accesses and other options, see **T32_ReadBreakpointObjByIndex**.

Example:

```
int i, num;
T32_Breakpoint bp[100];
T32_GetBreakpointList (&num, bp, 100);
printf ("number of breakpoints: %d\n", num);
for (i = 0; (i < num) && (i < 100); i++)
    printf ("address = 0x%x, enable = %d, type = 0x%x\n",
            bp[i].address, bp[i].enabled, bp[i].type);
```

Prototype:

```
int T32_WriteBreakpointObj (T32_BreakpointHandle bpHandle, int set);
```

Parameters:

```
bpHandle            ; handle to breakpoint object

set                 1: set breakpoint, 0: delete breakpoint
```

Returns:

0 for ok, otherwise Error value

This function sets or deletes a breakpoint in TRACE32.

A “breakpoint handle” must be declared, requested and set by the application as shown in the example and description below.

Example to write a software breakpoint onto a 32bit address:

```
T32_AddressHandle    myAddressHandle = NULL;
T32_BreakpointHandle myBpHandle      = NULL;

T32_RequestAddressObjA32 (&myAddr, 0x12345678L);
T32_RequestBreakpointObjAddr (&myBpHandle, myAddr);

T32_SetBreakpointObjImpl (myBpHandle, T32_BP_IMPL_SOFT)

T32_WriteBreakpointObj(myBpHandle, 1);

T32_ReleaseAllObjects ();   // release all T32 objects
```

Address Object handling: For a description of the address object, see chapter “**Address Object**”.

Breakpoint Object handling: For a description of the breakpoint object, see chapter “**Breakpoint Object**”.

Prototype:

```
int T32ReadBreakpointObj (T32_BreakpointHandle bpHandle);
```

Parameters:

```
bpHandle             ; handle to breakpoint object
```

Returns:

0 for ok, otherwise Error value

This function reads the characteristics of a breakpoint in TRACE32. The breakpoint to read is specified by the address given in the breakpoint handle.

A “breakpoint handle” must be declared, requested and set by the application as shown in the example and description below.

Example to read the breakpoint characteristics of a 32bit address:

```
uint32_t   address;
T32_AddressHandle    myAddrHandle = NULL;
T32_BreakpointHandle myBpHandle   = NULL;

T32_RequestAddressObjA32 (&myAddrHandle, 0x12345678L);
T32_RequestBreakpointObjAddr (&myBpHandle, myAddrHandle);

T32_ReadBreakpointObj (myBpHandle);

T32_GetBreakpointObjAddress (myBpHandle, &myAddrHandle);
T32_GetAddressObjAddr32 (myAddrHandle, &address);
T32_GetBreakpointObjType (myBpHandle, &type);

T32_ReleaseAllObjects ();   // release all T32 objects

/* breakpoint address is now in "addr" and type in "type" */
```

Address Object handling: For a description of the address object, see “**Address Object**”.

Breakpoint Object handling: For a description of the breakpoint object, see “**Breakpoint Object**”.

Prototype:

```
int T32ReadBreakpointObjByIndex (T32_BreakpointHandle bpHandle
    uint32_t index);
```

Parameters:

```
bpHandle              ; handle to breakpoint object

index                 index of breakpoint in breakpoint list
```

Returns:

0 for ok, otherwise Error value

This function reads the characteristics of a breakpoint in TRACE32. The breakpoint to read is specified by the index value. The index starts at 0 and ends with the number of breakpoints in TRACE32 minus one. The number of current breakpoints can be retrieved with **T32_QueryBreakpointObjCount**.

A "breakpoint handle" must be declared, requested and set by the application as shown in the example and description below.

Example to read the breakpoint characteristics of the second breakpoint in the breakpoint list:

```
uint32_t   address;
T32_AddressHandle    myAddrHandle = NULL;
T32_BreakpointHandle myBpHandle   = NULL;

T32_RequestBreakpointObj (&myBpHandle);

T32_ReadBreakpointObjByIndex (myBpHandle, 1);

T32_GetBreakpointObjAddress (myBpHandle, &myAddrHandle);
T32_GetAddressObjAddr32 (myAddrHandle, &address);
T32_GetBreakpointObjType (myBpHandle, &type);

T32_ReleaseAllObjects ();   // release all T32 objects

/* breakpoint address is now in "addr" and type in "type" */
```

Address Object handling: For a description of the address object, see "**Address Object**".

Breakpoint Object handling: For a description of the breakpoint object, see "**Breakpoint Object**".

Prototype:

```
int T32_QueryBreakpointObjCount (uint32_t* pCount);
```

Parameters:

```
pCount                variable to receive the number of breakpoints
```

Returns:

0 for ok, otherwise Error value

This function retrieves the number of breakpoints set in TRACE32. Use **T32_ReadBreakpointObjByIndex** to iterate and read the breakpoints then.

Prototype:

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

Parameters:

none

Returns:

0 for ok, otherwise Error value

Executes one single step.

Example:

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

Prototype:

```
int T32_StepMode ( int mode );
```

Parameters:

```
mode                    ; stepping mode
```

Returns:

0 for ok, otherwise Error value

Executes one step. 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 target (or start real-time 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

Break into target (or stop the real-time emulation asynchronously).

Example:

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

Prototype:

```
int T32_GetTriggerMessage ( char message[256] );
```

Parameters:

```
message     ; pointer to an array of 256 characters receiving 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 an user allocated character array of at least 256 elements.

Example:

```
char message[256];

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

Prototype:

```
int T32_GetSymbol ( const char *symbol,
                    uint32_t   *address,
                    uint32_t   *size,
                    uint32_t   *reserved );
```

Parameters:

```
symbol          ; pointer to symbol name
address         ; pointer to variable receiving the symbol address
size            ; pointer to variable receiving the symbol size (if any)
reserved        ; pointer to variable (reserved)
```

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` and `size` contain `(uint32_t)-1`.

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

Note: It is not possible to get the information of non-static local variables (as they have no address).

Example:

```
uint32_t address, size, reserved;

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, &reserved );

                     /* get information about a variable */

printf ("Symbol %s is located at 0x%x,\n", symname, address);
printf ("with a size of %d bytes.", size);
```

```
error = T32_GetSymbol ( srcline, &address, &size, &reserved );

                     /* get information about a source line */

printf ("Line 12 of file 'file.c' is located at 0x%x,\n", address);
printf ("the line is compiled occupying %d bytes of code.", size);
```

Actual workaround example for more complex symbols:

```
char      message[256];
uint16_t  mode;
int       address;

T32_Cmd ("print address.offset(v.address(ast.left))");
T32_GetMessage (message, &mode);
sscanf (message, "%x", &address);
printf ("ast.left = %8x\n", address);
```

Prototype:

```
int T32_GetSymbolFromAddress ( char *symbol,
                               uint32_t   address,
                               int        stringlength );
```

Parameters:

```
symbol           ; pointer to char array receiving the symbol name
address          ; symbol address
stringlength     ; maximum size symbol name (size of character array)
```

Returns:

0 for ok, otherwise communication error value.

This function returns the symbol name for a specified address.

Use **T32_SetMemoryAccessClass** if you need the symbol for a specific access class.

Example:

```
uint32_t address;
int      error;
char     symbol[256];

                        /* search symbol on address 0x1234 */

address = 0x1234;

error = T32_GetSymbolFromAddress ( symbol, address, 256 );

printf ("Symbol at address 0x%x is %s\n", address, symbol);
```

Prototype:

```
int T32_QuerySymbolObj (T32_SymbolHandle symbolHandle);
```

Parameters:

```
symbolHandle         ; handle to symbol object
```

Returns:

0 for ok, otherwise Error value

Queries information about a symbol.
If the symbol object is initialized with a symbol name, T32_QuerySymbolObj () fills the object with further information like symbol address.
If the symbol object is initialized with an address, T32_QuerySymbolObj () fills the object with further information like symbol name.

A “symbol handle” must be declared, requested and set by the application as shown in the example and description below.

Example to get a 32bit address of a given symbol name:

```
uint32_t myAddress;

T32_SymbolHandle  mySymbolHandle  = NULL;
T32_AddressHandle myAddressHandle = NULL;

T32_RequestSymbolObjName(&mySymbolHandle, "main");

T32_QuerySymbolObj(mySymbolHandle);

T32_GetSymbolObjAddress(mySymbolHandle, &myAddressHandle);
T32_GetAddressObjAddr32(myAddressHandle, &myAddress);

T32_ReleaseAllObjects ();   // release all T32 objects

/* symbol address is now in "myAddress" */
```

Example to get a symbol name of a given 32bit address:

```
char symName[64];

T32_SymbolHandle  mySymbolHandle;
T32_AddressHandle myAddressHandle;

T32_RequestAddressObjA32(&myAddressHandle, 0x1234);
T32_RequestSymbolObjAddr(&mySymbolHandle, myAddressHandle);

T32_QuerySymbolObj(mySymbolHandle);

T32_GetSymbolObjName(mySymbolHandle, symName, 64);

T32_ReleaseAllObjects ();   // release all T32 objects

/* symbol name is now in "symName" */
```

Address Object handling: For a description of the address object, see "**Address Object**".

Symbol Object handling: For a description of the symbol object, see "**Symbol Object**".

Prototype:

```
int T32_QueryAddressObjMmuTranslation (T32_AddressHandle handle,
    uint16_t translation);
```

Parameters:

```
handle                ; handle to address object

translation           ; type of translation
```

Returns:

0 for ok, otherwise Error value

Queries an MMU translation for a given address object.

`handle` specifies a fully qualified address object for which you want the translation. For a description of the address object, see “**Address Object**”.

`translation` specifies the type of translation that will be performed:

T32_MMUTRANSLATION_TO_PHYSICAL	translate to physical address
T32_MMUTRANSLATION_TO_LOGICAL	translate to logical (virtual) ddress
T32_MMUTRANSLATION_TO_LINEAR	translate to linear address (only x86/x64)

Example to translate a virtual address to physical address:

```
uint32_t log_address = 0x12345678L;
char log_access[16] = "D";
uint32_t phys_address;
char phys_access[32];
T32_AddressHandle myAddressHandle32;

T32_RequestAddressObjA32(&myAddressHandle32, log_address);
T32_SetAddressObjAccessString (myAddressHandle32, log_access);

T32_QueryAddressObjMmuTranslation (myAddressHandle32,
    T32_MMUTRANSLATION_TO_PHYSICAL);

T32_GetAddressObjAddr32 (myAddressHandle32, &phys_address);
T32_GetAddressObjAccessString (myAddressHandle32, phys_access, 32);

printf ("logical %s:%08x <==> physical %s:%08x\n",
    log_access, log_address, phys_access, phys_address);

T32_ReleaseAllObjects ();   // release all T32 objects
```

Prototype:

```
int T32_QueryAddressObjTargetSizeOfMau (T32_AddressHandle handle);
```

Parameters:

```
handle                ; handle to address object
```

Returns:

0 for ok, otherwise Error value

Queries the MAU (minimum addressable unit) of a target address in bits.

`handle` specifies a fully qualified address object for which you want the MAU size. For a description of the address object, see "**Address Object**".

This function sets an attribute in the object with the number if bits that a single address in the target system addresses. For "normal" memory, this will be 8 bits, but for special accesses (e.g. Arm CP15 registers), this may be different. Use **T32_GetAddressObjTargetSizeOfMau** to get the target MAU size of the address object.

NOTE:	This function only queries the target MAU size, it does not change the MAU addressing of the address object. If you want to change the addressing behavior of the address object when reading/writing the target, use **T32_SetAddressObjSizeOfMau**

Example to get the target MAU size of the Arm CP15 register area:

```
uint32_t mau;
T32_AddressHandle myAddressHandle;

T32_RequestAddressObjA32(&myAddressHandle, 0);
T32_SetAddressObjAccessString (myAddressHandle, "C15");

T32_QueryAddressObjTargetSizeOfMau (myAddressHandle);

T32_GetAddressObjTargetSizeOfMau (myAddressHandle, &mau);

printf ("target MAU size in bits for C15: %d.\n", mau);

T32_ReleaseAllObjects ();   // release all T32 objects
```

Prototype:

```
int T32_ReadVariableValue ( const char  *symbol,
                            uint32_t    *value,
                            uint32_t    *hvalue );
```

Parameters:

```
symbol           ; pointer to variable name
value            ; pointer to variable receiving the value
hvalue           ; pointer to variable receiving the upper 32bit
```

Returns:

0 for ok,
>0 for access error (e.g. symbol not found)
<0 for communication error.

This function provides the integer value for a specified variable name.
If the size of the variable is smaller or equal to 32bit, the value is stored in “value”.
If the size of the variable is 64bit, the upper 32bit are stored in “hvalue”.

Example:

```
uint32_t  value, hvalue;
int       state

state = T32_ReadVariableValue ("i", &value, &hvalue);

printf ("i = %d, state = %d, %s\n",
        value, state, state==0 ? "OK" : "NOK");
```

Prototype:

```
int T32_WriteVariableValue ( const char  *symbol,
                             uint32_t    value,
                             uint32_t    hvalue );
```

Parameters:

```
symbol           ; pointer to variable name
value            ; value to be written (lower 32bit)
hvalue           ; upper 32bit of value to be written
```

Returns:

0 for ok,
>0 for access error (e.g. symbol not found)
<0 for communication error.

This function sets the integer value for a specified variable name.
If the value does not fit into the variable, it is truncated to the size of the variable.

Example:

```
error = T32_WriteVariableValue ("i", 5, 0);
```

Prototype:

```
int T32_ReadVariableString ( const char  *symbol,
                             char        *string,
                             int          maxlen );
```

Parameters:

```
symbol          ; pointer to variable name
string          ; pointer to character array receiving the string
maxlen          ; maximum length of string (including zero termination)
```

Returns:

0 for ok,
>0 for access error (e.g. symbol not found)
<0 for communication error.

This function provides the content for a specified variable name as string.

Notes:

The lenght of the variable name is limited to 250 characters.

Example:

```
char  string[256];
int   state

state = T32_ReadVariableString ("i", string, sizeof(string));

printf ("i = \"%s\", state = %d, %s\n",
        string, state, state==0 ? "OK" : "NOK");
```

Prototype:

```
int T32_GetSource ( uint32_t  address,
                    char      filename[256],
                    uint32_t  *line );
```

Parameters:

```
address              ; address for which file and line are requested

filename             ; output parameter, set by API function

line                 ; output parameter, set by 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[256];
uint32_t 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], uint32_t *line );
```

Parameters:

```
filename          ; output parameter, set by API function

line              ; output parameter, set by API function
```

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.List`" or "`Data.List`").
If no previous selection was done, or if no source line is selected, the function returns with `filename` set to an empty string `(filename[0]=='\0')`.
`filename` **must** be an array of characters with at least 256 elements.

