Multi-ICE®

Version 2.2

TAPOp API Reference Guide

ARM®

ARM DUI 0154B

Multi-ICE
TAPOp API Reference Guide

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Change
September 2001	A	First release
February 2002	B	Updated for Multi-ICE Version 2.1

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Conformance Notices

This section contains *ElectroMagnetic Conformity* (EMC) notices and other important notices.

Federal Communications Commission Notice

This device is test equipment and consequently is exempt from part 15 of the FCC Rules under section 15.103 (c).

CE Declaration of Conformity

This equipment has been tested according to ISE/IEC Guide 22 and EN 45014. It conforms to the following product EMC specifications:

The product herewith complies with the requirements of EMC Directive 89/336/EEC as amended.

IEEE Reproduction Permission Notice

Figure 1-4 on page 1-26 reprinted with permission IEEE Std 1149.1-1990. IEEE Standard Test Access Port and Boundary-Scan Architecture Copyright 1998-2001 by IEEE. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

Contents
Multi-ICE TAPOp API Reference Guide

Preface

This preface introduces the Multi-ICE TAPOp API Reference Guide. It explains the structure of the user guide and lists other sources of information that relate to Multi-ICE and ARM debuggers.

This preface contains the following sections:

About this document

This document describes the ARM TAPOp interface used with Multi-ICE Version 2.2. This interface links the TAPOp server, a program that drives the Multi-ICE hardware, to the TAPOp client, a program that uses one or more TAP controllers.

Intended audience

This document is written for users of Multi-ICE on Windows or Unix platforms, using either the ARM *Software Development Toolkit* (SDT) or *ARM Developer Suite* (ADS) development environments. It is assumed that you are a software engineer with some experience of the ARM architecture, or a hardware engineer designing a product that is compatible with Multi-ICE.

Parts of this document assume you have some knowledge of JTAG technology. If you require more information on JTAG, refer to *IEEE Standard 1149.1*, available from the *Institute of Electrical and Electronic Engineers* (IEEE). Refer to the IEEE website for more information at:

`http://www.ieee.org/`

Organization

This document is organized into the following chapters:

Chapter 1 ***Programming with TAPOp***

Read this chapter for information about the software interface between a Multi-ICE server and its client. The chapter includes a description of the way the interface is accessed, the use of macros, and an introduction to the example programs supplied with the product.

Chapter 2 ***TAPOp API Reference***

Read this chapter for reference information about the software interface between a Multi-ICE server and its client. The chapter includes an API reference that describes the purpose of each of the TAPOp procedure calls, and a description of the possible error codes.

Typographical conventions

The following typographical conventions are used in this document:

bold — Highlights ARM processor signal names within text, and interface elements such as menu names. Can also be used for emphasis in descriptive lists where appropriate.

italic Highlights special terminology, cross-references, and citations.

`monospace` Denotes text that can be entered at the keyboard, such as commands, file names and program names, and source code.

`mono`space Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`monospace italic`

Denotes arguments to commands or functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

Further reading

This section lists publications by ARM Limited, and by third parties, that are related to this product.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com/arm/documentation` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at: `http://www.arm.com/arm/tech_faqs`.

ARM publications

This document contains information that is specific to Multi-ICE. The following documents also relate specifically to Multi-ICE:

- *ARM Multi-ICE Installation Guide* (ARM DSI 0005)
- *ARM Multi-ICE User Guide* (ARM DUI 0048)
- Multi-ICE file `Readme.txt`, supplied on the Multi-ICE distribution CD and installed with the product
- Multi-ICE file `proclist.txt`, a list of the processors supported by Multi-ICE and installed with the product.

If you are using Multi-ICE with the *ARM Developer Suite* (ADS) v1.2, refer to the following books in the ADS document suite for information on other components of ADS:

- *Installation and License Management Guide* (ARM DUI 0139)

- *Getting Started* (ARM DUI 0064)
- *CodeWarrior IDE Guide* (ARM DUI 0065)
- *AXD and armsd Debuggers Guide* (ARM DUI 0066)
- *Compilers and Libraries Guide* (ARM DUI 0067)
- *Linker and Utilities Guide* (ARM DUI 0151)
- *Assembler Guide* (ARM DUI 0068)
- *Developer Guide* (ARM DUI 0056)
- *Debug Target Guide* (ARM DUI 0058)
- *Trace Debug Tools User Guide* (ARM DUI 0118)
- *ARM Application Library Programmers Guide* (ARM DUI 0081).

The following additional documentation that might be useful is provided with the *ARM Developer Suite* (ADS):

- *ARM Architecture Reference Manual* (ARM DDI 0100). This is supplied in DynaText format as part of the online books, and in PDF format in *`ADS install directory`*`\PDF\ARM-DDI0100B_armarm.pdf`.

In addition, refer to the following documentation for specific information relating to ARM products:

- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Other publications

The following publications might also be useful to you, and are available from the indicated sources:

- *The Intel® XScale™ Core Developer's Manual*, Datasheet, advance information. Ref 27341401-002. Intel Corp. 2000.
- *Hot-Debug for Intel XScale Core Debug*, White paper. Ref 273539-002. Intel Corp. 2001.
- *IEEE Standard Test Access Port and Boundary Scan Architecture* (IEEE Std. 1149.1) describes the JTAG ports with which Multi-ICE communicates.

Feedback

ARM Limited welcomes feedback both on Multi-ICE and on the documentation.

Feedback on Multi-ICE

If you have any problems with Multi-ICE, please contact your supplier. To help us provide a rapid and useful response, please give:

- the Multi-ICE version you are using
- details of the platforms you are using, including both the host and target hardware types and operating system
- where appropriate, a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- if possible, sample output illustrating the problem

Feedback on this document

If you have any comments on this document, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Programming with TAPOp

This chapter defines the software interface between a Multi-ICE server and a client, for example, the Multi-ICE DLL. It provides a complete function reference and programming guidelines for writing client programs to use hardware connected to Multi-ICE.

This chapter contains the following sections:

1.1 About the TAPOp interface

The TAPOp interface allows ARM processors or third-party devices on an ASIC to be accessed through a Multi-ICE server. This enables you to attach applications to any ARM or non-ARM designs using a standard JTAG port. For example:

- DSP debuggers
- FPGA loader
- communications channel drivers
- test applications.

A TAPOp client can communicate with a Multi-ICE server using:

- *Remote Procedure Call* (RPC), allowing connection to networked workstations
- shared memory, for enhanced performance and for use on workstations without network software installed.

The choice of which communication mechanism to use is determined by the name used to identify the remote workstation. Connections using the name `localhost` use the shared memory method. Other connections use RPC. The connection method used is completely transparent to the client program.

1.1.1 Remote procedure call

RPC enables a program running on one workstation to execute functions on a remote workstation. RPC works by providing stub functions on the calling workstation that pack the inputs to the function into a data block and transmit it over a network connection to the remote workstation. The remote workstation unpacks the function input data, executes the function as normal, then packs up the function outputs to return over the network to the caller.

Subsystems involved in RPC

The RPC standard mandates that other subsystems are used:

- The workstation making the call connects to the other workstation using a *port mapper*, a process (typically called `portmap`) that acts as a connection manager. The workstation acting as an RPC server must register itself with the port mapper when it is ready to accept requests.
- Data is transferred over the network using TCP/IP.
- Data is transferred using a data formatting standard called *eXternal Data Representation* (XDR).

Accessing an RPC server

Initially, there must be three processes when an RPC client connects to its server:

- the port mapper
- the RPC server (the Multi-ICE server)
- the RPC client (the Multi-ICE DLL).

The port mapper and the RPC server must be running on the same computer, and the RPC client must be in network contact with both processes:

1. When the server starts, it attempts to connect to the local portmap service to pass on its connection details. This state is shown in Figure 1-1.