Example:

```
char     filename[256];
uint32_t 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");
```

This function is deprecated. Please use this function instead:

- T32_GetTraceState

Prototype:

```
int T32_AnaStatusGet ( uint8_t *state,
                       int32_t *size,
                       int32_t *min,
                       int32_t *max );
```

Parameters:

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

Returns:

0 for ok, otherwise communication error value

This function requests the state of the TRACE32 State Analyzer.
"`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:

```
uint8_t  state;
int32_t  size, min, max;

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

This function is deprecated. Please use this function instead:

- T32_ReadTrace

Prototype:

```
int T32_AnaRecordGet ( int32_t  recordnr,
                       uint8_t *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.
"`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 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	timestamp (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;
int32_t   recordnr = 100;
uint64_t  time;
uint8_t   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 += (uint64_t) buffer[20+i] << i*8;
     }
     printf ("\n");
     time = time * 625 / 8000;             /* calculate nanoseconds    */
  printf ("        = %u s, %u ms, %u us, %u ns\n",
          (unsigned int) (time / 1000000000L),
          (unsigned int) (time % 1000000000L / 1000000L),
          (unsigned int) (time % 1000000L / 1000L),
          (unsigned int) (time % 1000L));
}
```

Prototype:

```
int T32_GetTraceState ( int      tracetype,
                        int     *state,
                        int32_t *size,
                        int32_t *min,
                        int32_t *max );
```

Parameters:

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

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)
4	PowerProbe
5	Snooper
6	DTM

“`state`” contains the current trace state:

0	analyzer is switched off
1	analyzer is armed

2	analyzer triggered
3	analyzer recording breaked

"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:

```
int       state;
uint32_t  size, min, max;

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

Prototype:

```
int T32_ReadTrace ( int        tracetype,
                    int32_t    record,
                    int        n,
                    uint32_t   mask,
                    uint8_t   *buffer );
```

Parameters:

```
tracetype        ; type of trace and interpretation
record           ; record number of record to start reading from
n                ; number of records to read
mask             ; type of data to extract from the trace
buffer           ; 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 **T32_GetTraceState** 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 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 0 = Data bit 1 = Program bit 2 = First Cycle bit 3 = reserved bit 4 = Breakpoint Cycle bit 5 = reserved bit 6 = Write Cycle bit 7 = reserved
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;
int32_t   recordnr = 100;
uint64_t  time;
uint8_t   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 += (uint64_t) buffer[0+i] << i*8;
     }
     printf ("\n");
     time = time * 625 / 8000;       /* calculate nanoseconds */
     printf ("        = %u s, %u ms, %u us, %u ns\n",
          (unsigned int) (time / 1000000000L),
          (unsigned int) (time % 1000000000L / 1000000L),
          (unsigned int) (time % 1000000L / 1000L),
          (unsigned int) (time % 1000L));
}
```

This call function is deprecated. Please use one of these functions instead:

- **T32_NotifyBreakEnable**
- **T32_NotifyEditEnable**
- **T32_NotifyBreakConfigEnable**
- **T32_NotifyErrorEnable**
- **T32_NotifyRTSTriggerEnable**
- **T32_NotifyGoEnable**

Prototype:

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

Parameters:

```
event            ; number of the event to react on
function         ; pointer to callback 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 re-evaluate accumulated events.

"`event`" specifies the event type.

"`function`" points to a function that is called when the event occurred.

The following table shows the events and which callback function declarations are expected:

T32_E_BREAK	Core halted, e.g. on breakpoint

```
void callbackFunction (int parameter, uint64_t pc, uint64_t reason);
```

T32_E_EDIT	Remote API is called to open a file in an editor

```
void callbackFunction (int parameter, int lineNr, char* fileName);
```

T32_E_BREAKPOINTCONFIG	Breakpoints were changed

```
void callbackFunction (int parameter);
```

T32_E_RTSTRIGGER An RTS trigger event occurred

(for internal use only)

T32_E_ERROR An error occurred

```
void callbackFunction (int parameter, int code, char* message);
```

T32_E_GO Core state changed to running

```
void callbackFunction (int parameter);
```

See also **T32_NotifyEventEnable** .

NOTE:	Compile the API sources with `ENABLE_NOTIFICATION` defined to use notifications.

Example:

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

```
void targetHalted (int parameter, uint64_t pc, uint64_t reason)
{
    printf ("notifyStateBreak: target halted\n");
}

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

Prototype:

```
int T32_NotifyBreakEnable (
    void (*function)(int parameter, uint64_t pc, uint64_t reason) );
```

Parameters:

```
function         pointer to callback function that is called when the
                 event occurs
```

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 a break happens on the target (either running on a breakpoint, or a manual halt).
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 re-evaluate accumulated events.

See also **T32_NotifyEventEnable** .

NOTE:	Compile the API sources with `ENABLE_NOTIFICATION` defined to use notifications.

Example:

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

```
void targetHalted (int parameter, uint64_t pc, uint64_t reason)
{
    printf ("Notification: target halted on address %llx\n", pc);
}

void enableEvent (void) {
    if ( T32_NotifyBreakEnable(targetHalted) )
        printf ("Notification: Could not initialize! \n");
    else
        printf ( "Notification enabled.\n");
}
```

Prototype:

```
int T32_NotifyEditEnable (
    void (*function)(int parameter, int lineNr, const char* fileName) );
```

Parameters:

function	pointer to callback function that is called when the event occurs

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 user wants to edit a file within TRACE32.
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 re-evaluate accumulated events.

See also **T32_NotifyEventEnable** .

NOTE:	Compile the API sources with `ENABLE_NOTIFICATION` defined to use notifications.

Example:

Register the function editRequest to be called whenever the user wants to edit a file within TRACE32.

```
void editRequest (int parameter, int line, const char* file)
{
    printf ("Notification: edit request on line %d, file %s\n",
        line, file);
}

void enableEvent (void) {
    if ( T32_NotifyEditEnable(editRequest) )
        printf ("Notification: Could not initialize! \n");
    else
        printf ( "Notification enabled.\n");
}
```

Prototype:

```
int T32_NotifyBreakConfigEnable ( void (*function)(int parameter) );
```

Parameters:

```
function        pointer to callback function that is called when the
                event occurs
```

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 breakpoint configuration is changed.
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 re-evaluate accumulated events.

See also **T32_NotifyEventEnable** .

NOTE:	Compile the API sources with `ENABLE_NOTIFICATION` defined to use notifications.

Example:

Register the function breakConfig to be called whenever the breakpoint configuration is changed.

```
void breakConfig (int parameter)
{
    printf ("Notification: breakpoint configuration\n");
}

void enableEvent (void) {
    if ( T32_NotifyBreakConfigEnable(breakConfig) )
        printf ("Notification: Could not initialize! \n");
    else
        printf ( "Notification enabled.\n");
}
```

Prototype:

```
int T32_NotifyErrorEnable (
    void (*function)(int parameter, int code, const char* message) );
```

Parameters:

```
function         pointer to callback function that is called when the
                 event occurs
```

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 TRACE32 emits an error message.
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 re-evaluate accumulated events.

See also **T32_NotifyEventEnable** .

NOTE:	Compile the API sources with `ENABLE_NOTIFICATION` defined to use notifications.

Example:

Register the function errorMessage to be called whenever TRACE32 emits an error message.

```
void errorMessage (int parameter, int code, const char* message)
{
    printf ("Notification: error message: %s\n", message);
}

void enableEvent (void) {
    if ( T32_NotifyErrorEnable(errorMessage) )
        printf ("Notification: Could not initialize! \n");
    else
        printf ( "Notification enabled.\n");
}
```

Prototype:

```
int T32_NotifyRTSTriggerEnable (
    void (*function)(int parameter,
    uint64_t res1, uint64_t res2, uint64_t res3) );
```

Parameters:

`function`	`pointer to callback function that is called when the event occurs`

Returns:

0 for ok, otherwise communication error value

For internal use only

T32_NotifyGoEnable Register function to be called when core state change to running

Prototype:

```
int T32_NotifyGoEnable ( void (*function)(int parameter) );
```

Parameters:

```
function        pointer to callback function that is called when the
                event occurs
```

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 core state is changed to running.
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 re-evaluate accumulated events.

See also **T32_NotifyEventEnable** .

NOTE:	Compile the API sources with `ENABLE_NOTIFICATION` defined to use notifications.

Example:

Register the function Go to be called whenever the core state is changed to running.

```
void Go (int parameter)
{
    printf ("Notification: Core State changed to running\n");
}

void enableEvent (void) {
    if ( T32_NotifyGoEnable(Go) )
        printf ("Notification: Could not initialize! \n");
    else
        printf ( "Notification enabled.\n");
}
```

Prototype:

```
int T32_NotifyEventEnable ( char* event, void (*function) (int) );
```

Parameters:

```
event            ; name of "ON" event to react on
function         ; pointer to callback function, NULL for unregister
```

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. All events that are available with the **ON** command are allowed.
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 re-evaluate accumulated events.

"`event`" specifies the event name as available with the ON command, e.g. "PBREAK".
"`function`" points to a function that is called when the event takes place.
The parameter passed to the callback function is the parameter given to **T32_CheckStateNotify**.

See also **T32_NotifyStateEnable** .

NOTE:	Compile the API sources with `ENABLE_NOTIFICATION` defined to use notifications.

Example:

Register the function sysupEvent to be called whenever the debugger goes into "System Up" state.

```
void sysupEvent (int arg) {
    printf ("--- SYSUP event happened ---\n");
}
```

```
int main () {
...
    if ( T32_NotifyEventEnable ("SYSUP", sysupEvent) )
        printf ("Notify Sysup: registration failed! \n");
    else
        printf ("Sysup notificiation initialized.\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 re-evaluate events accumulated since the last call to T32_CheckStateNotify. If a callback function for any of these events was registered with **T32_NotifyStateEnable** or **T32_NotifyEventEnable**, the appropriate 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 its own thread, it is the application program's responsibility to periodically call this function.

When compiling the API sources with `ENABLE_AUTONOTIFY` defined, all high-level API functions call `T32_CheckStateNotify(0)` at the end of the function call, thus enabling an automatic evaluation of the events without manually calling T32_CheckStateNotify. However, the callbacks are still only called if API functions are used; as long as no API functions are called, no callback function will be called either.

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 ( WSAGETSELECTEVENT(lParam) )
          {
          case FD_READ:
               T32_CheckStateNotify(&apphandle);
               break;
          }
          break;
     case WM_DESTROY:        /* message: window being destroyed        */
          break;
     default:                /* Passes it on if unproccessed           */
          return ( DefWindowProc(hWnd, message, wParam, lParam) );
     }
     return (0);
}
```

Prototype:

```
int T32_APILock ( int Timeout );
```

Parameters:

```
Timeout        ; timeout of lock command in milliseconds
```

Returns:

T32_OK or error code. When the system is already locked by other client, the function returns T32_ERR_STD_LOCKED.

This function can be used to create a critical section when multiple clients access one instance of TRACE32. Note that all clients must use this function. Clients not using this function can access the Remote API as usual. Note that the lock is automatically released after 2 seconds, which can be extended by calling T32_APILock again.

Timeout can have different values:

0	Lock if RemoteAPI server is not in use or return T32_ERR_STD_LOCKED to indicate that the system is locked already by another client.
n	Wait n ms to get the lock. Returns T32_ERR_STD_LOCKED if the command was not successful. The TIMEOUT= setting must be increased to a value greater n.

Prototype:

```
int T32_APIUnlock ( void );
```

Parameters:

```
none
```

Returns:

0 for OK, otherwise error value.

Release the Remote API lock acquired by calling **T32_APILock**.

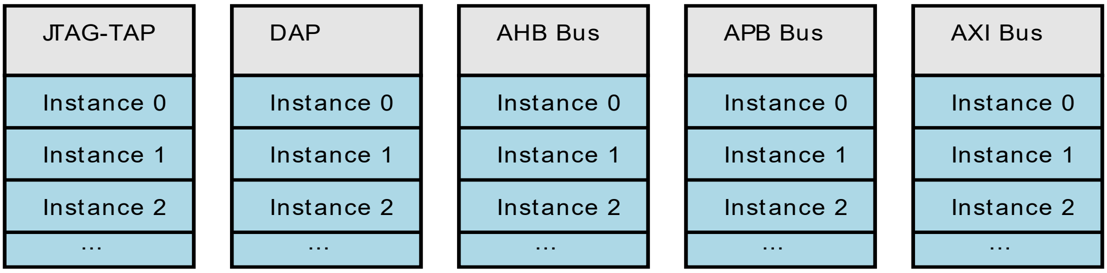
Example:

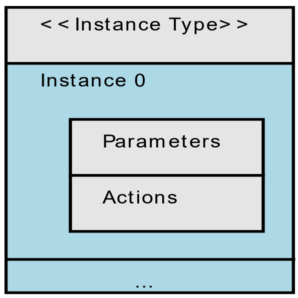
With 2 clients. Each client has its own port number..

```
;client 1
while ((1)) {
T32_APILock(10000);
T32_Cmd("print 1.");
T32_Cmd("print 2.");
T32_Cmd("print 3.");
T32_APIUnlock();
};

;client 2
while ((1)) {
T32_APILock(10000);
T32_Cmd("print 4.");
T32_Cmd("print 5.");
T32_Cmd("print 6.");
T32_APIUnlock();
};
;This will print
;......,4,5,6,1,2,3,4,5,6,1,2,3,4,5,6,1,2,3,4,5,6,....
;in the area window when both clients run.
```

The Direct Access API Functions are used to access IP blocks of the target that are not handled by the debugger. The API provides functions at different abstraction level to minimize the communication overhead. So it is possible to toggle certain pin of the debuggers probe, program complete JTAG shifts or modify registers on a certain bus. The API functions are divided into functions that set parameters and functions that execute actions based to the previously set parameters to save communication time. Multiple sets of parameters are possible, because there are multiple IP block in the target. These sets are called Instances. Depending to the accessed IP blocks there are different instance types that can be used.



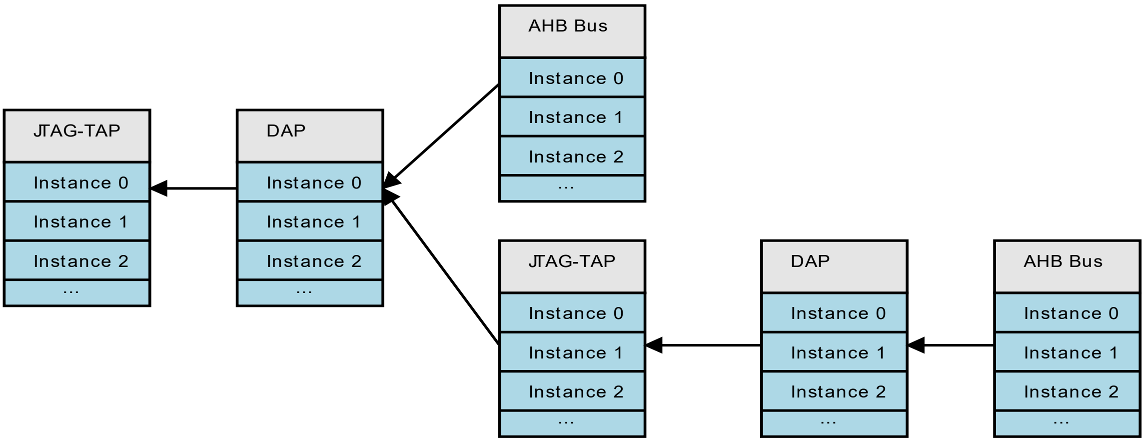


Following instance types can be used:

Instance Type	Identifier	Meaning
JTAG-TAP	T32_DIRECTACCESS_INSTANCETYPE_TAP	JTAG-TAP that can be accessed by debugger probe or by target internal source e.g. JTAG-AP of a DAP
DAP	T32_DIRECTACCESS_INSTANCETYPE_DAP	Arm Debug Access Port that can be accessed by JTAG or Serial Wire Debug (SWD).
AHB Bus	T32_DIRECTACCESS_INSTANCETYPE_AHB	Arm AHB Bus that can be accessed by a DAP.

APB Bus	T32_DIRECTACCESS_INSTANCETYPE_APB	Arm APB Bus that can be accessed by a DAP.
AXI Bus	T32_DIRECTACCESS_INSTANCETYPE_AXI	Arm AXI Bus that can be accessed by a DAP.

In the target system the IP-Blocks are interconnected with the result that the communication protocols are encapsulated along the connection path from the physical pins to a certain IP-Block. To model these interconnections the API can also interconnect its instances by setting parameters.



Most of the parameters can be read by `T32_DirectAccessGetInfo` or written by `T32_DirectAccessSetInfo`. To speed to the communication the API functions allow bundle accesses by `T32_BundledAccess` functions.

Many parameters of the debug port are not handled by API parameters and functions. They must be configured by `T32_Cmd` as done in the PRACTICE scripts e.g.:

```
T32_Cmd("SYStem.JtagClock 10Mhz");  //Setup JTAG clock for API calls
```

Before any function can access to the target it is necessary to enable the output driver of the debug port:

```
uint8_t cmd;
cmd = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
T32_TAPAccessDirect(T32_DIRECTACCESS_HOLD, 1, &cmd, NULL);
```

Bundled Accesses and Exclusive Access

There are two possible modes to access the debug port,

- the Single Access Mode and
- the Bundled Access Mode.

For a sequence of accesses (e.g. to read memory on a lower abstraction level), the Bundled Access Mode is recommended.

For Single Access Mode, two predefined Handles are available, which control the behavior of the debugger after the API access:

Handle for Single Access Mode	Effect
`T32_DIRECTACCESS_HOLD`	All debugger actions concerning the debug port will be suspended. The API has exclusive access to the debug port.
`T32_DIRECTACCESS_RELEASE`	Allows the debugger to access the debug port after this API access

For Bundled Access Mode, the access handle must be acquired by calling `T32_BundledAccessAlloc`. All accesses will be stored, instead of being executed immediately. Those bundled accesses are executed with a call to `T32_BundledAccessExecute` in the given order. While a bundled access is executed, the API holds exclusive access to the debug port. `T32_BundledAccessExecute` can be called multiple times, but finally `T32_BundledAccessFree` must be called to free the allocated memory.

Prototype:

```
T32_BUNDLEDACCESS_HANDLE T32_BundledAccessAlloc ( void );
```

Parameters:

none

Returns:

Handle for bundled accesses

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

Example:

```
uint8_t status;
uint8_t pvrnr[4];
uint8_t tap_instr = TAP_COP_PVR;

T32_BUNDLEDACCESS_HANDLE handle = T32_BundledAccessAlloc ();

T32_TAPAccessShiftIR (handle, 8, &tap_instr, &status);
T32_TAPAccessShiftDR (handle, 32, NULL, pvrnr);
T32_TAPAccessExecute (handle, T32_DIRECTACCESS_RELEASE);

T32_BundledAccessFree (handle);
```

Prototype:

```
int T32_BundledAccessFree ( T32_BUNDLEDACCESS_HANDLE connection );
```

Parameters:

```
connection    ; access handle
```

Returns:

0 for ok, otherwise Error value

Use this function to release the handle returned by `T32_BundledAccessAlloc` when it is no longer needed.

Example:

see `T32_BundledAccessAlloc` for an example

Prototype:

```
int T32_BundledAccessExecute ( T32_BUNDLEDACCESS_HANDLE connection,
                               T32_BUNDLEDACCESS_HANDLE connectionhold);
```

Parameters:

```
connection          ; Handle for a bundled access
connectionhold      ; access handle
```

Returns:

0 for ok, otherwise Error value

Use this function, to execute all actions associated with given handle.

Example:

see T32_BundledAccessAlloc for an example

Prototype:

```
int T32_DirectAccessRelease ( void );
```

Parameters:

none

Returns:

0 for ok, otherwise Error value

If debugger accesses are suspended due to direct access or the `T32_BundledAccessExecute` call with the access handle `T32_DIRECTACCESS_HOLD`, use this function to resume debugger accesses.

Example:

```
// Retrieve the PVR value (PowerPC)
uint8_t status;
uint8_t pvrnr[4];
uint8_t tap_instr = TAP_COP_PVR;

T32_TAPAccessShiftIR (T32_DIRECTACCESS_HOLD, 8, &tap_instr, &status);
T32_TAPAccessShiftDR (T32_DIRECTACCESS_HOLD, 32, NULL, pvrnr);

// At this point, the debugger is still locked

T32_DirectAccessRelease ();
```

T32_ParamFromUint32 — Set instance parameter

Prototype:

```
T32_Param T32_ParamFromUint32(uint32_t value);
```

Parameters:

`value`	initialization value

Returns:

union containing the data for the passed value.

The function is used to create `T32_Param` union that can be passed to functions without the need of temporary variables.

Example:

```
//Set IRPRE 4 of a JTAG-TAP instance index 2
T32_DirectAccessSetInfo(
   T32_DIRECTACCESS_HOLD, T32_DIRECTACCESS_INSTANCETYPE_TAP, 2,
   T32_DIRECTACCESS_TAP_IRPRE_UINT32, T32_ParamFromUint32(4));
```

T32_DirectAccessSetInfo — Set instance parameter

Prototype:

```
int T32_DirectAccessSetInfo(
    T32_BUNDLEDACCESS_HANDLE Handle,
    int nInstanceType,
    unsigned int nInstance,
    int nInfoID,
    T32_Param value);
```

Parameters:

`Handle`	bundled access handle
`nInstanceType`	instance type. see Instance Types Identifier.

`nInstance`	instance index.
`nInfoID`	Instance type depended parameter ID.
`value`	new value for parameter

Returns:

0 for ok, otherwise Error value

Use this function to configure parameters of a certain instance.

Example:

```
//Set IRPRE 4 of a JTAG-TAP instance index 2
T32_DirectAccessSetInfo(
   T32_DIRECTACCESS_HOLD, T32_DIRECTACCESS_INSTANCETYPE_TAP, 2,
   T32_DIRECTACCESS_TAP_IRPRE_UINT32, T32_ParamFromUint32(4));
```

T32_DirectAccessGetInfo — Set instance parameter

Prototype:

```
int T32_DirectAccessGetInfo(
    T32_BUNDLEDACCESS_HANDLE Handle,
    int nInstanceType,
    unsigned int nInstance,
    int nInfoID,
    T32_Param *value);
```

Parameters:

`Handle`	bundled access handle
`nInstanceType`	instance type. see Instance Types Identifier.
`nInstance`	instance index.
`nInfoID`	Instance type depended parameter ID.
`value`	return value for parameter

Returns:

0 for ok, otherwise Error value

Use this function get parameters of a certain instance.

Example:

```
//Get IRPRE of a JTAG-TAP instance index 2
T32_Param res;
T32_DirectAccessSetInfo(
   T32_DIRECTACCESS_HOLD, T32_DIRECTACCESS_INSTANCETYPE_TAP, 2,
   T32_DIRECTACCESS_TAP_IRPRE_UINT32, &res);
printf("result was %d", res.uint32);
```

Parameter: automatic Tristate

- **Identifier**: T32_DIRECTACCESS_TRISTATE_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	**Effect**
0	no action
1	In multi-debugger mode, this parameter specifies the state of the debug port, which is expected when the debugger takes control and set before the debugger switches to Tristate mode. This value has to be identical for all debuggers connected to this debug port.

- **Default:** 0

Parameter: Debug Port is in Serial Wire Mode

- **Identifier**: T32_DIRECTACCESS_SWD_UNIT32
- **Set**: No, **Get**: Yes
- **Type**: UINT32

Values	**Effect**
0	`SYStem.CONFIG.DebugPortType` is not SWD
1	`SYStem.CONFIG.DebugPortType` is SWD

- **Effect**: In SWD mode the function `T32_DAPAccessScan` has a different behavior. See `T32_DAPAccessScan`.

- **Identifier**: T32_DIRECTACCESS_INSTANCE_EXISTS_UNIT32
- **Set**: No, **Get**: Yes
- **Type**: UINT32

Values	Effect
0	instance is not configured
1	instance is configured by a previous call of `T32_DirectAccessSetInfo`

Prototype:

```
int T32_DirectAccessResetAll (T32_TAPACCESS_HANDLE  Handle);
```

Parameters:

`Handle`	TAP access handle

Returns:

0 for ok, otherwise Error value

Effect:

All parameter data and instances will be reset to the state before any API call was made.

This chapter describes all functions available for direct access to the JTAG TAP controller.

Before calling any of the JTAG access functions described below, enable the debugger's trace port:

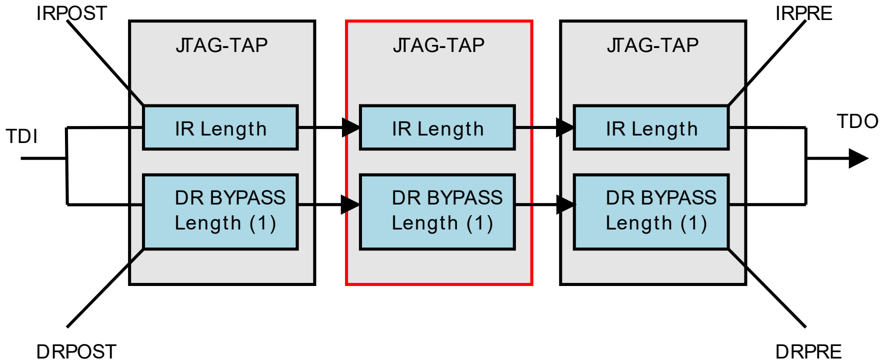
```
//Enable output of debug port driver
buffer[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
if (T32_TAPAccessDirect(T32_DIRECTACCESS_HOLD, 1, buffer, NULL))
    goto error;
```

The functions `T32_TAPAccessShiftIR`, `T32_TAPAccessShiftDR` and `T32_TAPAccessDirect` are provided for JTAG access. These functions need a handle to access the TAP controller.

Parameter: IRPRE, IRPOST, DRPRE, DRPOST

- **Identifier**: T32_DIRECTACCESS_TAP_IRPRE_UINT32, T32_DIRECTACCESS_TAP_IRPOST_UINT32, T32_DIRECTACCESS_TAP_DRPRE_UINT32, T32_DIRECTACCESS_TAP_DRPOST_UINT32
- **Type**: UINT32
- **Set**: Yes, **Get**: Yes
- **Effect**: Configures the position of the TAP controller within the JTAG Chain. The IR parameters describe the instruction register width, the DR parameters are used for the data register width when the BYPASS instruction was issued. Usually the data register of one TAP has a width of one in this case. The PRE parameters are used to describe the amount of bits that are shifted before the data of the accesses TAP controller is shifted in order to complete the shift. The POST

parameters are used to define how many bits are shifted after the data of the accessed TAP controller. The TAP position is used for higher level shift functions that are dedicated to the accessed TAPs instruction or data register.



Parameter: Parking TAP state after register shift

- **Identifier**: T32_DIRECTACCESS_TAP_PARKSTATE_UINT32
- **Set**: Yes, **Get**: Yes
- **Type**: UINT32

Values	Parking state
T32_TAPSTATE_RUN_TEST_IDLE	RUN-TEST-IDLE
T32_TAPSTATE_SELECT_DR_SCAN	SELECT-DR-SCAN. Shifts will be more efficient, but the RUN-TEST-IDLE state will never be reached.

- **Default:** depend to architecture of started TRACE32 executable.
- **Effect**: configure to which TAP state the state machine shall be driven after an access to an DR or IR register is done. The option will reset all shift pattern defined by `T32_TAPAccessSetShiftPattern`.

Parameter: Multi-Core TAP state

- **Identifier**: T32_DIRECTACCESS_TAP_MCTAPSTATE_UINT32
- **Set**: Yes, **Get**: Yes
- **Type**: UINT32

Values	Parking state
T32_TAPSTATE_RUN_TEST_IDLE	RUN-TEST-IDLE
T32_TAPSTATE_SELECT_DR_SCAN	SELECT-DR-SCAN

- **Default:** depends to the option `SYStem.CONFIG.TAPSTATE`
- **Effect**: The multi-core TAP state is used to allow other components to continue operation from a certain TAP state without re-initializing the TAP controller. When another TAP controller instance is accessed the TAP state changes from the parking state to the multi-core TAP state and then from the multi-core TAP state to the new specific park state. See also parameter `T32_DIRECTACCESS_TAP_AUTO_MULTICORETAPSTATE_UINT32`. The TAP state changing is done by an empty data register path shift. Therefore it's recommended to shift the BYPASS instruction before the access is switched from the current TAP controller instance to another one.

Parameter: electrical TCK pin configuration

- **Identifier**: T32_DIRECTACCESS_TAP_MCTCKLEVEL_UINT32
- **Set**: Yes, **Get**: Yes
- **Type**: UINT32

Values	Parking state
0	TCK has no pull-up resistor
1	TCK has a pull-up resistor

- **Default:** depends to `SYStem.CONFIG.TCKLevel`
- **Effect**: When the state machine changes from multi-core tap state to park tap state the extra TCK cycle is considered that was generated by the pull-up resistor when the debug port changes to tristate.

Parameter: Switch for Multi-Core TAP state behavior

- **Identifier**: T32_DIRECTACCESS_TAP_AUTO_MULTICORETAPSTATE_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	Parking state
0	inactive
1	active

- **Default:** active
- **Effect**: When the option is active the API will enter/leave the multi-core TAP state automatically when the access to this TAP controller instance is switched. The Multi-core TAP state is only necessary for IP blocks that are accessed by different multiple tools or software parts. **It is recommended to set the option to inactive in case the API is the only component that accesses this TAP controller.** It is also possible to set the option to inactive and do this actions by regular API calls.