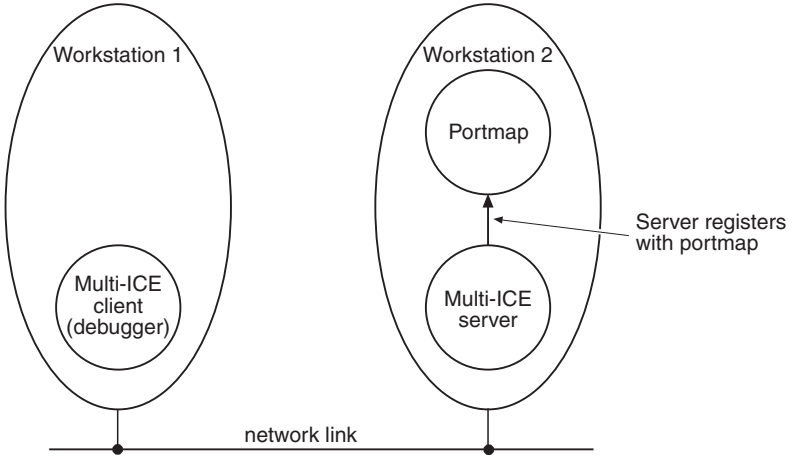

Figure 1-1 RPC server contacts local portmap

2. When the local portmap service is aware of the RPC server, it can pass on the details to a client process such as the Multi-ICE DLL. The DLL connects to the portmap process requesting the Multi-ICE server details. This step is shown in Figure 1-2 on page 1-4.

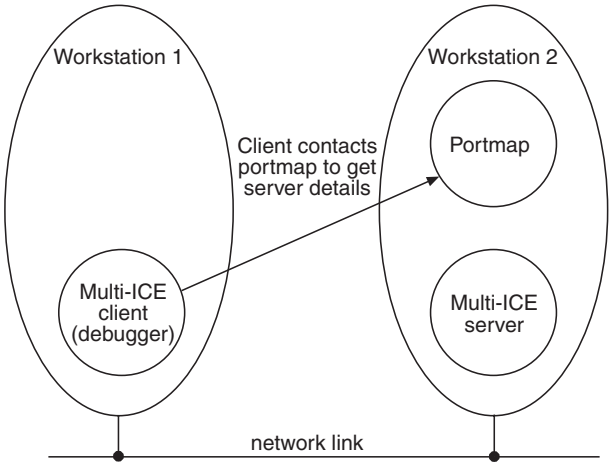

Figure 1-2 RPC client asks portmap for RPC server details

3. The RPC client and the RPC server can connect to each other directly. This is the state shown in Figure 1-3.

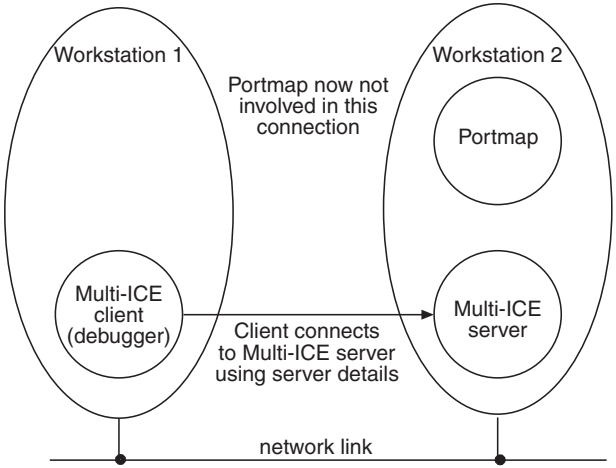

Figure 1-3 RPC client connects to RPC server directly

When the client and server are connected, the portmap process is no longer involved in RPC activity. When the RPC server quits, however, the server must inform the portmap service that it is no longer able to service requests.

1.2 Accessing the Multi-ICE server at the TAPOp level

This section describes how the Multi-ICE server is accessed using RPC and how the TAPOp *Application Program Interface* (API) is used. It contains the following sections:

- *How the Multi-ICE server uses RPC*
- *Making a connection to the server* on page 1-6
- *Multiple clients of the TAPOp layer* on page 1-7
- *TAP controller identification* on page 1-8
- *Order of output of TDI and TMS bits passed over tapop.h* on page 1-8
- *Accessing long scan chains* on page 1-9
- *Efficiency* on page 1-9
- *Error detection and automatic connection deselection* on page 1-10
- *Header file tapshare.h* on page 1-10.

1.2.1 How the Multi-ICE server uses RPC

Multi-ICE uses the *Open Network Computing* (ONC) version of the RPC software. Because RPC uses TCP/IP as its transport mechanism, it is just as easy to connect to a Multi-ICE server over a network or modem as it is to connect locally. Sources for the client-side of the RPC-based TAPOp interface can be found on the installation disk.

When connecting to a Multi-ICE server running on the same workstation, a Multi-ICE client uses a shared memory procedure call mechanism. Because the difference between the mechanisms is hidden, this manual refers only to RPC connections.

The operation of the TAPOp interface follows this pattern:

1. The server starts and is configured to use a particular target device.
2. A client opens a connection to the required device.
3. The client makes RPC calls to the server.
4. The server scans data through the scan chains of the device and returns any results.
5. The client disconnects.

The client can cycle through steps 2, 3, and 4 as many times as required to perform the task.

Each TAPOp call performs one TAP operation. For example, there are TAP operations to write a value to the *Instruction Register* (IR) or read a device scan chain. Because of the low level of the interface and the high overhead of RPC calls, you can batch up multiple RPC calls into macros that are run by the server. This is similar to JAVA applets

downloaded from a web server to a client (browser), because the browser is faster than the link. In Multi-ICE, the client downloads macros to the server because the server is faster than the link. This gives a significant performance improvement.

1.2.2 Making a connection to the server

This section contains the following sections:

- *Opening a TCP connection*
- *Opening a TAPOp connection*
- *Closing a TAPOp connection* on page 1-7.

Opening a TCP connection

For a client to communicate with a server, `TAPOp_RPC_Initialise` must be called to open a connection to the transport layer (TCP). The server location is identified by a callback function `GetServerName` that the client must supply. This opens a two-way channel between the client and server through which procedure calls can be made to the server.

More than one TCP connection to the server can be opened at the same time from the same client. The standard distribution of `rpcclient.c` opens three TCP connections by default and two of these can be used to overlap RPC calls to improve performance. This is done in the Multi-ICE DLL during downloads where multiple threads are used to pipeline RPC calls. When the client has finished, the TCP connection is closed using `TAPOp_RPC_Finalise`.

Opening a TAPOp connection

After a TCP connection has been made to a Multi-ICE server, the client must indicate to the server which device to use. This is known as opening a TAPOp connection, and at any one time there is a single connection between the client and a single device on the server. The TAPOp connection is identified by a connection *Identifier* (ID) that is used in all subsequent calls to the server. The client must close this connection when finishing a debug session.

The connection ID is a logical identifier that the server uses to recognize which client it is talking to, and it identifies a particular device on a particular TAP controller. It is allocated by the server when the client makes a `TAPOp_OpenConnection` call to the server.

At the same time, another TAPOp connection can be present to another device, even on the same TAP controller, using a different connection ID, but a single device can only be connected to a single connection ID. For example, if the client opens three connections to the same device, calls from the client on any of the TCP connections must use the same TAPOp connection ID.

Closing a TAPOp connection

To close the TAPOp connection, call `TAPOp_CloseConnection`. All the macros defined by the client are deleted and storage is freed.

1.2.3 Multiple clients of the TAPOp layer

There can be several simultaneous clients to the TAPOp layer, each one connected to a different TAP controller. Alternatively, clients can connect to the same TAP controller but only if they do not share any resources other than:

- the TAP controller IR
- the use of a scan chain select register.

Two debuggers that access distinct sets of scan chains can both be clients. For example, a DSP scan chain connected to an extra scan chain of the ARM TAP controller. However, two debuggers that access the same scan chain cannot rely on the TAPOp interface to separate their accesses, particularly in the case of potentially sharable resources such as EmbeddedICE breakpoint registers. For example, two debuggers that talk to the same processor must cooperate at a higher level. The ARM *Remote Debug Interface* (RDI) is a suitable level for an ARM processor.

To manage several clients using this interface simultaneously, most of the operations in this interface implicitly request that the client becomes the sole user of the Multi-ICE hardware for the duration of the request. A boolean parameter, `deselect`, indicates if the client is ready to give up this ownership when the operation is complete. When a request to the server is made, if another client has ownership of the Multi-ICE hardware the call fails and the operation is not performed. It is the responsibility of the client to try again. Retrying calls is part of the job of the `TAPCheck` macro.

When a client relinquishes ownership of the Multi-ICE hardware, the TAP interface guarantees that the next time that client gains access to the TAP controller:

- it is in the same TAP state (for example, `Run-Test/Idle`)
- the same instruction is in the IR (for example, `SCAN_N`)
- the same scan chain is selected.

In return, the interface requires that ownership is relinquished only when the TAP controller is in either `Run-Test/Idle` or `Select-DR-Scan` state. This is only an issue for the `TAPOp_AnySequence_W` and `TAPOp_AnySequence_RW` operations, because all other operations leave the TAP controller in one of these two states.

The Multi-ICE server keeps track of the following for each TAPOp connection:

- the last value written to the IR for each TAP controller

- the state the TAP controller was in when ownership is relinquished, so that this state can be restored when ownership reverts to that client
- the last scan chain selected using a SCAN_N instruction.

1.2.4 TAP controller identification

The Multi-ICE server can automatically detect the number of TAP controllers and any details required for each TAP controller, for example the length of the IR register. You also have the option to manually load a configuration file.

All incoming function calls include the connection ID. This can be used to look up the position of the TAP controller in the scan chain, where TAP 0 is nearest to **TDI**. It is therefore necessary for a TAPOp client to inform the server of the TAP controller position and the scan chains it requires when opening a TAPOp connection.

To get a list of devices for a particular server, call TAPOp_GetDriverDetails. This returns a list of device names (for example, ARM7TDMI), their TAP positions, and flags indicating if the devices are connected.

1.2.5 Order of output of TDI and TMS bits passed over tapop.h

The Multi-ICE hardware contains 40-bit data registers so that scan chains with up to 40 bits are accessed efficiently. **TDI**, **TDO**, and **TMS** data is passed over tapop.h using a 40-bit type called ScanData40 constructed from a 32-bit word containing the least significant bits and a byte containing the most significant bits. This type is defined in tapop.h.

```
typedef struct ScanData40 {    unsigned32 low32;
    unsigned8 high8;
} ScanData40;
```

The bits are output as follows:

1. Bit 0 of low32 to bit 31 of low32.
2. Bit 0 of high8 to bit 7 of high8.

Similarly, for an output (**TDO**) block the first **TDO** bit input is placed in bit 0 of low32, and the last in bit 7 of high8.

If a data field is specified as reversed then the same data is written to, or read from, the Multi-ICE reversed data register. Bits are entered from the least significant bit of low32. For example, to write four bits of reversed data, fill bits 0 to 3 in low32. The bits enter the scan chain in the order bit 3, bit 2, bit 1, bit 0.

1.2.6 Accessing long scan chains

TAPOp_AccessDR_W calls contain WRoffset and WRmask parameters. In addition, TAPOp_AccessDR_RW calls contain an RDoffset parameter. The purpose of these parameters is to reduce the number of bits that get written to less than 40 bits using WRmask, or increase the number of bits that can be read/written to more than 40 bits by supplying a read/write offset (RDoffset and WRoffset parameters). For example, to access a 50-bit scan chain use the following steps:

1. Access 40 bits using `WRmask = all ones`.

   Note

   Because the WRmask parameter is a pointer to an array of bits, you can use a shortcut, specifying NULL pointer to mean a mask that is all ones. This only works for a direct TAPOp call, not a macro call.

2. Access the remaining 10 bits using a 10-bit WRmask and a 40-bit offset.

Note

To access a scan chain over 255 bits in length, you must use the TAPOp_AnySequence_RW and TAPOp_AnySequence_W calls to navigate the JTAG state machine and to perform the read or write of the scan chain in the Shift-DR state.

1.2.7 Efficiency

In a Multi-ICE system TAPOp functions are sent across an RPC layer to a workstation and the results must be sent back again. As a result, when a large number of calls are made across this interface, there is a reduction in performance. To reduce the number of calls, you can use TAPOp macros to batch up TAPOp operations and so make a single RPC call perform multiple TAPOp operations.

There are limits to the number of TAPOp operations that can be grouped together because decisions must be made based on **TDO** data. TAPOp macros provide very limited decision-making abilities, so macros must be split so the decisions are made in the TAPOp client.

Large continuous data transfers prevent other TAPOp clients accessing the server. It is recommended that data size is limited to small numbers of kilobytes when communicating over a network to the server.

1.2.8 Error detection and automatic connection deselection

Return codes from TAPOp calls other than `TAPOp_NoError` or `TAPOp_UnableToSelect`, are considered fatal. This means that the TAPOp client might not be able to recover its session without losing data or at least aborting the operation.

Because the TAPOp interface can be used by several client debuggers at once, the connection that has an error is automatically deselected. This ensures that one TAPOp client receiving a fatal error does not block out others.

When a TAPOp call is made other TAPOp clients might also be connected to the same server. It is the responsibility of the client to manage selection errors caused by multiple clients. A macro, `TAPCheck`, is provided in `macros.h` to assist in this task.

The `TAPCheck` macro must be used around all TAPOp and ARMTAP calls. It makes the call and performs error checking on the return value as follows:

- If the call returns `TAPOp_NoError` then it does nothing.
- If the call returns `TAPOp_UnableToSelect`, it retries the same call.
- If the call returns anything else, a call to `TAPOp_ReadMICEflags` is made to try to diagnose the failure. If the flags indicate that the target power is off or has been off, or the target has been reset, then the returned error from the called procedure is overwritten with a more appropriate code.

You must define the following function to allow a fatal error to be dealt with cleanly:

```
void give_up(void)
```

This function might be empty, but more typically contains code to close the client connection.

An example of `TAPCheck` is given in Example 1-4 on page 1-19. Clients can provide their own `TAPCheck` macro based on the supplied example.

1.2.9 Header file tapshare.h

You can use TAPOp functions to read and write data that is held by the server for the various TAPOp clients connected to it, allowing these applications to communicate with each other in a limited manner.

There are two sets of data:

Data that is private to each TAP controller (processor)

There are flags held for each processor. Some flags are debugger read-only and some are read/write.

Note

Use of these flags is optional for a TAPOp client, but if they are used, they provide a way to start and stop processors almost synchronously when several applications are involved.

Data that is common to all TAP controllers

The server does nothing with this data. It maintains it so the TAPOp clients can use it to communicate between themselves. The size of this data is arbitrary, and is currently four words (16 bytes).

It is not necessary for a TAPOp client to have a selected connection in order to use the private data functions, because they do not affect the TAP controller in any way. However, to allow atomic Read-Modify-Write of the common data, the connection must be selected, so a deselect parameter is available.

1.3 Compiling TAPOp client examples

There are two examples of TAPOp applications supplied on the Multi-ICE CD-ROM. You can use them as a starting point from which to develop your own clients. The following platforms are supported:

- Win32 API (Windows 95, Windows 98, Windows Me, Windows NT 4, or Windows 2000)
- HP-UX 10.20
- Solaris 2.6 and Solaris 7.0

1.3.1 Sources and executables for Windows clients

The sources required for building TAPOp applications using the Win32 API are provided on the CD and are installed if the **TAPOp source files** option is checked at install time. Precompiled executables for the examples are also provided in the `examples` subdirectory in the files `example1.exe` and `example2.exe`.