Parameter: Select TAP controller instance for next commands

- **Identifier**: T32_DIRECTACCESS_TAP_SELECT_SHIFT_PATTERN_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32
- **Default:** 0
- **Effect**: The parameter set up the TAP controller instance for the commands `T32_TAPAccessSetInfo, T32_TAPAccessShiftRaw, T32_TAPAccessShiftIR, T32_TAPAccessShiftDR.`

Parameter: Select predefined shift pattern

- **Identifier**: T32_DIRECTACCESS_TAP_SELECT_SHIFT_PATTERN_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32
- **Default:** 0
- **Effect**: Select pattern that was configured by `T32_TAPAccessSetShiftPattern` to be used for the next `T32_TAPAccessShiftIR` or `T32_TAPAccessShiftDR` commands.

Parameter: Configure JTAG TAP behind DAP

- **Identifier**: T32_DIRECTACCESS_TAP_DAP_INSTANCE_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	Parking state
0xFFFFFFFF	No DAP instance selected
0..n	Used DAP instance

- **Default:** 0xFFFFFFFF
- **Effect**: When this parameter is set, the JTAG TAP instance is behind a JTAG-AP of a DAP controller.

Parameter: DAP access port

- **Identifier**: T32_DIRECTACCESS_TAP_DAP_ACCESSPORT_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32
- **Default:** 0
- **Effect**: Set the used DAP access port for the JTAG-AP

Parameter: JTAG port of JAG-AP of DAP access port

- **Identifier**: T32_DIRECTACCESS_TAP_DAP_JTAGACCESSPORTINDEX_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32
- **Default:** 0
- **Effect**: Set the JTAG port of the JTAG-AP of the DAP

Example:

```
//Shift Bypass pattern on JTAG-AP Tap

//Configuration
#define PRIMARY_TAP_INSTANCE_INDEX 0
#define JTAGAP_TAP_INSTANCE_INDEX 1
#define DAP_INSTANCE_INDEX 0
//analoge to setting SYStem.CONFIG.JTAGACCESSPORT 2.
#define JTAGAP_ACCESSPORT 2
 //analoge to setting SYStem.CONFIG.COREJTAGPORT  6.
#define JTAGAP_ACCESSPORT_INDEX 5

//Setup Debug Port
if (T32_Cmd("SYStem.JtagClock 1Mhz"))
    goto error;

//Reset previous configuration
if (T32_DirectAccessResetAll(T32_DIRECTACCESS_HOLD))
    goto error;

//----------------- Configure Primary JTAG ----------------------------
//set park state to Select-DR-Scan
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
    T32_DIRECTACCESS_INSTANCETYPE_TAP, PRIMARY_TAP_INSTANCE_INDEX,
    T32_DIRECTACCESS_TAP_PARKSTATE_UINT32,
    T32_ParamFromUint32(T32_TAPSTATE_SELECT_DR_SCAN))) goto error;

//----------------- Configure DAP -------------------------------------
//set JTAG TAP instance
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
    T32_DIRECTACCESS_INSTANCETYPE_DAP, DAP_INSTANCE_INDEX,
    T32_DIRECTACCESS_DAP_TAP_INSTANCE_UINT32,
    T32_ParamFromUint32(PRIMARY_TAP_INSTANCE_INDEX))) goto error;

//------------------ Configure JTAG-AP --------------------------------
//set DAP instance
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
    T32_DIRECTACCESS_INSTANCETYPE_TAP, JTAGAP_TAP_INSTANCE_INDEX,
    T32_DIRECTACCESS_TAP_DAP_INSTANCE_UINT32,
    T32_ParamFromUint32(DAP_INSTANCE_INDEX))) goto error;
//set JTAG Access Port
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
    T32_DIRECTACCESS_INSTANCETYPE_TAP, JTAGAP_TAP_INSTANCE_INDEX,
    T32_DIRECTACCESS_TAP_DAP_ACCESSPORT_UINT32,
    T32_ParamFromUint32(JTAGAP_ACCESSPORT))) goto error;
//set JTAG Access Port
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
    T32_DIRECTACCESS_INSTANCETYPE_TAP, JTAGAP_TAP_INSTANCE_INDEX,
    T32_DIRECTACCESS_TAP_DAP_JTAGACCESSPORTINDEX_UINT32,
    T32_ParamFromUint32(JTAGAP_ACCESSPORT_INDEX))) goto error;
```

```
//------------------ Start Actions ------------------------------------
//Enable output of debug port driver
buffer[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
if (T32_TAPAccessDirect(T32_DIRECTACCESS_HOLD, 1, buffer, NULL))
    goto error;

//Reset Primary JTAG
if (T32_TAPAccessJTAGResetWithTMS(T32_DIRECTACCESS_HOLD,
    PRIMARY_TAP_INSTANCE_INDEX))
    goto error;

//Select Secondary JTAG for further operations that don't provide the
instance parameter
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
T32_DIRECTACCESS_INSTANCETYPE_TAP, JTAGAP_TAP_INSTANCE_INDEX,
T32_DIRECTACCESS_TAP_CURRENTINSTANCE_UINT32,
T32_ParamFromUint32(JTAGAP_TAP_INSTANCE_INDEX)))
    goto error;

//Reset Secondary JTAG
if (T32_TAPAccessJTAGResetWithTMS(T32_DIRECTACCESS_HOLD,
    JTAGAP_TAP_INSTANCE_INDEX))
    goto error;

//Execute Shift IR BYPASS command for 32bit JTAG-AP TAP
buffer[0]=0xFF; buffer[1]=0xFF; buffer[2]=0xFF; buffer[3]=0xFF;
if (T32_TAPAccessShiftIR(T32_DIRECTACCESS_HOLD, 32, buffer, NULL))
   goto error;

error:
//Release Direct Access API
    T32_DirectAccessRelease();
```

Prototype:

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

Parameters:

`irpre`	Number of instruction register bits of all cores in the JTAG chain between the dedicated core and the TDO signal pin. The setting is the same as `T32_DIRECTACCESS_TAP_IRPRE_UINT32`.
`irpost`	Number of instruction register bits of all cores in the JTAG chain between TDI signal and the dedicated core. The setting is the same as `T32_DIRECTACCESS_TAP_IRPOST_UINT32`.
`drpre`	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. The setting is the same as `T32_DIRECTACCESS_TAP_DRPRE_UINT32`.
`drpost`	Number of cores in the JTAG chain between the TDI signal and the dedicated core (one data register bit per core which is in BYPASS mode). The setting is the same as `T32_DIRECTACCESS_TAP_DRPOST_UINT32`.
`tristate`	TRUE, if more than one debugger is connected to JTAG port. With this option, the debugger switches to tristate mode after each access. The setting is the same as `T32_DIRECTACCESS_TRISTATE_UINT32`.
`tapstate`	In multi-debugger mode, this parameter specifies the state of the TAP controller, which is expected when the debugger takes control and set before the debugger switches to tristate mode. This value has to be identical for all debuggers connected to this JTAG port. The setting is the same as `T32_DIRECTACCESS_TAP_MCTAPSTATE_UINT32`. See table below for a list of possible states
`tcklevel`	In multi-debugger mode, this is the level of the TCK signal when all debuggers are tristated. The setting is the same as `T32_DIRECTACCESS_TAP_MCTCKLEVEL_UINT32`.
`reserved`	no effect. leave 0.

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  handle,
                           int                   numberofbits,
                           uint8_t              *poutbits,
                           uint8_t              *pinbits );
```

Parameters:

`handle`	TAP access handle
`numberofbits`	amount of bits to scan
`poutbits`	buffer containing data scanned into the TAP controller, or NULL to scan in Zeros
`pinbits`	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:

```
uint8_t status;
uint8_t tap_instr = TAP_STATUS;

T32_TAPAccessShiftIR (T32_DIRECTACCESS_RELEASE, 8, &tap_instr, &status);
```

Prototype:

```
int T32_TAPAccessShiftDR ( handle  connection,
                           int                  numberofbits,
                           uint8_t             *poutbits,
                           uint8_t             *pinbits );
```

Parameters:

`handle`	TAP access handle
`numberofbits`	amount of bits to scan
`poutbits`	buffer containing data scanned into the TAP controller, or NULL to scan in Zeros
`pinbits`	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)
uint8_t status;
uint8_t pvrnr[4];
uint8_t tap_instr = TAP_COP_PVR;

T32_TAPAccessShiftIR (T32_DIRECTACCESS_HOLD, 8, &tap_instr, &status);
T32_TAPAccessShiftDR (T32_DIRECTACCESS_RELEASE, 32, NULL, pvrnr);
// Write Zeros
```

Prototype:

```
int T32_TAPAccessDirect ( T32_TAPACCESS_HANDLE  handle,
                          int                   nbytes,
                          uint8_t              *poutbytes,
                          uint8_t              *pinbytes );
```

Parameters:

`handle`	TAP access handle
`nbytes`	size in bytes of the array poutbytes
`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. **NOTE:** This command does not work with return clock from target (RTCK). Clock accurate arbitrary shifts should be done by **"T32_TAPAccessShiftRaw RAW JTAG Shifts"** (api_remote_c.pdf).

NOTE: Availability and functionality of direct access commands depends on the used debugger and/or target hardware.

Example:

```
// reset target
uint8_t commands[8];
uint8_t result[8];
uint8_t 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_DIRECTACCESS_RELEASE, 8, commands, result);

hreset_state = result[7];
```

Prototype:

```
int T32_TAPAccessJTAGResetWithTMS ( T32_TAPACCESS_HANDLE  Handle,
                            unsigned int           nTapInstance);
```

Parameters:

`Handle`	TAP access handle
`nTapInstance`	JTAG Tap instance index

Returns:

0 for ok, otherwise Error value

Effect:

The function drives the JTAG state machine through Test-Logic-Rest and enter the park state defined by parameter `T32_DIRECTACCESS_TAP_PARKSTATE_UINT32`. The function must be used in case in case the JTAG Tap behind a JTAG-AP of a DAP.

Example:

```
//Enable output of debug port driver
buffer[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
if (T32_TAPAccessDirect(T32_DIRECTACCESS_HOLD, 1, buffer, NULL))
    goto error;

//Reset JTAG
if (T32_TAPAccessJTAGResetWithTMS(T32_DIRECTACCESS_HOLD,0))
    goto error;
```

Prototype:

```
int T32_TAPAccessJTAGResetWithTRST ( T32_TAPACCESS_HANDLE  Handle,
                            unsigned int            nTapInstance,
                            int32_t nTRSTAssertTimeUS,
                            int32_t nDelayAfterTRSTReleaseUS);
```

Parameters:

`Handle`	TAP access handle
`nTapInstance`	JTAG Tap instance index
`nTRSTAssertTimeUS`	Duration of TRST signal is asserted: -1 : 10[us] + 1 JTAG clock cycle 0 <= t <= n : t [us]
`nDelayAfterTRSTReleaseUS`	Pause time after TRST is de-asserted -1 : 20[us] + 1 JTAG clock cycle 0 <= t <= n : t [us]

Returns:

0 for ok, otherwise Error value

Effect:

The function uses the TRST signal to set the JTAG state Test-Logic-Rest and enter the park state defined by parameter `T32_DIRECTACCESS_TAP_PARKSTATE_UINT32` finally. The function must be used in case in case the JTAG Tap behind a JTAG-AP of a DAP.

Example:

```
//Enable output of debug port driver
buffer[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
if (T32_TAPAccessDirect(T32_DIRECTACCESS_HOLD, 1, buffer, NULL))
    goto error;

//Reset JTAG
if (T32_TAPAccessJTAGResetWithTRST(T32_DIRECTACCESS_HOLD,0,-1,-1))
    goto error;
```

Prototype:

```
int T32_TAPAccessSetShiftPattern ( T32_TAPACCESS_HANDLE  handle,
                                   unsigned int nTapInstance,
                                   uint32_t nReturnIRCount,
                                   uint32_t nReturnDRCount,
                                   uint32_t nGotoIRCount,
                                   uint32_t nGotoDRCount,
                                   uint32_t nReturnIR,
                                   uint32_t nReturnDR,
                                   uint32_t nGotoIR,
                                   uint32_t nGotoDR,
                                   unsigned int nPattern)
```

Parameters:

`handle`	TAP access handle
`nTapInstance`	JTAG TAP instance
`nReturnIRCount`	Number of bits shifted from Exit1-IR to the park state
`nReturnDRCount`	Number of bits shifted from Exit1-DR to the park state
`nGotoIRCount`	Number of bits shifted from the park state to Shift-IR.
`nGotoDRCount`	Number of bits shifted from the park state to Shift-DR.
`nReturnIR`	Pattern used to shift from Exit1-IR to the park state
`nReturnDR`	Pattern used to shift from Exit1-DR to the park state
`nGotoIR`	Pattern used to shift from the park state to Shift-IR
`nGotoDR`	Pattern used to shift from the park state to Shift-DR
`nPattern`	Pattern index to specify which parameter set is changed. Up to 16 parameter sets can be used that can be selected by parameter `T32_DIRECTACCESS_TAP_SELECT_SHIFT_PATTERN_UINT32` later.

Returns:

0 for ok, otherwise Error value

The function is used to define how the functions `T32_TAPAccessShiftIR` and `T32_TAPAccessShiftDR` enter and the shift TAP state Shift-IR or Shift-DR from the parking state and how the TAP state is changed from Exit1-IR/Exit1-DR to the park state. The parameters that describe the pattern are used starting by the least significant bit e.g. when `nReturnDRCount` is 0x5 and `nReturnDR` is 0x1, then the TAP states Update-DR (1), Run-Test-Idle (0), Run-Test-Idle (0), Run-Test-Idle (0), Run-Test-Idle (0) are driven.