The sources and project files work with Microsoft Visual C++ 6. These files are also compatible with Microsoft Visual C++ 5.

A file `dethost.h` is included in the sources. This file maps various operating system `#defines` onto a number of internal `#defines`, which control the compilation of the Multi-ICE RPC client source code.

1.3.2 Sources for UNIX clients

Sources for the Unix platforms listed below are provided on the CD but are not installed by the install script:

- Solaris 2.6 and Solaris 7.0
- HP-UX 10.20.

To extract the source files:

1. Insert the Multi-ICE CD into the CD-ROM drive of your Solaris or HP-UX computer.
2. Invoke a terminal window and ensure the CD-ROM is mounted. Refer to the system documentation for further information.
3. Create a directory to contain the sources, and `cd` to it. For example:

   ```
   mkdir /home/jbloggs/multiice
   cd /home/jbloggs/multiice
   ```

4. The file `source.tar` is stored in the directory `unix` under the CD-ROM mount point. Extract the source files to your source area using the `tar` command:


```
tar xf /mnt/cdrom/unix/source.tar
```

where */mnt/cdrom* is the mount point of the CD-ROM

Makefiles are included to build the two provided examples. Both makefiles are set up to build under Solaris, so if you are using HP-UX, you must comment out the three lines under the text `for Solaris` and uncomment the three lines under `for HPUX`. If you are not using the standard compiler for your platform, for example, you are using `gcc`, you must also edit the compiler names and options as required.

A file `dethost.h` is included in the sources. This file maps various operating system `#defines` onto a number of internal `#defines`, which control the compilation of the Multi-ICE RPC client source code.

Known problem with HP-UX clients

The RPC functionality under UNIX is provided by operating system libraries, and so the ONC RPC library is not required. However, HP-UX has no interface to set the RPC timeout, and so clients running under HP-UX cannot set the timeout period.

1.3.3 TAPOp example 1

This example connects to TAP controller 0, and displays the ID code for the TAP controller. The example requires that:

- the server has been started and configured
- the first TAP position is free for connection.

This example demonstrates:

- initialization and finalization of the RPC transport layer
- use of `TAPOp_GetDriverDetails` to find out which cores the Multi-ICE server has configured
- use of `TAPOp_OpenConnection` to open a connection to the server and obtain a connectId
- use of `ARMTAP_AccessIR` to write an instruction (IDCODE) to the TAP IR
- use of `ARMTAP_AccessDR_RW` to read out data from a scan chain (in this case, the ID code register)
- use of `TAPOp_CloseConnection` to close a connection to the server.

To run the example:

1. Start ADW, ADU or AXD.
2. Load `example.axf`.

3. Select **Go**.

1.3.4 TAPOp example 2

This example makes a connection to a TAP controller (in the same way as for *TAPOp example 1* on page 1-13), then waits for data coming from the DCC. If there is data available, it is printed to the debugger console.

The example uses the techniques shown in *TAPOp example 1* on page 1-13, plus:

- use of `TAPOp_DefineMacro` and `TAPOp_RunMacro` procedure calls
- use of `ARMTAP_AccessDR_RW_And_Test` within a TAPOp macro to exit prematurely.

To run the example:

1. Start ADW, ADU, or AXD.
2. Select **View** → **Debugger Internals**.
3. Change the `semihosting_enabled` value to 0.
4. Select **Options** → **Configure Debugger** and configure to use the Channel Viewer.
5. Load `ccout4ever.axf`.
6. Start the Channel Viewer.
7. Select **Go**.
8. Close your debugger.
9. Run `example2.exe`.

1.3.5 Building your own client programs

It is imperative that the data types defined in `basetype.h` are defined correctly for the compiler you are using. The Windows and UNIX sources are set up for Microsoft Visual C++ 6 and the host C compilers on Solaris and HP-UX.

The correct definitions are: **`unsigned8`** is exactly 8 bits, **`unsigned16`** is exactly 16 bits, and **`unsigned32`** is exactly 32 bits.

If you do not use the provided workspace files or makefiles, you must include the following source files in your project:

- `mice_clnt.c`
- `mice_xdr.c`
- `oncrpc.lib`

- rpcclient.c

For Win32 applications you must also include nonrpcclient.c.

You must also #include files into those source files in your project that call TAPOp functions. Two directories must be added to the header include path:

- *base_path*/include
- *base_path*/include/rpc

base_path is the directory containing the source files.

To use:

Standard TAPOp_ and ARMTAP_ functions and structures

```
#include “tapop.h”
```

TAPOp server macro functions

```
#include “macros.h”
```

TAPOp advanced server features and chip drivers

```
#include “tapshare.h”
```

Note

The definitions previously included in the file armtapop.h are now in tapop.h. Code that uses armtapop.h must be changed to compile with Multi-ICE 2.2 source, by deleting all #includes of armtapop.h.

You must link precompiled RPC library file oncrpc.lib with the application to resolve the RPC functions used by micr_xdr.c.

1.4 Using TAPOp macros

This section describes how to write and run TAPOp macros. It contains the following sections:

- *Writing a macro*
- *Single operation macro example* on page 1-18
- *Multiple operation macro example* on page 1-20
- *Complex macro example* on page 1-22
- *Passing fixed and variable parameters to TAPOp macros* on page 1-24.

The following list gives a functional summary of the macro procedure calls:

Creating macros `TAPOp_DefineMacro`

Deleting macros `TAPOp_DeleteMacro, TAPOp_DeleteAllMacros`

Displaying macros (for debug)

`TAPOp_DisplayMacro`

Running macros `TAPOp_RunMacro, TAPOp_RunBufferedMacro, TAPOp_FillMacroBuffer`

Synchronized stop and start macros

`TAPOp_SetControlMacros`

The macro procedure calls are given in full in the alphabetical listing of all procedure calls in *TAPOp function alphabetic reference* on page 2-26.

1.4.1 Writing a macro

To write a macro:

1. Decide how instructions must be grouped together to optimize the speed of transfer. Better performance results from macros containing a large number of operations.

2. Convert the parameters of the normal TAPOp operations to the macro versions. The structure for the data required for the macro versions of the instructions is in the header file `macstruct.h`.

It is good practice to get the nonmacro version of a client working before attempting to turn it into a macro, because it is harder to debug when in macro format.

The prototype for the standard `ARMTAP_AccessDR_W` TAPOp function call is shown in Example 1-1 on page 1-17.

Example 1-1 ARMTAP_AccessDR_W C function prototype declaration

```
extern TAPOp_ErrorARMTAP_AccessDR_W(unsigned8 connectId, ScanData40 *TDIbits,
                  unsigned8 TDIrev, unsigned8 len, unsigned8 WRoffset,
                  ScanData40 *WRmask, unsigned8 nclks, unsigned8 deselect)
```

When creating the ARMTAP_AccessDR_W instruction macro, a structure containing the parameters shown in Example 1-2 is used.

Example 1-2 ARMTAP_AccessDR_W macro structure declaration

```
typedef struct MAC_ARMTAP_AccessDR_WIn {    unsigned32      TDIbits1;
    unsigned8       TDIbits2;
    unsigned8       TDIrev;
    unsigned8       len;
    unsigned8       WRoffset;
    unsigned32      WRmask1;
    unsigned8       WRmask2;
    unsigned8       nclks;
} MAC_ARMTAP_AccessDR_WIn;
```

There are a number of differences:

- The connectId is not present. This parameter is passed to the TAPOp_RunMacro function.
- The deselect parameter is not present. This parameter is passed to the TAPOp_RunMacro function and is considered when the macro terminates.
- TDIbits and WRmask are passed as two parameters rather than as one ScanData40 type.

You must decide which parameters are fixed (define time) and which are variable (runtime). This depends on the specific programming task. There are two things to consider when choosing:

- There is a trade-off between managing many different fixed macros and a smaller number of variable macros.
- There are some tasks that can be performed very efficiently using macros because of the option to repeat macro execution from a large data buffer with a single call. A good example of this is writing data using an LDMIA.

You are recommended to use this strategy:

1. Perform the initial coding of the program without using macros. This enables you to find out what TAPOp sequences your application requires.
2. Run and analyze this program to determine if there are specific speed or efficiency bottlenecks. Address these by writing macros for the parts of the system involved.
3. Analyze the program again. If there are still problems with speed or efficiency, attempt to use macros more widely, and apply more general techniques for efficiency improvement (for example, avoiding calling functions, performing calculations outside loops, and caching results).

Proper error checking is essential with all TAPOp and ARMTAP functions. It is strongly recommended that the macro definition operation is performed within (a version of) the `TAPCheck` macro.

1.4.2 Single operation macro example

Example 1 sets up a macro to run an LDMIA instruction on an ARM processor using `ARMTAP_AccessDR_W`. It assumes that a valid `connectId` has already been returned by a call to `TAPOp_OpenConnection`. This operation can be performed using the standard TAPOp function call shown in Example 1-3.

Example 1-3 Executing an LDMIA instruction using a standard function call

```
ScanData40  opcode = {0xE89E3FFF, 0}; /* op-code for LDMIA instr */
unsigned8   TDIrev = 1,
            len = 32,
            nclks = 1,
            deselect = 1;
TAPCheck(ARMTAP_AccessDR_W(connectId,&opcode,TDIrev,len,0,NULL,nclks,deselect));
```

To define the macro

To illustrate setting up a simple macro, the macro required to execute a single LDMIA instruction is shown with all parameters fixed.

Macros are defined by writing the parameters for each TAPOp function into an array and passing that array the name of the function to execute to `TAPOp_DefineMacro`. An example is shown in Example 1-4 on page 1-19. The pattern for any macro definition is, for each TAP function that is included in the macro:

1. Initialize the fixed parameter array.

2. Use the NREnterParam*xx* routines to enter the parameter values into the parameter array (substitute the parameter type code for *xx*).

3. Call TAPOp_DefineMacro to add this call to the end of the macro.

These steps can be simplified if you use the predefined C macros:

- InitParams
- NREnterParamBytes
- NREnterParamU32
- NREnterParamU16
- NREnterParamU8.

These C macros assume the existence of an array of type unsigned8 Values[MACRO_ARGUMENT_AREA_SIZE] and a variable int ValPtr to reference the current position in this array. There is a C preprocessor definition of the symbol MACRO_ARGUMENT_AREA_SIZE in macros.h.

An example of the process of setting up and defining a macro is given in Example 1-4, conforming to the following sequence:

1. Values and ValPtr are declared as global variables.

2. For every macro that you require, you must include code that:
   a. Calls InitParams to set up ValPtr.
   b. Writes data values corresponding to the parameters of a TAPOp call using a sequence of NREnterParamx*x* calls.
   c. Calls TAPOp_DefineMacro with parameters that include the macro identifier, the name of the function to call, and the details of the parameter values array.

3. The function TAPOp_DisplayMacro is called to cause the Multi-ICE server to write out the details of the macro to the debug window.

   It is recommended that if you are unsure of whether the details of a macro are correct that a TAPOp_DisplayMacro call is made. This enables you to check what the server has stored.

Example 1-4 Defining a macro using ARMTAP_AccessDR_W

```
#define MACRO1     1
#define LDMIA      (unsigned32) 0xE89E3FFF /* op-code for LDMIA instruction */
#define SC_DATABUS (unsigned8) 33          /* length of the scan chain */
int           ValPtr;
unsigned8      Values[MACRO_ARGUMENT_AREA_SIZE]; /* constants in macros.h */
```

```
/* All parameters are fixed. */
InitParams;                 /* Reset ValPtr */
NREnterParamU32(LDMIA);     /* Place TDIbits1 in Values array */
NREnterParamU8(0);          /* Place TDIbits2 in Values array */
NREnterParamU8(1);          /* Place TDIrev in Values array */
NREnterParamU8(SC_DATABUS); /* Place len in Values array */
NREnterParamU8(0);          /* Place WRoffset in Values array */
NREnterParamU32(0xFFFFFFFF);/* Place WRmask1 in Values array */
NREnterParamU8(0xFF);       /* Place WRmask2 in Values array */
NREnterParamU8(1);          /* Place nclks in Values array */
/* macro line can now be added */
TAPCheck(TAPOp_DefineMacro(connectId, MACRO1, "ARMTAP_AccessDR_W:12345678",
                           1, Values, ValPtr));
```

In the call of TAPOp_DefineMacro in Example 1-4 on page 1-19 the string 12345678 means that parameters 1 to 8 are fixed.

To run the macro

The macro is run as shown in Example 1-5.

Example 1-5 Running the ARMTAP_AccessDR_W macro

```
int lnerr,lperr,     /* Variables for error position detecting */
    resultvalues,    /* In this example, no data is returned, but */
    resultsize;      /* variables must be defined */
InitParams;          /* Reset ValPtr */
resultsize = 0;
TAPCheck(TAPOp_RunMacro(connectId, MACRO1, Values, ValPtr, &lnerr, &lperr,
                        &resultvalues, &resultsize, 1, 1));
```

Because all the parameters are fixed, you do not have to load any parameters for TAPOp_RunMacro. It is still necessary, however, to use InitParams to indicate that there are no variable parameters.

1.4.3 Multiple operation macro example

To demonstrate sending one of the parameters at runtime, a macro line is entered to accept the least significant 32 TDIbits at runtime for three instructions. The purpose of this macro is to send the following instructions to the data register:

- LDMIA instruction
- two NOPs.

To define the macro

The code in Example 1-6 shows you how to define the parameterized macro.

Example 1-6 Defining a parameterized macro

```
#define MACRO2      2
#define SC_DATBUS    (unsigned8)33
int        ValPtr;
unsigned8  Values[MACRO_ARGUMENT_AREA_SIZE];
/* Send 1 data word out.*/
/* Parameters 2 to 8 are fixed, parameter 1 sent at run-time */
InitParams;                 /* Reset ValPtr */
NREnterParamU8(0);          /* Place param2 (TDIbits2) in Values array */
NREnterParamU8(1);          /* Place param3 (TDIrev) in Values array */
NREnterParamU8(SC_DATABUS); /* Place param4 (len) in Values array */
NREnterParamU8(0);          /* Place param5 (WRoffset) in Values array */
NREnterParamU32(0xFFFFFFFF); /* Place param6 (WRmask1) in Values array */
NREnterParamU8(0xFF);       /* Place param7 (WRmask2) in Values array */
NREnterParamU8(1);          /* Place param8 (nclks) in Values array */
/* 'line' can now be added */
TAPCheck(TAPOp_DefineMacro(connectId, MACRO2, "ARMTAP_AccessDR_W:2345678",
                           3, Values, ValPtr));
```

To run the macro

The code in Example 1-7 shows you how to run the macro defined in Example 1-6. The macro uses variable parameters and so the `TAPOp_RunMacro` call requires data in the `Values` array. You must format the data into the `Values` array using one of the macro enter functions, for example `NREnterParamU32`, `NREnterParamU16`, and `NREnterParamU8`.

Example 1-7 Running the parameterized macro

```
#define LDMIA   (unsigned32) 0xE89E3FFF   /* op-code for LDMIA instr */
#define NOP     (unsigned32) 0xE1A00000   /* op-code for NOP instr */
int lnerr,lperr,         /* Variables for error position detecting */
    resultvalues,        /* In this example, no data is returned, but */
    resultsize;          /* variables must be defined */
InitParams;              /* Reset ValPtr */
resultsize = 0;
NREnterParamU32(LDMIA);  /* Place param1 (TDIbits1) for first iteration */
NREnterParamU32(NOP);    /* Place param1 (TDIbits1) for second iteration */
```

```
NREnterParamU32(NOP);    /* Place param1 (TDIbits1) for third iteration */
TAPCheck(TAPOp_RunMacro(connectId,MACRO2,Values,ValPtr,&lnerr,&lperr,
                        &resultvalues, &resultsize, 1, 1));
```

1.4.4 Complex macro example

Example 1-8 shows how three lines are added to a macro. It uses a combination of fixed and variable parameters and also uses multiple iterations of the instruction in a single line.