The parameter `T32_DIRECTACCESS_TAP_PARKSTATE_UINT32` resets all values defined by this function to the following defaults:

Parameter / Park state	**Run-Test-Idle**	**Select-DR-Scan**
`nReturnIRCount`	2	2
`nReturnDRCount`	2	2
`nGotoIRCount`	4	3
`nGotoDRCount`	3	2
`nReturnIR`	1	3
`nReturnDR`	1	3
`nGotoIR`	3	1
`nGotoDR`	1	1

Example:

```
//Enable output of debug port driver
buffer[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
if (T32_TAPAccessDirect(T32_DIRECTACCESS_HOLD, 1, buffer, NULL))
    goto error;

//Initialize all patterns by default and set park state to Select-DR-Scan
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
 T32_DIRECTACCESS_INSTANCETYPE_TAP, 3 /*Instance*/,
 T32_DIRECTACCESS_TAP_PARKSTATE_UINT32,
 T32_ParamFromUint32(T32_TAPSTATE_SELECT_DR_SCAN)))
    goto error;

//Reset JTAG
if (T32_TAPAccessJTAGResetWithTRST(T32_DIRECTACCESS_HOLD,3,-1,-1))
    goto error;
```

```
//Reconfigure pattern with index 5 to execute 3 Run-Test-Idle cycles
if (T32_TAPAccessSetShiftPattern(T32_DIRECTACCESS_HOLD, 3 /*Instance*/,
/* nReturnIRCount*/ 5,
/* nReturnDRCount*/ 5,
/* nGotoIRCount*/ 3,
/* nGotoDRCount*/ 2,
/* nReturnIR*/ 0x11,//-> Update-IR -> 3*Run-Test-Idle -> Select-DR-Scan
/* nReturnDR*/ 0x11,//-> Update-DR -> 3*Run-Test-Idle -> Select-DR-Scan
/* nGotoIR*/ 1,
/* nGotoDR*/ 1,
/* pattern*/ 5)) goto error;

//Select pattern 5 for TAP instance 3
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
 T32_DIRECTACCESS_INSTANCETYPE_TAP, 3 /*Instance*/,
 T32_DIRECTACCESS_TAP_SELECT_SHIFT_PATTERN_UINT32,
 T32_ParamFromUint32(5)))
    goto error;

//Select TAP instance 3 for the next access of T32_TAPAccessShiftIR
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
 T32_DIRECTACCESS_INSTANCETYPE_TAP, 3 /*Instance*/,
 T32_DIRECTACCESS_TAP_CURRENTINSTANCE_UINT32,
 T32_ParamFromUint32(3)))
    goto error;

//Execute Shift IR BYPASS command for 32bit TAP
buffer[0]=0xFF; buffer[1]=0xFF; buffer[2]=0xFF; buffer[3]=0xFF;
if (T32_TAPAccessShiftIR(T32_DIRECTACCESS_HOLD, 32, buffer, NULL))
   goto error;
```

Prototype:

```
int T32_TAPAccessShiftRaw ( T32_TAPACCESS_HANDLE  handle,
                            int                   numberofbits,
                            uint8_t              *pTMSBits,
                            uint8_t              *pTDIBits,
                            uint8_t              *pTDOBits,
                            int                   options );
```

Parameters:

`handle`	TAP access handle
`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 is used to send/receive arbitrary TDI/TMS/TDO patterns. 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 0 1 0 0 0 0 0 for TDI.

It is possible to pass a NULL pointer for any of the `pT??Bits` 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 both transferred the maximum pattern size is limited to 1/2 * (`T32_TAPACCESS_MAXBITS` - 64). If TDI or TMS are omitted, 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()
{
  uint8_t pTDI[1];
  uint8_t pTMS[1];
  uint8_t 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_DIRECTACCESS_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_DIRECTACCESS_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_DIRECTACCESS_HOLD , 2 , pTMS,
       0 , 0, SHIFTRAW_OPTION_NONE))
    goto error;

error:
  T32_DirectAccessRelease();
  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()
{
  uint8_t pTDI[1];
  uint8_t pTMS[1];
  uint8_t pTDO[1];
  int     err = 0;

  T32_BUNDLEDACCESS_HANDLE handle = T32_BunledAccessAlloc ();
  /*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_BundledAccessExecute(handle,T32_DIRECTACCESS_HOLD))
    goto error;
error:
  T32_DirectAccessRelease();
  T32_BundledAccessFree(handle);
  return err;
}
```

ICD User Signal API Functions

The user signal API function provides access to specific signals of the debug hardware.

Prototype:

```
int T32_DirectAccessUserSignal ( T32_TAPACCESS_HANDLE  handle,
                        int                   NumberOfAccesses,
                        const uint32_t *pOutAccesses,
                        uint32_t *pInAccesses);
```

Parameters:

handle	bundled access handle
NumberOfAccesses	amount of entries in pOutAccesses and pInAccesses
pOutAccesses	array containing user signal access commands
pInAccesses	array receiving the results of the user signal access commands

Returns:

0 for ok, otherwise Error value

The primary use of this function is to directly access the lines of the debug cable that are not handled by T32_TAPAccessDirect. The pOutAccesses buffer can also contain multiple commands. Any command-word consists of one word that is created by the signal name and command. The size of the return value is always identical with the command size.

For a direct access to the debug cable signals, commands can be generically generated. Basically the supported signals are:

System signals:

T32_DIRECTACCESS_USERSIGNAL_POWER

T32_DIRECTACCESS_USERSIGNAL_RESET

Additional confidential signals can be available in separate header files

The signals are ORed with the commands and passed in an array by parameter pOutAccesses. Possible commands are :

Signal Access Commands:

T32_DIRECTACCESS_USERSIGNAL_SET_ON	assert signal
T32_DIRECTACCESS_USERSIGNAL_SET_OFF	de-assert signal

T32_DIRECTACCESS_USERSIGNAL_GET	read current state of signal
T32_DIRECTACCESS_USERSIGNAL_WAS_TRIGGER	read latch of signal. some signal can have a latch to find pulses on the line in time.

The T32_DirectAccessUserSignal function support also the creation of asynchronous events on rising or falling edge of a signal. The events are checked by every call to TRACE32 and indicated by the return value of T32_USERSIGNALEVENT. Once that return value is received by calling any other function of the Direct Access API part the events must be retrieved by the command T32_DIRECTACCESS_USERSIGNAL_GET_NEXT_EVENT. In case an error was preempted by that mechanism the preempted error can be retrieved by T32_DIRECTACCESS_USERSIGNAL_GET_LAST_ERROR.

Signal Event Generator Commands:

T32_DIRECTACCESS_USERSIGNAL_ENABLE_EVENT_RISING	install event generator for event. poll a signal and generate an event when the signal changes to the asserted state
T32_DIRECTACCESS_USERSIGNAL_ENABLE_EVENT_FALLING	install event generator for event. poll a signal and generate an event when the signal changes to the de-asserted state
T32_DIRECTACCESS_USERSIGNAL_DISABLE_EVENT_RISING	remove event generator for event
T32_DIRECTACCESS_USERSIGNAL_DISABBLE_EVENT_FALLING	remove event generator for event
T32_DIRECTACCESS_USERSIGNAL_GET_EVENT_COUNT	retrieve the amount of queued events
T32_DIRECTACCESS_USERSIGNAL_GET_NEXT_EVENT	retrieve a queued event from the FIFO. The response of the command is the command that was used to install the event generator for that event or 0 in case no event is queued.
T32_DIRECTACCESS_USERSIGNAL_CLEAR_ALL_EVENTS	remove all collected events from the event FIFO.
T32_DIRECTACCESS_USERSIGNAL_CLEAR_EVENT	remove all collected events of a certain signal from the event FIFO.

T32_DIRECTACCESS_USERSIGNAL_DISABLE_AND_CLEAR_ALL_EVENTS	disable all event generators and remove all events from the FIFO.
T32_DIRECTACCESS_USERSIGNAL_POLL_EVENTS	execute all event generators to generate new events if the conditions are true in between a bundled call.
T32_DIRECTACCESS_USERSIGNAL_GET_LAST_ERROR	retrieve preempted error.

Example:

```
//reset and install event generators for Power and Reset signals
uint32_t commands[5] = {
    T32_DIRECTACCESS_USERSIGNAL_DISABLE_AND_CLEAR_ALL_EVENTS|
    T32_DIRECTACCESS_USERSIGNAL_ENABLE_EVENT_RISING|
    T32_DIRECTACCESS_USERSIGNAL_POWER,
    T32_DIRECTACCESS_USERSIGNAL_ENABLE_EVENT_FALLING|
    T32_DIRECTACCESS_USERSIGNAL_POWER,
    T32_DIRECTACCESS_USERSIGNAL_ENABLE_EVENT_RISING|
    T32_DIRECTACCESS_USERSIGNAL_RESET,
    T32_DIRECTACCESS_USERSIGNAL_ENABLE_EVENT_FALLING|
    T32_DIRECTACCESS_USERSIGNAL_RESET };

::T32_DirectAccessUserSignal(T32_DIRECTACCESS_HOLD, sizeof(commands)/4,
commands, NULL);
```

Example:

```
//routine to retrieve events
static int CheckForEvents(int result) {
    if (result == T32_USERSIGNALEVENT) {
        int err;
        int preempted_error;
        uint32_t out[100];
        uint32_t in[100];
        uint32_t count;
        out[0] = T32_DIRECTACCESS_USERSIGNAL_GET_LAST_ERROR;
        out[1] = T32_DIRECTACCESS_USERSIGNAL_GET_EVENT_COUNT;
        err = ::T32_DirectAccessUserSignal(T32_DIRECTACCESS_HOLD,
                                           2, out, in);
        if (err != 0 && err != T32_USERSIGNALEVENT)
            return err;
        preempted_error = in[0];
        count = in[1];
        printf("CheckForEvent: event count: %d\n", count);
        while (count > 0) {
            int next = (count > 100)?100:count;
            int ev;
            for (ev = 0; ev < next ; ++ev)
                out[ev] = T32_DIRECTACCESS_USERSIGNAL_GET_NEXT_EVENT;
            err = result = ::T32_DirectAccessUserSignal(
                             T32_DIRECTACCESS_HOLD, next, out, in);
            if (err != 0 && err != T32_USERSIGNALEVENT)
                return err;
            for (ev = 0; ev < next; ++ev)
                printf("CheckForEvent: event : command 0x%x signal
0x%x\n", in[ev] & 0xFFFF0000, in[ev] & 0x0000FFFF);
            count -= next;
        }
        return preempted_error;
    } else
        return result;
}
```

```
//do JTAG access and check for events
int res;
res = CheckForEvents(::T32_TAPAccessShiftRaw(T32_DIRECTACCESS_HOLD,
NULL, NULL, NULL, 100,
SHIFTRAW_OPTION_TMS_ZERO|GTL_JTAG_PROBE_SHIFT_RAW_TDI_ONE));
```

The DAP access functions and parameters are used to access the Arm DAP at low level by DAP scan calls and to provide an instance node for DAP dependent instances as AHB,APB,AXI busses or JTAG behind DAP by a JTAG-AP.

In case the DAP is JTAG based it requires to have the park state on Select-DR-Scan, e.g.:

```
T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
                        T32_DIRECTACCESS_INSTANCETYPE_TAP, 0,
                        T32_DIRECTACCESS_TAP_PARKSTATE_UINT32,
                        T32_ParamFromUint32(T32_TAPSTATE_SELECT_DR_SCAN))
```

Parameter: JTAG TAP Controller

- **Identifier**: T32_DIRECTACCESS_DAP_TAP_INSTANCE_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	Parking state
0xFFFFFFFF	No JTAG TAP instance selected. This makes only sense in case of Serial Wire Debug.
0..n	Used JTAG TAP instance.

- **Default:** 0xFFFFFFFF

Effect: Defines which JTAG TAP controller is used to access the Arm DAP.

Parameter: Serial Wire Debug TARGETSEL

- **Identifier**: T32_DIRECTACCESS_DAP_SWDP_TARGETSEL_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	Effect
0xFFFFFFFF	no TARGETSEL instruction is executed
0 <= p <= n	select SWD port p by TARGETSEL instruction

- **Default:** 0xFFFFFFFF

Effect: In case of Serial Wire Debug mode, the end point can be selected.

Parameter: Timeout

- **Identifier**: T32_DIRECTACCESS_DAP_TIMEOUT_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32
- **Values:** timeout of operations in milliseconds
- **Default:** 50 [ms]

Effect: Configure timeout of operations done by the scan function and higher level functions of busses or JTAG-AP that use the scan function finally.

Parameter: Option field for T32_DAPAccessScan

- **Identifier**: T32_DIRECTACCESS_DAP_SCAN_DAP_OPTION_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	Effect
T32_SCAN_DAP_OPTION_OUTBUFFER_IMMEDIATELY	`T32_DAPAccessScan` returns the output data immediately. In case of JTAG this requires extra shift.
T32_SCAN_DAP_OPTION_OUTBUFFER_DELAYED	T32_DAPAccessScan returns the output data by the next call of `T32_DAPAccessScan`.
T32_SCAN_DAP_OPTION_OUTBUFFER_DEPEND_TO_CONFIG	`T32_DAPAccessScan` returns the output matching to the physical interface. In case of JTAG the output data is delayed by one call of `T32_DAPAccessScan`. In case of Serial Wire Debug the output data is returned immediately.

- **Default:** T32_SCAN_DAP_OPTION_OUTBUFFER_IMMEDIATELY

Effect: Defines how the `T32_DAPAccessScan` functions works.

Prototype:

```
int T32_DAPAccessScan(T32_BUNDLEDACCESS_HANDLE handle,
                      unsigned int nDapInstance,
                      int nRegisterSet,
                      int nRW,
                      uint32_t nRegisterAddress,
                      uint32_t nDataIn,
                      uint32_t *pDataOut)
```

Parameters:

`handle`	TAP access handle
`nDapInstance`	DAP instance index
`nRegisterSet`	Register set used by the access. `T32_DAPACCESS_REGISTERSET_DP`: access to the DP registers `T32_DAPACCESS_REGISTERSET_AP`: access to the AP registers
`nRW`	Access type used by the access. `T32_DAPACCESS_RW_READ`: read access `T32_DAPACCESS_RW_READWRITE`: read and write access
`nRegisterAddress`	Register address.
`nDataIn`	Data written to the register.
`pDataOut`	Data read from the register. May be `NULL` if no return data is expected.

Returns:

0 for ok, otherwise Error value

Use this function to access an AP or DP register of a DAP.

Example:

```
T32_DAPAccessScan(/*Handle*/ T32_DIRECTACCESS_HOLD,
                /*nDapInstance*/ 0,
                /*nRegisterSet*/ T32_DAPACCESS_REGISTERSET_DP,
                /*nRW*/ T32_DAPACCESS_RW_READWRITE,
                /*nRegisterAddress*/ 0x1 /*CTRLSTAT*/,
                /*nDataIn*/ 0x54000020 /*Debug Reset Request*/,
                /*pDataOut*/ NULL);
```

Prototype:

```
int T32_DAPAccessInitSWD(T32_BUNDLEDACCESS_HANDLE Handle,
                         unsigned int nDapInstance);
```

Parameters:

`handle`	TAP access handle
`nDapInstance`	DAP instance index

Returns:

0 for ok, otherwise Error value

Use this function to initialize the Serial Wire Debug port before any other DAP access is done.

Example:

```
#define DAP_INSTANCE_INDEX 0

//Configure debug port
if (T32_Cmd("SYStem.JtagClock 1Mhz"))
    goto error;
if (T32_Cmd("SYStem.CONFIG.DEBUGPORTTYPE SWD"))
    goto error;

//Reset previous configuration
if (T32_DirectAccessResetAll(T32_DIRECTACCESS_HOLD))
    goto error;

//Setup TARGETSEL mechanism
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
T32_DIRECTACCESS_INSTANCETYPE_DAP, DAP_INSTANCE_INDEX,
T32_DIRECTACCESS_DAP_SWDP_TARGETSEL_UINT32,
T32_ParamFromUint32(0xFFFFFFFF /*no TARGETSEL used*/))
    goto error;
```

```
//Enable output of debug port driver
buffer[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
if (T32_TAPAccessDirect(T32_DIRECTACCESS_HOLD, 1, buffer, NULL))
    goto error;