This macro sends the following instructions and data to the data register at runtime:

- LDMIA instruction
- two NOPs
- 14 x 32-bit data words
- two NOPs
- STMIA instruction.

A NOP instruction with the breakpoint bit set is fixed in the server macro.

To define the macro

The code in Example 1-8 shows you how to define the multiple-line macro.

Example 1-8 Defining a multiple line macro

```
#define MACRO3      3                       /* macro number 3 */
#define NOP         (unsigned32)0xE1A00000  /* A no-op for ARM7TDMI */
#define SC_DATBUS   (unsigned8)  33
int         ValPtr;
unsigned8  Values[MACRO_ARGUMENT_AREA_SIZE];
void define_send14_macro(void)
{
    /* Parameters 2 to 8 are fixed, parameter 1 sent at run-time. */
    InitParams;                   /* Reset ValPtr */
    NREnterParamU8(0);            /* Place TDIbits2 in Values array */
    NREnterParamU8(1);            /* Place TDIrev in Values array */
    NREnterParamU8(SC_DATABUS);   /* Place len in Values array */
    NREnterParamU8(0);            /* Place WRoffset in Values array */
    NREnterParamU32(0xFFFFFFFF);  /* Place WRmask1 in Values array */
    NREnterParamU8(0xFF);         /* Place WRmask2 in Values array */
    NREnterParamU8(1);            /* Place nclks in Values array */
    /* 'line' 1 can now be added - but it is entered 19 times */
    TAPCheck(TAPOp_DefineMacro(connectId, MACRO3, "ARMTAP_AccessDR_W:2345678",
                               19, Values, ValPtr));
```

```
    /* send a NOP with the breakpoint bit (32) set.*/
    /* All 8 parameters are fixed.*/
    InitParams;                   /* Reset ValPtr */
    NREnterParamU32(NOP);         /* Place TDIbits1 in Values array */
    NREnterParamU8(1);            /* Place TDIbits2 in Values array */
    NREnterParamU8(1);            /* Place TDIrev in Values array */
    NREnterParamU8(SC_DATABUS);   /* Place len in Values array */
    NREnterParamU8(0);            /* Place WRoffset in Values array */
    NREnterParamU32(0xFFFFFFFF);  /* Place WRmask1 in Values array */
    NREnterParamU8(0xFF);         /* Place WRmask2 in Values array */
    NREnterParamU8(1);            /* Place nclks in Values array */
    /* 'line' 2 can now be added */
    TAPCheck(TAPOp_DefineMacro(connectId, MACRO3, "ARMTAP_AccessDR_W:12345678",
                               1, Values, ValPtr));
    /* Send 1 data word out.*/
    /*Parameters 2 to 8 are fixed, parameter 1 sent at run-time */
    InitParams;                   /* Reset ValPtr */
    NREnterParamU8(0);            /* Place TDIbits2 in Values array */
    NREnterParamU8(1);            /* Place TDIrev in Values array */
    NREnterParamU8(SC_DATABUS);   /* Place len in Values array */
    NREnterParamU8(0);            /* Place WRoffset in Values array */
    NREnterParamU32(0xFFFFFFFF);  /* Place WRmask1 in Values array */
    NREnterParamU8(0xFF);         /* Place WRmask2 in Values array */
    NREnterParamU8(1);            /* Place nclks in Values array */
    /* 'line' 3 can now be added */
    TAPCheck(TAPOp_DefineMacro(connectId, MACRO3, "ARMTAP_AccessDR_W:2345678",
                               1, Values, ValPtr));
    /* check that macro has been entered OK */
    TAPCheck(TAPOp_DisplayMacro(connectId, MACRO3));
}
```

To run the macro

You can now run the macro defined in Example 1-8 on page 1-22 using the code shown in Example 1-9. The macro uses variable parameters and so the `TAPOp_RunMacro` call requires data in the `Values` array. You must format the data into the `Values` array using one of the macro enter functions, for example `NREnterParamU32`, `NREnterParamU16`, and `NREnterParamU8`.

Example 1-9 Running the multiple line macro

```
#define LDMIA  (unsigned32)  0xE89E3FFF
#define STMIA  (unsigned32)  0xE8AE3FFF
#define NOP    (unsigned32)  0xE1A00000
TAPOp_Error run_send14_macro(unsigned32 *data, int *lnerr, int *lperr)
{
```

```
    int j,                /* loop counter */
        lnerr, lperr,     /* Variables for error position detecting*/
        resultvalues,     /* In this example, no data is returned, but*/
        resultsize;       /* variables must be defined */
    InitParams;           /* reset ValPtr */
    /* Send parameters for 'line' 1, 19 unsigned32 words required */
    NREnterParamU32(LDMIA);              /* 1 */
    NREnterParamU32(NOP);                /* 2 */
    NREnterParamU32(NOP);                /* 3 */
    for (j=0; j<14; j++) {
        NREnterParamU32(data[j]);        /* 4 to 17 */
    }
    NREnterParamU32(NOP);                /* 18 */
    NREnterParamU32(NOP);                /* 19 */
    /* 'line' 2 is now 'skipped' as it requires no further parameters */
    /* Send parameter for 'line' 3, 1 unsigned32 word required */
    NREnterParamU32(STMIA);
    resultsize = 0;
    TAPCheck(TAPOp_RunMacro(connectId, MACRO3, Values, ValPtr, lnerr, lperr,
                            &resultvalues, &resultsize, 1, 1));
    return t_err;
}
```

1.4.5 Passing fixed and variable parameters to TAPOp macros

To simplify the passing of both fixed and variable parameters to TAPOp macros, a set of predefined C macros is provided in `macros.h`.

The macros that enter parameters use the following variables, which must be defined:

```
int          ValPtr;
unsigned8    Values[MACRO_ARGUMENT_AREA_SIZE];
```

The value of `MACRO_ARGUMENT_AREA_SIZE` is defined in the file `macros.h`, which is in the installed TAPOp source directory.

C macros for passing fixed and variable parameters

The following macros return `IErr_NotEnoughMacroArgumentSpace` if `ValPtr` is greater than `MACRO_ARGUMENT_AREA_SIZE`:

`InitParams` Resets ValPtr, required before the first fixed parameter of each macro line at define time, and before the first variable parameter at runtime.

`EnterParamBytes (void *byte_ptr, int nbytes)`

Enters a number of bytes into the `Values` array.

`EnterParamU8 (unsigned8 byte)`

Enters a single byte into the `Values` array.

`EnterParamU16 (unsigned16 halfword)`

Enters a 16-bit halfword as two bytes into the `Values` array.

`EnterParamU32 (unsigned32 word)`

Enters a 32-bit word as four bytes into the `Values` array.

The following macros do *Not Return* (NR) an error if `ValPtr` is greater than `MACRO_ARGUMENT_AREA_SIZE` although an error message is displayed and the array boundary is not exceeded. The macro calls `fprintf(stderr, "...");` to write the message:

`NREnterParamBytes (void *byte_ptr, int nbytes)`

Enters a number of bytes into the `Values` array.

`NREnterParamU8 (unsigned8 byte)`

Enters a single byte into the `Values` array.

`NREnterParamU16 (unsigned16 halfword)`

Enters a 16-bit halfword as two bytes into the `Values` array.

`NREnterParamU32 (unsigned32 word)`

Enters a 32-bit word as 4 bytes into the `Values` array.

1.5 TAP controller state transitions

Figure 1-4 shows the TAP controller state transitions.

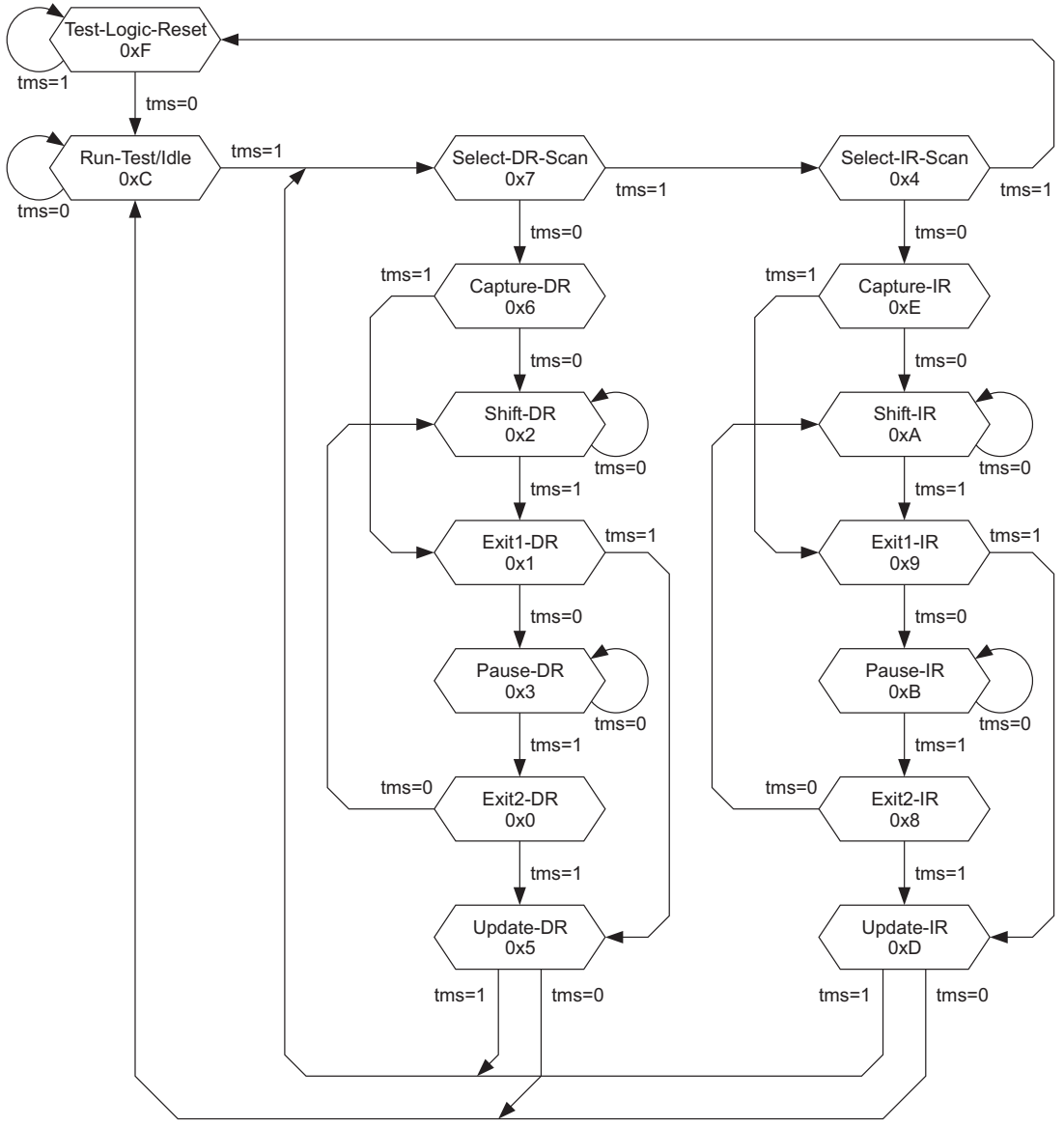

Figure 1-4 Test access port (TAP) controller state transitions

From IEEE Std. 1149.1-1999. Copyright 1998-2002 IEEE. All rights reserved.

Chapter 2
TAPOp API Reference

This chapter defines the software interface between a Multi-ICE server and a client, for example, the Multi-ICE DLL. It provides a complete function reference and programming guidelines for writing client programs to use hardware connected to Multi-ICE. This chapter contains the following sections:

2.1 TAPOp calls listed by function

This section lists TAPOp calls according to their general function:

- *TAP controller and scan chain access*
- *Data read and write* on page 2-3
- *Connection control* on page 2-3
- *Debugging* on page 2-4
- *Macro usage* on page 2-4
- *User I/O* on page 2-5
- *Other functions* on page 2-5.

Each procedure is listed in *TAPOp function alphabetic reference* on page 2-26.

2.1.1 TAP controller and scan chain access

Table 2-1 shows the TAPOp functions that access the TAP controller and scan chains.

Table 2-1 TAP controller and scan chain access

Type of TAP operation	Generic TAP operations (idles in Run-Test/Idle)	ARM specific operations (idles in Select-DR-Scan)

IR access (write-only)	`TAPOp_AccessIR`	`ARMTAP_AccessIR`
		`ARMTAP_AccessIR_1Clk`
		`ARMTAP_AccessIR_nClks`

DR (scan chain) write	`TAPOp_AccessDR_W`	`ARMTAP_AccessDR_W`
		`ARMTAP_AccessDR_NoClk_W`
		`ARMTAP_AccessDR_1Clk_W`

DR (scan chain) read/write	`TAPOp_AccessDR_RW`	`ARMTAP_AccessDR_RW` `ARMTAP_AccessDR_RW_And_Test`
TAP controller manual control	`TAPOp_AnySequence_W` `TAPOp_AnySequence_RW`	-
ARM clock control	-	`ARMTAP_ClockARM`

The *Instruction Register* (IR) and *Data Register* (DR) relate to:

- the TAP controller instruction register where, for example, a `SCAN_N` instruction can be placed
- the device data or instruction register(s) where data relating to the device under test is placed.

Because the natural idle state for ARM devices is in Select-DR-Scan, but for standard JTAG it is Run-Test-Idle, there are two versions of most of these operations. The two versions traverse the TAP state machine (see *TAP controller state transitions* on page 1-26) in different ways but are otherwise the same.

2.1.2 Data read and write

Table 2-2 shows the TAPOp functions that manipulate information stored on the Multi-ICE server on behalf of client applications. Predefined flags are described in more detail in *TAPOp inter-client communication flags* on page 2-22.

Table 2-2 Data read/write

Data type	Function
Private data	TAPOp_ReadPrivateFlags TAPOp_WritePrivateFlags
Common data	TAPOp_ReadCommonData TAPOp_WriteCommonData

2.1.3 Connection control

Table 2-3 shows the TAPOp functions that manipulate server connections. An RPC connection must be open to make a TAPOp connection, so the columns *RPC connection control* and *TAPOp connection control* are not alternatives.

The function TAPOp_PingServer is automatically called in a separate thread by the Windows support code, so your Windows application does not have to call this.

Table 2-3 Connection control

Type of connection	RPC connection control	TAPOp connection control
Opening connections	TAPOp_RPC_Initialise GetServerName	TAPOp_OpenConnection TAPOp_GetDriverDetails TAPOp_GetModelDetails
Closing connections	TAPOp_RPC_Finalise	TAPOp_CloseConnection
Connection heartbeat	-	TAPOp_PingServer
Miscellaneous	TAPOp_RPC_SetTimeout	-

2.1.4 Debugging

The functions shown in Table 2-4 are provided to simplify debugging of client programs by showing more information about the connection than normal. You are recommended not to call these functions from production code.

Table 2-4 Debugging

Debug facility	Function
Displaying macros	`TAPOp_DisplayMacro`
RPC logging	`TAPOp_SetLogging`

2.1.5 Macro usage

The functions shown in Table 2-5 enable you to manipulate server macros. Server macros encapsulate sequences of functions to be sent to the server ahead of time and then run on demand. The time required to communicate with the server is significantly reduced, making access to the device much faster.