//Initialize SWD port
if (T32_DAPAccessInitSWD(T32_DIRECTACCESS_HOLD, DAP_INSTANCE_INDEX))
    goto error;
printf("Init SWD Done\n");

error:
    //Release Direct Access API
    T32_DirectAccessRelease();
```

DAP Bus Access API Functions

The DAP bus access functions and parameters are used to access the AHB, APB or AXI busses behind a DAP.

Parameter: used DAP instance

- **Identifier**: T32_DIRECTACCESS_AHB_DAP_INSTANCE_UINT32,
  T32_DIRECTACCESS_APB_DAP_INSTANCE_UINT32,
  T32_DIRECTACCESS_AXI_DAP_INSTANCE_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	Parking state
0xFFFFFFFF	No DAP instance selected
0..n	Used DAP instance

- **Default:** 0xFFFFFFFF

Effect: The parameter must be set in order to configure which DAP is connected to the bus.

Parameter: DAP access port

- **Identifier**: T32_DIRECTACCESS_AHB_DAPACCESSPORT_UINT32,
  T32_DIRECTACCESS_APB_DAPACCESSPORT_UINT32,
  T32_DIRECTACCESS_AXI_DAPACCESSPORT_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Bus Type	Default
T32_DIRECTACCESS_INSTANCETYPE_AHB	0
T32_DIRECTACCESS_INSTANCETYPE_APB	1
T32_DIRECTACCESS_INSTANCETYPE_AXI	3

Effect: Set the used DAP access port for the bus.

Parameter: Bus endianness

- **Identifier**: T32_DIRECTACCESS_AHB_BIGENDIAN_UINT32,
  T32_DIRECTACCESS_APB_BIGENDIAN_UINT32,
  T32_DIRECTACCESS_AXI_BIGENDIAN_UINT32
- **Set**: Yes , **Get**: No
- **Type**: UINT32

Values	Endianness
0	Little
1	Big

- **Default:** 0

Effect: rearrange bytes of read/written of T32_DAPAPAccessReadWrite in order to work with a data buffer independent of the used access width.

Parameter:

- **Identifier**: T32_DIRECTACCESS_AHB_SYSPOWERUPREQ_UINT32,
  T32_DIRECTACCESS_APB_SYSPOWERUPREQ_UINT32,
  T32_DIRECTACCESS_AXI_SYSPOWERUPREQ_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	Effect
1	request power
0	do not request power

- **Default:** 1

Effect: Set the system power request control signals for every access by `T32_DAPAPAccessReadWrite`.

- **Identifier**: T32_DIRECTACCESS_AHB_CORTEXM_UINT32
- **Set**: Yes, **Get**: No
- **Type**: UINT32

Values	Special access
1	yes
0	no

- **Default:** 0

Effect: In case of an Cortex-M device the is only one AHB bus connected to the DAP. This AHB bus needs to be accessed in a special way by `T32_DAPAPAccessReadWrite`.

Prototype:

```
int T32_DAPAPAccessReadWrite(T32_BUNDLEDACCESS_HANDLE Handle,
                             int nAPType,
                             unsigned int nInstance,
                             int nRW,
                             uint64_t nAddress,
                             uint8_t * pData,
                             unsigned int nByteWidth,
                             unsigned int nByteSize,
                             int bNonIncrement,
                             uint32_t nHProtFlags);
```

Parameters:

`handle`	TAP access handle
`nAPType`	Bus Type. `T32_DIRECTACCESS_INSTANCETYPE_AHB`: AHB Bus `T32_DIRECTACCESS_INSTANCETYPE_APB`: APB Bus `T32_DIRECTACCESS_INSTANCETYPE_AXI`: AXI Bus
`nInstance`	Bus instance index.
`nRW`	Access Type. `T32_DAPAPACCESS_RW_READ`: read access `T32_DAPAPACCESS_RW_WRITE`: write access
`nAddress`	Memory address.
`pData`	Pointer to read/write data.
`nByteWidth`	Width of access. 1: 1 byte wide access 2: 2 byte wide access 4: 4 byte wide access 8: 8 byte wide access

`nByteSize`	Size of read or written data in bytes.
`bNonIncrement`	Address incremental control. 0: increment addresses 1: multiple read/write at one single address
`nHProtFlags`	HProt-Flags matching to the bus type.

Returns:

0 for ok, otherwise Error value

Use this function to access the memory on an APB, AHB or AXI bus.

Example:

```
//Read Arm Coresight ETM ID registers when ETM is located at
//JTAG -> DAP -> APB -> ADR:0x8011C000

//Configuration
#define TAP_INSTANCE_INDEX 0
#define DAP_INSTANCE_INDEX 0
#define APB_INSTANCE_INDEX 0

//Setup Debug Port
if (T32_Cmd("SYStem.JtagClock 1Mhz"))
    goto error;

//Reset previous configuration
if (T32_DirectAccessResetAll(T32_DIRECTACCESS_HOLD))
    goto error;

//Configure JTAG
//set park state to Select-DR-Scan
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
    T32_DIRECTACCESS_INSTANCETYPE_TAP, TAP_INSTANCE_INDEX,
    T32_DIRECTACCESS_TAP_PARKSTATE_UINT32,
    T32_ParamFromUint32(T32_TAPSTATE_SELECT_DR_SCAN))) goto error;

//Configure DAP
//set JTAG TAP instance
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
    T32_DIRECTACCESS_INSTANCETYPE_DAP, DAP_INSTANCE_INDEX,
    T32_DIRECTACCESS_DAP_TAP_INSTANCE_UINT32,
    T32_ParamFromUint32(TAP_INSTANCE_INDEX))) goto error;
```

```
//Configure APB
//set DAP instance
if (T32_DirectAccessSetInfo(T32_DIRECTACCESS_HOLD,
    T32_DIRECTACCESS_INSTANCETYPE_APB, APB_INSTANCE_INDEX,
    T32_DIRECTACCESS_AHB_DAP_INSTANCE_UINT32,
    T32_ParamFromUint32(DAP_INSTANCE_INDEX))) goto error;

//Enable output of debug port driver
buffer[0] = T32_TAPACCESS_nENOUT | T32_TAPACCESS_SET_0;
if (T32_TAPAccessDirect(T32_DIRECTACCESS_HOLD, 1, buffer, NULL))
    goto error;

//Reset JTAG
if (T32_TAPAccessJTAGResetWithTMS(T32_DIRECTACCESS_HOLD,
    TAP_INSTANCE_INDEX))
    goto error;


//Read IDs at APB bus
if (T32_DAPAPAccessReadWrite(T32_DIRECTACCESS_HOLD,
    /*Instance Type*/ T32_DIRECTACCESS_INSTANCETYPE_APB,
    /*Instance*/ APB_INSTANCE_INDEX,
    /*Read/Write*/ T32_DAPAPACCESS_RW_READ,
    /*Address*/ 0x8011CFF0,
    /*Data*/ buffer,
    /*Access Width*/ 4,
    /*Access Length*/ 4*4,
    /*NoneIncrement*/ 0,
    /*HProt*/ 0x0))
    goto error;
printf("ETM IDs : 0x%x 0x%x 0x%x 0x%x\n", buffer[0], buffer[4],
    buffer[8], buffer[12]);

error:
//Release Direct Access API
    T32_DirectAccessRelease();
```

Remote Lua API Functions

The TRACE32 Lua API allows the user to load and execute Lua scripts directly in the debugger. This feature can be used to accelerate execution of certain debug commands by avoiding the interaction between the T32 host SW and the debug driver. A Lua interpreter is built into the debugger, supporting the complete Lua language. In addition, Lauterbach has extended the Lua language with a set of T32 specific libraries, which allow the user to, for example, use the JTAG shift interface directly from the Lua script. Please refer to **“TRACE32 Lua Library”** (lua_library.pdf) for more details.

This section describes functions to load/execute Lua scripts using the remote API. Refer to the **Lua** command group for using the Lua in the command line.

T32_ExecuteLua

Loads/executes a Lua script in single access mode. If the Lua script involves TAP access, the default settings are used.

Prototype:

```
int T32_ExecuteLua(const char* filename,
                   int mode,
                   const uint8_t* inputBuf,
                   int inputBufLen,
                   uint8_t* ouputBuf,
                   int outputBufLen);
```

Parameters:

`filename`	Path to the Lua script to be loaded.
`mode`	Mode of execution: • bit 0: - 0: do not execute the Lua script. - 1: execute the Lua script. • bit 1: - 0: do not force to replace the Lua script if exists aleady. - 1: force to replace the Lua script if exsits already. • bit 2: - 0: do not load the script to the target. - 1: load the script to the target. • bit [31:3]: unused. • Example: - 0x1: no load, only execute the script if exist already. - 0x4: only load the script. - 0x7: load and execute the script.
`inputBuf`	Pointer to input buffer. The data in the input buffer will be send to the debugger together with the script itself. Inside the Lua script, functions from the **"TRACE32 Lua Library"** (lua_library.pdf) can be used to retrieve data from the input buffer.
`inputBufLen`	Length of input buffer, must be smaller than 0x1000 bytes. If you have data size more than this, define it directly in your Lua script.
`ouputBuf`	Pointer to output buffer. Inside the Lua script, functions from the **"TRACE32 Lua Library"** (lua_library.pdf) can be used to write data to the output buffer. The data written by the Lua script is automatically transferred back to the user.
`outputBufLen`	Length of output buffer, must be smaller than 0x1000 bytes.

Example:

```
;load the script jtag.lua, no input/output buffer specified
T32_ExecuteLua("C:\\lua\\jtag.lua",0x4,NULL,0,NULL,0);

;overwrite the script
T32_ExecuteLua("C:\\lua\\jtag.lua",0x6,input,8,output,0x220);

;execute the script (it is loaded already)
T32_ExecuteLua("C:\\lua\\jtag.lua",0x1,input,8,output,0x220);

;do everything in one shot
T32_ExecuteLua("C:\\lua\\jtag.lua",0x7,input,8,output,0x220);
```

T32_DirectAccessExecuteLua

Loads/executes a Lua script in bundle mode. If the Lua script involves TAP access, it shared the configuration previously done for the bundle.

Prototype:

```
int T32_DirectAccessExecuteLua(T32_BUNDLEDACCESS_HANDLE Handle,
                               const char* filename,
                               int mode,
                               const uint8_t* inputBuf,
                               int inputBufLen,
                               uint8_t* ouputBuf,
                               int outputBufLen);
```

Parameters:

`handle`	TAP access handle
`filename`	Path to the Lua script to be loaded.
`mode`	Mode of execution: • bit 0: - 0: do not execute the Lua script. - 1: execute the Lua script. • bit 1: - 0: do not force to replace the Lua script if exists aleady. - 1: force to replace the Lua script if exsits already. • bit 2: - 0: do not load the script to the target. - 1: load the script to the target. • bit [31:3]: unused. • Example: - 0x1: no load, only execute the script if exist already. - 0x4: only load the script. - 0x7: load and execute the script.
`inputBuf`	Pointer to input buffer. The data in the input buffer will be send to the debugger together with the script itself. Inside the Lua script, functions from the **"TRACE32 Lua Library"** (lua_library.pdf) can be used to retrieve data from the input buffer.
`inputBufLen`	Length of input buffer, must be smaller than 0x1000 bytes. If you have data size more than this, define it directly in your Lua script.
`ouputBuf`	Pointer to output buffer. Inside the Lua script, functions from the **"TRACE32 Lua Library"** (lua_library.pdf) can be used to write data to the output buffer. The data written by the Lua script is automatically transferred back to the user.
`outputBufLen`	Length of output buffer, must be smaller than 0x1000 bytes.

Example:

```
; allocate a handle
handle1 = T32_BundledAccessAlloc ();
; do TAP configurations as introduced above
; ...
; ...

; now use the Lua feature
T32_DirectAccessExecuteLua(handle1,"C:\\lua\\jtag.lua",0x7,input,8,outpu
t,0x22);
; execute the bundle
T32_TAPAccessExecute (handle1, T32_DIRECTACCESS_RELEASE);
T32_BundledAccessFree (handle1);
```

API Object Handling

This chapter describes the data objects used by the object oriented API functions:

- **Buffer Object**
- **Address Object**
- **Bundle Object**
- **Register Object**
- **RegisterSet Object**
- **Breakpoint Object**
- **Symbol Object**

The object oriented API follows a specific naming convention shown in the following table.

Object types	T32_<*objtype*>Obj
Object handle types	T32_<*objtype*>Handle
Allocating objects	T32_Request<*objtype*>Obj T32_Request<*objtype*>Obj<*initial*>
Reallocating objects	T32_Resize<*objtype*>
Freeing objects	T32_Release<*objtype*>Obj T32_ReleaseAllObjects
Getting object attributes	T32_Get<*objtype*>Obj<*attribute*>
Setting object attributes	T32_Set<*objtype*>Obj<*attribute*>
Copying existing object	T32_Copy<*objtype*>Obj
Copying from/into objects	T32_Copy<*what*>From/To<*objtype*>Obj
Reading from target	T32_Read<*objtype*>Obj
Reading by signifier	T32_Read<*objtype*>ObjBy<*signifier*>
Writing to target	T32_Write<*objtype*>Obj
Getting info from TRACE32	T32_Query<*objtype*>Obj
Sending info to TRACE32	T32_Send<*objtype*>Obj

Buffer Object

A buffer object holds a memory buffer allocated in the API. See an usage example in **T32_ReadMemoryObj**.

Object handle:

```
T32_BufferHandle myBufferHandle;
```

Declares a buffer handle. No buffer object is yet created.

Object functions:

```
int T32_RequestBufferObj (T32_BufferHandle *pHandle,
    const int initial_size);
```

Creates (allocates) a buffer object.
`pHandle` points to the declared buffer handle.
`initial_size` specifies the number of bytes in initially allocate. It may be zero. The buffer object is resized if its size is not sufficient for its usage.

```
int T32_ReleaseBufferObj (T32_BufferHandle *pHandle);
```

Releases (frees) a buffer object.
`pHandle` points to the buffer handle to release. It’s contents is no longer valid then.

```
int T32_ResizeBufferObj (T32_BufferHandle handle, const int size);
```

Resizes the allocated storage of the buffer object.
`handle` specifies the buffer object to resize.
`size` specifies the new size of the buffer object.

```
int T32_CopyDataFromBufferObj (uint8_t *localbuffer,
    int lbsize, T32_BufferHandle handle);
```

Copies the data of a buffer object into a byte array.
`localbuffer` points to the byte array where the data is copied to.
`lbsize` specifies the size of the byte array.
`handle` specifies the buffer object to copy the data from.

```
int T32_CopyDataToBufferObj (T32_BufferHandle handle,
    int size, uint8_t *localbuffer);
```

Copies the data of a byte array into the buffer object.
`handle` specifies the buffer object to copy the data to.
`size` is the number of bytes to copy.
`localbuffer` points to a byte array where to copy the data from.

```
int T32_GetBufferObjStoragePointer (
    uint8_t** ppointer, T32_BufferHandle handle);
```

Get a byte array pointer that points to the buffer object internal storage.
Note: this function exposes API internal data is not guaranteed to be compatible with future API releases!
`ppointer` will get the pointer to the internal data storage.
`handle` specifies the buffer object.

Address Object

An address object holds the attributes of a target address, that are:

- Target address
- Target access class
- Memory access width
- Core ID
- Space ID
- Attributes
- Size of MAU (minimum addressable unit)

See an usage example in **T32_ReadMemoryObj**.

Object handle:

```
T32_AddressHandle myAddressHandle;
```

Declares an address handle. No address object is yet created.

Object functions:

```
int T32_RequestAddressObj (T32_AddressHandle *pHandle,
    const T32_AddressObjType addrType);
```

Creates (allocates) a target address object.
Note: only for advanced usage. Please use one of the dedicated requests below.
`pHandle` points to the declared address handle.
`addrType` specifies the type of address object to be created.

```
int T32_RequestAddressObjA32 (T32_AddressHandle *pHandle,
    const uint32_t address);
```

Creates (allocates) a target address object with a 32bit address.
`pHandle` points to the declared address handle.
`address` specifies an initial target address for this address object.

```
int T32_RequestAddressObjA64 (T32_AddressHandle *pHandle,
    const uint64_t address);
```

Creates (allocates) a target address object with a 64bit address.
`pHandle` points to the declared address handle.
`address` specifies an initial target address for this address object.

```
int T32_ReleaseAddressObj (T32_AddressHandle *pHandle);
```

Releases (frees) an address object.
`pHandle` points to the address handle to release. It's contents is no longer valid then.

```
int T32_SetAddressObjAddr32 (T32_AddressHandle handle,
    uint32_t address);
```

Set the 32bit target address of an address object.
`handle` specifies the address object.
`address` specifies the new target address.
The address is byte- (octet-) oriented, if not explicitly changed with `T32_SetAddressObjSizeOfMau()`.
See also **Conventions for Target Memory Access**.

```
int T32_GetAddressObjAddr32 (T32_AddressHandle handle,
    uint32_t *pAddress);
```

Get the 32bit target address of an address object.
`handle` specifies the address object.
`pAddress` points to the variable getting the target address.
The address is byte- (octet-) oriented, if not explicitly changed with `T32_SetAddressObjSizeOfMau()`.
See also **Conventions for Target Memory Access**.

```
int T32_SetAddressObjAddr64 (T32_AddressHandle handle,
    uint64_t address);
```

Set the 64bit target address of an address object.
`handle` specifies the address object.
`address` specifies the new target address.
The address is byte- (octet-) oriented, if not explicitly changed with `T32_SetAddressObjSizeOfMau()`.
See also **Conventions for Target Memory Access**.

```
int T32_GetAddressObjAddr64 (T32_AddressHandle handle,
    uint64_t *pAddress);
```

Get the 64bit target address of an address object.
`handle` specifies the address object.
`pAddress` points to the variable getting the target address.
The address is byte- (octet-) oriented, if not explicitly changed with `T32_SetAddressObjSizeOfMau()`.
See also **Conventions for Target Memory Access**.

```
int T32_SetAddressObjAccessString (T32_AddressHandle handle,
    const char* accessString);
```

Set the access class of an address object.
`handle` specifies the address object.
`accessString` points a null-terminated string containing the access class specifier as listed in the **Processor Architecture Manuals** **without the colon**.

```
int T32_GetAddressObjAccessString (T32_AddressHandle handle,
    char* accessString, uint8_t maxlen);
```

Get the access class of an address object.
`handle` specifies the address object.
`accessString` points to a character array allocated by the user.
`maxlen` specifies the maximum length of the character array.
The character array will receive a string containing an access class specifier as listed in the **Processor Architecture Manuals** **without the colon**.

```
int T32_SetAddressObjWidth (T32_AddressHandle handle, uint16_t width);
```

Set the access width of an address object.
`handle` specifies the address object.
`width` specifies the access width in bytes, with which the debugger tries to access this address.

```
int T32_SetAddressObjCore (T32_AddressHandle handle, uint16_t core);
```

Set the core ID of an address object.
`handle` specifies the address object.
`core` specifies the core ID in multicore systems, with which the debugger tries to access this address.

```
int T32_SetAddressObjSpaceId (T32_AddressHandle handle,
    uint32_t spaceid);
```

Set the space ID of an address object.
`handle` specifies the address object.
`spaceid` specifies the space ID in MMU spaced systems, with which the debugger tries to access this address.

```
int T32_SetAddressObjAttr (T32_AddressHandle handle,
    uint32_t attributes);
```

Set attributes of an address object.
`handle` specifies the address object.
`attributes` are or'ed bit patterns that specify special access attributes:

T32_ADDROBJATTR_EACCESS	Use run-time memory access (Access class E:)
T32_ADDROBJATTR_VERIFY	Verify if write access succeeded (by reading value back)
T32_ADDROBJATTR_RTSTRIGGERACK	Acknowledge RTS trigger to resume RTS processing. (use after receiving notification T32_E_RTSTRIGGER)
T32_ADDROBJATTR_GO	Start core after this access.
T32_ADDROBJATTR_BREAK	Halt core after this access.

`T32ADDROBJATTR_EACCESS:` read dual ported from this address ("emulation access").
`T32ADDROBJATTR_VERIFY`; verify after write to this address

```
int T32_SetAddressObjSizeOfMau (T32_AddressHandle handle,
    T32_SizeOfMauType SizeOfMau);
```

Set the MAU size of an address object.
`handle` specifies the address object.
`SizeOfMau` contains the MAU size to set.

When a MAU (minimum addressable unit) size is set, all read and write operations using this address object will calculate the address according to the MAU size. E.g. if you set a 16bit MAU size, each address refers to a 16bit unit. Address "1" will then point to the 16th bit in memory.

Valid values as MAU size are:

T32_SIZEOFMAU_NOTSET	MAU is not set; default is 8bit
T32_SIZEOFMAU_TARGET	MAU of target is used
T32_SIZEOFMAU_8BIT	MAU is 8bit
T32_SIZEOFMAU_12BIT	MAU is 12bit
T32_SIZEOFMAU_16BIT	MAU is 16bit
T32_SIZEOFMAU_24BIT	MAU is 24bit

T32_SIZEOFMAU_32BIT	MAU is 32bit
T32_SIZEOFMAU_64BIT	MAU is 64bit
T32_SIZEOFMAU_128BIT	MAU is 128bit
T32_SIZEOFMAU_256BIT	MAU is 256bit

```
int T32_GetAddressObjSizeOfMau (T32_AddressHandle handle,
    T32_SizeOfMauType* pSizeOfMau);
```

Get the MAU size of an address object.
`handle` specifies the address object.
`pSizeOfMau` points to the variable getting the MAU size.

See description of **T32_SetAddressObjSizeOfMau**.

```
int T32_GetAddressObjTargetSizeOfMau (T32_AddressHandle handle,
    T32_SizeOfMauType* pTargetSizeOfMau);
```

Get the target MAU size of an address object.
`handle` specifies the address object.
`pTargetSizeOfMau` points to the variable getting the MAU size.

The target MAU (minimum addressable uint) size must be queried previously with the function **T32_QueryAddressObjTargetSizeOfMau**.
For valid values as MAU size see **T32_SetAddressObjSizeOfMau**.

NOTE:	See the section “**Conventions for Target Memory Access**” for important conventions regarding the byte addresses and accesses.

Memory Bundle Object

A memory bundle object holds a list of memory buffers with associated addresses allocated in the API. See a usage example in **T32_TransferMemoryBundleObj**.

Object handle:

```
T32_MemoryBundleHandle myBundleHandle;
```

Declares a bundle handle. No bundle object is yet created.

Object functions:

```
int T32_RequestMemoryBundleObj (T32_MemoryBundleHandle *pHandle,
    const int initial_size);
```

Creates (allocates) a bundle object.
`pHandle` points to the declared bundle handle.
`initial_size` specifies the number of buffer objects initially allocated. It may be zero. The bundle object is resized if its size is not sufficient for its usage.

```
int T32_ReleaseMemoryBundleObj (T32_MemoryBundleHandle *pHandle);
```

Releases (frees) a bundle object.
`pHandle` points to the bundle handle to release. It’s contents is no longer valid thereafter.

```
int T32_AddToBundleObjAddrLength (T32_MemoryBundleHandle bundleHandle,
    const T32_AddressHandle addrHandle, const T32_Length length);
```

Adds an ‘empty’ buffer to a bundle object.
`bundleHandle` is the handle to the memory bundle object to add to.
`addrHandle` specifies the address of the buffer.
`length` specifies the size in byte of the buffer.
This function is for adding a buffer to be used for reading memory.

```
int T32_AddToBundleObjAddrLengthByteArray (
    T32_MemoryBundleHandle bundleHandle,
    const T32_AddressHandle addrHandle, const T32_Length length,
    uint8_t *localbuffer);
```

Adds a ‘filled’ buffer to a bundle object.
`bundleHandle` is the handle to the memory bundle object to add to.
`addrHandle` specifies the address of the buffer.

`length` specifies the size in byte of the buffer.
`localbuffer` points to a byte array where to copy the data from to fill the bundle buffer.
This function is for adding a buffer to be used for writing memory.

```
int T32_GetBundleObjSize (T32_MemoryBundleHandle bundleHandle,
    T32_Size *size);
```

Gets the number of buffers in a memory bundle object.
`bundleHandle` is the handle to the memory bundle to get the size of.
`size` points to the bundle size after the call.

```
int T32_GetBundleObjSyncStatusByIndex (
    T32_MemoryBundleHandle bundleHandle,
    T32_BufferSynchStatus *pSyncStatus, T32_Index index);
```

Gets the status of a bundle buffer.
`bundleHandle` is the handle to the memory bundle containing the buffer to get the status of.
`pSyncStatus` points to the status of the buffer after the call. See below.
`index` is the index of the buffer in the bundle to get the status of. The index must be less than the bundle size.

`T32_BufferSynchStatus` can have one of the following values:

```
T32_BUFFER_NOTSYNCHED,   // buffer not synchronized with target
T32_BUFFER_READ,         // buffer was read from target
T32_BUFFER_WRITTEN,      // buffer war written to target
T32_BUFFER_ERROR         // error while transferring this buffer
```

```
int T32_CopyDataFromBundleObjByIndex (uint8_t* localbuffer, int lbsize,
    T32_MemoryBundleHandle bundleHandle, T32_Index index);
```

Copies the contents of a bundle buffer to a local buffer.
`localbuffer` points to a byte array where to copy the buffer data to.
`lbsize` is the number of bytes to copy.
`bundleHandle` is the handle to the memory bundle containing the buffer to copy from.
`index` is the index of the buffer in the bundle to copy from. The index must be less than the bundle size.

Register Object

A register object holds the attributes of a target register, that are:

- Register name
- Register ID
- Value
- Size
- Core ID

See an usage example in **T32_ReadRegisterObj**.

Object handle:

```
T32_RegisterHandle myRegisterHandle;
```

Declares a register handle. No register object is yet created.

Object functions:

```
int T32_RequestRegisterObj (T32_RegisterHandle *pHandle,
    const T32_RegisterObjType regType);
```

Creates (allocates) a register object.
Note: only for advanced usage. Please use one of the dedicated requests below.
`pHandle` points to the declared register handle.
`regType` specifies the type of register object to be created.

```
int T32_RequestRegisterObjR32 (T32_RegisterHandle *pHandle);
```

Creates (allocates) a register object for a 32bit register.
`pHandle` points to the declared register handle.

```
int T32_RequestRegisterObjR64 (T32_RegisterHandle *pHandle);
```

Creates (allocates) a register object for a 64bit register.
`pHandle` points to the declared register handle.

```
int T32_RequestRegisterObjR128 (T32_RegisterHandle *pHandle);
```

Creates (allocates) a register object for an 128bit register.
`pHandle` points to the declared register handle.

```
int T32_RequestRegisterObjR256 (T32_RegisterHandle *pHandle);
```

Creates (allocates) a register object for an 256bit register.
`pHandle` points to the declared register handle.

```
int T32_RequestRegisterObjR512 (T32_RegisterHandle *pHandle);
```

Creates (allocates) a register object for an 512bit register.
`pHandle` points to the declared register handle.

```
int T32_RequestRegisterObjR32Name (T32_RegisterHandle *pHandle,
    const char* regName);
```

Creates (allocates) a register object for a 32bit register.
`pHandle` points to the declared register handle.
`regName` specifies an initial register name for this register object.

```
int T32_RequestRegisterObjR64Name (T32_RegisterHandle *pHandle,
    const char* regName);
```

Creates (allocates) a register object for a 64bit register.
`pHandle` points to the declared register handle.
`regName` specifies an initial register name for this register object.

```
int T32_RequestRegisterObjR128Name (T32_RegisterHandle *pHandle,
    const char* regName);
```

Creates (allocates) a register object for an 128bit register.
`pHandle` points to the declared register handle.
`regName` specifies an initial register name for this register object.

```
int T32_RequestRegisterObjR256Name (T32_RegisterHandle *pHandle,
    const char* regName);
```

Creates (allocates) a register object for an 256bit register.
`pHandle` points to the declared register handle.
`regName` specifies an initial register name for this register object.

```
int T32_RequestRegisterObjR512Name (T32_RegisterHandle *pHandle,
    const char* regName);
```

Creates (allocates) a register object for an 512bit register.
`pHandle` points to the declared register handle.
`regName` specifies an initial register name for this register object.

```
int T32_RequestRegisterObjR32Id (T32_RegisterHandle *pHandle,
    uint32_t regId);
```

Creates (allocates) a register object for a 32bit register.
`pHandle` points to the declared register handle.
`regId` specifies an initial register ID for this register object. Contact Lauterbach if you need a mapping of the register IDs for your CPU.

```
int T32_RequestRegisterObjR64Id (T32_RegisterHandle *pHandle,
    uint32_t regId);
```

Creates (allocates) a register object for a 64bit register.
`pHandle` points to the declared register handle.
`regId` specifies an initial register ID for this register object. Contact Lauterbach if you need a mapping of the register IDs for your CPU.

```
int T32_ReleaseRegisterObj (T32_RegisterHandle *pHandle);
```

Releases (frees) a register object.
`pHandle` points to the register handle to release. It’s contents is no longer valid then.

```
int T32_SetRegisterObjName (T32_RegisterHandle handle,
    const char* regName);
```

Set the register name of a register object.
`handle` specifies the register object.
`regName` specifies the new register name.

```
int T32_GetRegisterObjName (T32_RegisterHandle handle,
    char* regName, uint8_t maxlen);
```

Get the register name of a register object.
`handle` specifies the register object.
`regName` points to a character array allocated by the user.
`maxlen` specifies the maximum length of the character array.

```
int T32_SetRegisterObjId (T32_RegisterHandle handle,
    uint32_t regId);
```

Set the register ID of a register object.
`handle` specifies the register object.
`regId` specifies the new register ID.

```
int T32_GetRegisterObjId (T32_RegisterHandle handle,
    uint32_t *pRegId);
```

Get the register ID of a register object.
`handle` specifies the register object.
`pRegId` points to the variable getting the register ID.

```
int T32_SetRegisterObjValue32 (T32_RegisterHandle handle,
    uint32_t value);
```

Set the 32bit value of a register object (not the register on the target).
`handle` specifies the register object.
`value` specifies the new register object value.

```
int T32_GetRegisterObjValue32 (T32_RegisterHandle handle,
    uint32_t *pValue);
```

Get the 32bit value of a register object (not the register on the target).
`handle` specifies the register object.
`pValue` points to the variable getting the register object value.

```
int T32_SetRegisterObjValue64 (T32_RegisterHandle handle,
    uint64_t value);
```

Set the 64bit value of a register object (not the register on the target).
`handle` specifies the register object.
`value` specifies the new register object value.

```
int T32_GetRegisterObjValue64 (T32_RegisterHandle handle,
    uint64_t *pValue);
```

Get the 64bit value of a register object (not the register on the target).
`handle` specifies the register object.
`pValue` points to the variable getting the register object value.

```
int T32_SetRegisterObjValueArray (T32_RegisterHandle handle,
    uint8_t *pArray, uint8_t maxlen);
```

Set the value of a register object (not the register on the target) by a byte array. Array element 0 is the least significant byte, the last byte in the array is the most significant byte.
`handle` specifies the register object.
`pArray` specifies the array holding the new register value as a byte array.
`maxlen` specifies the length of the byte array.

```
int T32_GetRegisterObjValueArray (T32_RegisterHandle handle,
    uint8_t *pArray, uint8_t maxlen);
```

Write the value of a register object (not the register on the target) into a byte array. Array element 0 gets the least significant byte, the last byte in the array gets the most significant byte.
`handle` specifies the register object.
`pArray` specifies the array receiving the new register value as a byte array.
`maxlen` specifies the length of the byte array.

Example:

```
int i;
uint8_t regValue[16];
T32_RegisterHandle handle;
T32_RequestRegisterObjR128Name(&handle, "XMM0");
T32_ReadRegisterObj(myRegisterHandle);
T32_GetRegisterObjValueArray(handle, regValue, 16);
for (i = 0; i < 16; i++)
    printf ("%02x", regValue[15-i]);
T32_ReleaseRegisterObj(&handle);
```

```
int T32_SetRegisterObjCore (T32_RegisterHandle handle, uint16_t core);
```

Set the core ID of a register object.
`handle` specifies the register object.
`core` specifies the core ID in multicore systems, with which the debugger tries to access this register.

RegisterSet Object

A register set object is a container of several **register objects** and holds:

- number of registers in set
- all the registers in set

See an usage example in **T32_ReadRegisterSetObj**.

Object handle:

```
T32_RegisterSetHandle myRegisterSetHandle;
```

Declares a register set handle. No register set object is yet created.

Object functions:

```
int T32_RequestRegisterSetObj (T32_RegisterSetHandle *pHandle,
    int numRegisters, RegisterObjType regType);
```

Creates (allocates) a register set object.
Note: only for advanced usage. Please use one of the dedicated requests below.
`pHandle` points to the declared register handle.
`numRegisters` specifies the initial number of registers in set.
`regType` specifies the type of the initial registers.

```
int T32_RequestRegisterSetObjR32 (T32_RegisterSetHandle *pHandle,
    int numRegisters);
```

Creates (allocates) a register set object holding 32bit registers.
`pHandle` points to the declared register handle.
`numRegisters` specifies the initial number of registers in set.

```
int T32_RequestRegisterSetObjR64 (T32_RegisterSetHandle *pHandle,
    int numRegisters);
```

Creates (allocates) a register set object holding 64bit registers.
`pHandle` points to the declared register handle.
`numRegisters` specifies the initial number of registers in set.

```
int T32_ReleaseRegisterSetObj (T32_RegisterSetHandle *pHandle);
```

Releases (frees) a register set object and all its held registers.
`pHandle` points to the register set handle to release. It’s contents is no longer valid then.

```
int T32_SetRegisterSetObjNames (T32_RegisterSetHandle handle,
    const char** names, int numNames);
```

Set the register names of the registers within a register set object.
`handle` specifies the register set object.
`names` points to a string array holding the register names to set
`numNames` specifies the number of names to set.

```
int T32_SetRegisterSetObjValues32 (T32_RegisterSetHandle handle,
    const uint32_t *values, int numValues);
```

Set the 32bit values of the registers within a register set object (not the registers on the target).
`handle` specifies the register set object.
`values` points to an 32bit integer array holding the values to set.
`numValues` specifies the number of values to set.

```
int T32_GetRegisterSetObjValues32 (T32_RegisterHandle handle,
    uint32_t *values, int numValues);
```

Get the 32bit values of the registers within a register set object (not the register on the target).
`handle` specifies the register object.
`values` points to an 32bit integer array getting the values.
`numValues` specifies the number of values to get.

Breakpoint Object

A breakpoint object holds the attributes of a breakpoint, that are:

- **breakpoint address** (range)
- **breakpoint type** (program, read, write)
- **breakpoint implementation** (soft, onchip, marker, “auto”)
- **breakpoint action** (stop, spot, alpha, beta, charlie, delta, echo)
- **enabled status**
- **optional core**

See an usage example in **T32_WriteBreakpointObj**.

Object handle:

```
T32_BreakpointHandle myBpHandle;
```

Declares a breakpoint handle. No breakpoint object is yet created.

Object functions:

```
int T32_RequestBreakpointObj (T32_BreakpointHandle *pHandle);
```

Creates (allocates) a breakpoint object.
`pHandle` points to the declared breakpoint handle.

```
int T32_RequestBreakpointObjAddr (T32_BreakpointHandle *pHandle,
    const T32AddressHandle addrHandle);
```

Creates (allocates) a breakpoint object.
`pHandle` points to the declared breakpoint handle.
`addrHandle` specifies an initial address object for this breakpoint object.

```
int T32_ReleaseBreakpointObj (T32_BreakpointHandle *pHandle);
```

Releases (frees) a breakpoint object.
`pHandle` points to the breakpoint handle to release. It’s contents is no longer valid then.

Breakpoint Address

```
int T32_SetBreakpointObjAddress (T32_BreakpointHandle handle,
    T32_AddressHandle addrHandle);
```

Set the address of a breakpoint object.
`handle` specifies the breakpoint object.
`addrHandle` specifies the new address.

```
int T32_GetBreakpointObjAddress (T32_BreakpointHandle handle,
    T32_AddressHandle* pAddrHandle);
```

Get the address of a breakpoint object.
`handle` specifies the breakpoint object.
`pAddrHandle` points to the address handle receiving the breakpoint address.

Breakpoint Type

```
int T32_SetBreakpointObjType (T32_BreakpointHandle handle,
    uint32_t type);
```

Set the breakpoint type of a breakpoint object.
`handle` specifies the breakpoint object.
`type` specifies the new breakpoint type by one of these constants:

T32_BP_TYPE_PROGRAM	Program breakpoint
T32_BP_TYPE_READ	Read access breakpoint
T32_BP_TYPE_WRITE	Write access breakpoint
T32_BP_TYPE_RW	Read/Write access breakpoint

```
int T32_GetBreakpointObjType (T32_BreakpointHandle handle,
    uint32_t* pType);
```

Get the breakpoint type of a breakpoint object.
`handle` specifies the breakpoint object.
`pType` points to the variable receiving the breakpoint type as mentioned above.

```
int T32_SetBreakpointObjImpl (T32_BreakpointHandle handle,
    uint32_t impl);
```

Set the breakpoint implementation of a breakpoint object.
`handle` specifies the breakpoint object.
`impl` specifies the new breakpoint implementation by one of these constants:

T32_BP_IMPL_SOFT	Software breakpoint
T32_BP_IMPL_ONCHIP	Onchip breakpoint

```
int T32_GetBreakpointObjImpl (T32_BreakpointHandle handle,
    uint32_t* pImpl);
```

Get the breakpoint implementation of a breakpoint object.
`handle` specifies the breakpoint object.
`pImpl` points to the variable receiving the breakpoint implementation by one of these constants:

T32_BP_IMPL_AUTO	Automatic breakpoint (not active)
T32_BP_IMPL_SOFT	Software breakpoint
T32_BP_IMPL_ONCHIP	Onchip breakpoint
T32_BP_IMPL_MARK	Marker

If the breakpoint is active (not disabled), a breakpoint read operation will return the actual used implementation. If the breakpoint is disabled, a breakpoint read operation will return the implementation that was specified when setting the breakpoint.

```
int T32_SetBreakpointObjAction (T32_BreakpointHandle handle,
    uint32_t act);
```

Set the breakpoint action of a breakpoint object.
`handle` specifies the breakpoint object.
`act` specifies the new breakpoint action by one of these constants:

T32_BP_ACTION_STOP	Stop breakpoint
T32_BP_ACTION_SPOT	Spot breakpoint
T32_BP_ACTION_ALPHA	Alpha breakpoint
T32_BP_ACTION_BETA	Beta breakpoint
T32_BP_ACTION_CHARLIE	Charlie breakpoint
T32_BP_ACTION_DELTA	Delta breakpoint
T32_BP_ACTION_ECHO	Echo breakpoint

Alpha, Beta, Charlie, Delta and Echo are only possible for Stop breakpoints.

```
int T32_GetBreakpointObjAction (T32_BreakpointHandle handle,
    uint32_t* pAct);
```

Get the breakpoint action of a breakpoint object.
`handle` specifies the breakpoint object.
`pAct` points to the variable receiving the breakpoint action.

Enabled Status

```
int T32_SetBreakpointObjEnable (T32_BreakpointHandle handle,
    uint8_t enable);
```

Enable or disable a breakpoint. The default of a breakpoint object is “enabled”.
`handle` specifies the breakpoint object.
`enable`: if set to 0, the breakpoint is disabled, else enabled.

NOTE:	This function sets only the attribute in the object, without any setting in the debugger. To enable/disable breakpoints in the debugger, use a subsequent **T32_WriteBreakpointObj**.

```
int T32_GetBreakpointObjEnable (T32_BreakpointHandle handle,
    uint8_t* pEnable);
```

Get the enabled status of a breakpoint object.
`handle` specifies the breakpoint object.
`pEnable` points to the variable receiving the enabled status. If set to 0, the breakpoint is disabled, else enabled.

NOTE:	This function reads only the attribute in the object, without querying the debugger. To read the enabled status of a breakpoint in the debugger, use a preceding **T32_ReadBreakpointObj**.

Optional Core

```
int T32_SetBreakpointObjCore (T32_BreakpointHandle bpHandle,
    uint16_t core);
```

Specify the core where you want to set the breakpoint. The breakpoint action, such as stop, takes effect only if the program is executed on the specified core.
`handle` specifies the breakpoint object.
`core`: core to be set as core attribute of the breakpoint handle

```
int T32_GetBreakpointObjCore (T32_BreakpointHandle bpHandle,
    uint16_t* pCore);
```

Get the core attribute of a breakpoint object.
`handle` specifies the breakpoint object.
`pCore` points to the variable receiving the breakpoint core.

Symbol Object

A symbol object holds the attributes of a target application symbol, that are:

- symbol name
- symbol path (\\program\module\symbol)
- address
- size

See an usage example in **T32_QuerySymbolObj**.

Object handle:

```
T32_SymbolHandle mySymbolHandle;
```

Declares a symbol handle. No symbol object is yet created.

Object functions:

```
int T32_RequestSymbolObj (T32_SymbolHandle *pHandle);
```

Creates (allocates) a symbol object.
`pHandle` points to the declared symbol handle.

```
int T32_RequestSymbolObjName (T32_SymbolHandle *pHandle,
    const char* symName);
```

Creates (allocates) a symbol object.
`pHandle` points to the declared symbol handle.
`symName` specifies an initial symbol name for this symbol object.

```
int T32_RequestSymbolObjAddr (T32_SymbolHandle *pHandle,
    const T32AddressHandle addrHandle);
```

Creates (allocates) a symbol object.
`pHandle` points to the declared symbol handle.
`addrHandle` specifies an initial address object for this symbol object.

```
int T32_ReleaseSymbolObj (T32_SymbolHandle *pHandle);
```

Releases (frees) a symbol object.
`pHandle` points to the symbol handle to release. It’s contents is no longer valid then.

```
int T32_SetSymbolObjName (T32_SymbolHandle handle,
    const char* symName);
```

Set the symbol name of a symbol object. The symbol address within the symbol object is invalidated.
`handle` specifies the symbol object.
`symName` specifies the new symbol name.

```
int T32_GetSymbolObjName (T32_SymbolHandle handle,
    char* symName, uint8_t maxlen);
```

Get the symbol name of a symbol object.
`handle` specifies the symbol object.
`symName` points to a character array allocated by the user.
`maxlen` specifies the maximum length of the character array.

```
int T32_SetSymbolObjAddress (T32_SymbolHandle symHandle,
    T32_AddressHandle addrHandle);
```

Set the address of a symbol object. The symbol name within the symbol object is invalidated.
`symHandle` specifies the symbol object.
`addrHandle` specifies the new address.

```
int T32_GetSymbolObjAddress (T32_SymbolHandle symHandle,
    T32_AddressHandle* pAddrHandle);
```

Get the address of a symbol object.
`symHandle` specifies the symbol object.
`pAddrHandle` points to the address handle receiving the symbol address.

```
int T32_GetSymbolObjSize (T32_SymbolHandle symHandle, uint64_t* pSize);
```

Get the size of a symbol object.
`symHandle` specifies the symbol object.
`pSize` points to the variable receiving the symbol size.

Document Revision Information

Version	Date	Change
4.9	19.06.15	Revised to better show “API Object Handling”
4.8	07.03.14	Added TRACE32 Lua Remote API functions.
4.7	10.12.13	New function T32_DirectAccessUserSignal().
4.6	05.03.13	New functions T32_SetMemoryAccessClass() and T32_WriteRegisterByName()
4.5	07.02.13	Extend TAP Access API to support Arm DAP, AHB, APB, AXI, JTAG-AP, SWD. New functions: T32_DirectAccessResetAll(), T32_DirectAccessSetInfo(), T32_DirectAccessGetInfo(), T32_TAPAccessJTAGResetWithTMS(), T32_TAPAccessJTAGResetWithTRST(), T32_TAPAccessSetShiftPattern(), T32_DAPAccessScan(), T32_DAPAccessInitSWD(), T32_DAPAPAccessReadWrite()
4.4	14.12.10	T32_Terminate(), T32_SetMode() documented. Buffer size corrections for T32_GetMessage() / T32_GetTriggerMessage()
4.3	05.11.10	New functions: T32_ReadVariableValue(), T32_ReadVariableString(), T32_ReadRegisterByName(), T32_GetBreakpointList()