Macros defined for one connection are not available to others, and are deleted when the connection is terminated.

Table 2-5 Macro usage

Macro function	Normal macro usage	Advanced macro usage
Creating macros	`TAPOp_DefineMacro`	-
Deleting macros	`TAPOp_DeleteMacro`	-
	`TAPOp_DeleteAllMacros`	-
Displaying macros (for debug)	`TAPOp_DisplayMacro`	-
Running macros	`TAPOp_RunMacro`	`TAPOp_RunBufferedMacro` `TAPOp_FillMacroBuffer`
Synchronized stop/start macros	-	`TAPOp_SetControlMacros`

2.1.6 User I/O

Table 2-6 shows the functions that write to the user output bits within the interface unit, and control the two reset signals. The server flags contain the state of the user input bits.

Table 2-6 User I/O

Signal	Function
Output bits	`TAPOp_WriteMICEUser1` `TAPOp_WriteMICEUser2`
System reset	`TAPOp_SystemResetSignal` `TAPOp_TestResetSignal`

2.1.7 Other functions

Table 2-7 shows those functions that do not fit into the other categories.

Table 2-7 Other functions

Description	Function
Flush buffer	`TAPOp_FlushScanQueue`
Multiple TAP control	`TAPOp_SetAutoBypassInstruction`
Delay for specified period	`TAPOp_Wait`

2.2 TAPOp type definitions

This section lists the TAPOp public types:

- *int8*
- *int16*
- *int32* on page 2-7
- *unsigned8* on page 2-7
- *unsigned16* on page 2-7
- *unsigned32* on page 2-8
- *MultiICE_DriverDetails* on page 2-8
- *ScanData40* on page 2-9.

2.2.1 int8

An integer type used by the TAPOp API as the type of variables that must be exactly 8 bits in size.

Declaration

```
typedef signed char int8; /* or whatever is 8 bits on your system */
```

Usage

This is one of several signed integer types that are usedby the API in place of the standard C type name when the actual size of the variable matters.

2.2.2 int16

An integer type used by the TAPOp API as the type of variables that must be exactly 16 bits in size.

Declaration

```
typedef signed short int16; /* or whatever is 16 bits on your system */
```

Usage

This is one of several signed integer types that are usedby the API in place of the standard C type name when the actual size of the variable matters.

2.2.3 int32

An integer type used by the TAPOp API as the type of variables that must be exactly 32 bits in size.

Declaration

```
typedef signed int int32; /* or whatever is 32 bits on your system */
```

Usage

This is one of several signed integer types that are usedby the API in place of the standard C type name when the actual size of the variable matters.

2.2.4 unsigned8

An integer type used by the TAPOp API as the type of variables that must be exactly 8 bits in size.

Declaration

```
typedef unsigned char unsigned8; /* or whatever is 8 bits on your system */
```

Usage

This is one of several unsigned integer types that are usedby the API in place of the standard C type name when the actual size of the variable matters.

2.2.5 unsigned16

An integer type used by the TAPOp API as the type of variables that must be exactly 16 bits in size.

Declaration

```
typedef unsigned short unsigned16; /* or whatever is 16 bits on your system */
```

Usage

This is one of several unsigned integer types that are usedby the API in place of the standard C type name when the actual size of the variable matters.

2.2.6 unsigned32

An integer type used by the TAPOp API as the type of variables that must be exactly 32 bits in size.

Declaration

```
typedef unsigned int unsigned32; /* or whatever is 32 bits on your system */
```

Usage

This is one of several unsigned integer types that are usedby the API in place of the standard type when the actual size of the variable matters.

2.2.7 MultiICE_DriverDetails

A structure containing details, such as the device name, of the devices connected to a Multi-ICE server.

Declaration

```
struct DrvDetails{
    u_char TAP_pos;
    u_char DriverVersionReqd;
    u_char IsConnected;
    u_char DriverName[255];
    u_char DriverOptions[255];
};typedef struct DrvDetails DrvDetails;typedef DrvDetails
MultiICE_DriverDetails;
```

where:

`TAP_pos` An integer indicating the position of the device in the scan chain, starting at `0` for the device nearest to the TDO pin on the Multi-ICE hardware interface.

Note

- The combination of the `TAP_pos` and a connection to a server uniquely identify a device.
- If `TAP_pos` is `-1`, the named device is sharing a TAP controller with another device in the scan chain.

`DriverVersionReqd` The version number of the driver `.mul` file required for this device to be usable by the client.

`IsConnected` A boolean that is TRUE if a client is connected to the device and FALSE otherwise.

A connection is made by `TAPOp_OpenConnection` and is broken by `TAPOp_CloseConnection`.

`DriverName` The name of the driver. For example, `ARM966E-S`, or `ARM7TDMI`, or `XC9572`. Drivers cannot be renamed.

`DriverOptions` Driver options string, used by the user to customize the behavior of the client driver software. For more details of driver option strings, see the description of the configuration file syntax in the *Multi-ICE User Guide*.

Usage

A Multi-ICE client application can ask the Multi-ICE server for the details of the devices that are configured on the server. A client calling `TAPOp_GetDriverDetails` must scan the returned counted array of `DrvDetails` structures to find out the drivers that are configured.

2.2.8 ScanData40

A 40-bit integer type used to store data being written to, or read from, the Multi-ICE hardware data register.

Declaration

```
typedef struct ScanData40{
    unsigned32 low32;
    unsigned8  high8;
}
```

where:

`low32` The least significant 32 bits of the 40-bit integer.

`high8` The most significant 8 bits of the 40-bit integer.

Usage

Functions that transfer data to or from a JTAG scan chain use a 40-bit wide Multi-ICE hardware data register. `ScanData40` is the data type used for these data transfers by the TAPOp API functions.

Note

When using TAPOp macro functions (for example, `TAPOp_DefineMacro`) a `ScanData40` parameter is included as two separate parameters, one an `unsigned32` and one an `unsigned8`.

2.3 TAPOp constant and macro definitions

TAPOp defines the following public constants and macros:

- *BIG_END_CLIENT*
- *bool*
- *EnterParamBytes* on page 2-12
- *EnterParamU32, EnterParamU16, EnterParamU8* on page 2-12
- *HAS_ONCRPC_BUILTIN* on page 2-13
- *InitParams* on page 2-13
- *MACRO_ARGUMENT_AREA_SIZE* on page 2-14
- *MACRO_RESULT_SIZE* on page 2-14
- *NREnterParamBytes* on page 2-14
- *NREnterParamU32, NREnterParamU16, NREnterParamU8* on page 2-14
- *TAPCheck macro* on page 2-15
- *TAPOp_Error* on page 2-17
- *WAIT_FOR_FIRST_CONNECTION* on page 2-21
- *WAIT_FOR_NORMAL_OPERATIONS* on page 2-21.

2.3.1 BIG_END_CLIENT

This macro, defined in dethost.h, is computed by looking for macros that might be defined by the compiler. It defines whether results received from the server must be byte-swapped or not.

Declaration

```
#define BIG_END_CLIENT 1
```

The macro is defined if any of the symbols `__hppa`, `__svr4__` or `__SVR4__` is defined.

2.3.2 bool

This macro is defined as an 8-bit wide value containing 0 or nonzero and used as a type name in the source.

Declaration

```
#define bool unsigned char
```

2.3.3 EnterParamBytes

This macro writes the supplied array of byte values into the macro parameter space, checking for overflow.

Declaration

```
#define EnterParamBytes(bytearray, nbytes)    { ... }
```

Call this macro with the start address of a byte array and an integer number of bytes. The buffer is checked for possible overflow and, if copying would overflow the buffer, the macro **return**s from the enclosing function with the error **IErr_NotEnoughMacroArgumentSpace**.

Preconditions

You must have called `InitParams` before calling any of these macros.

The data in the array must be known when the TAPOp macro is defined.

2.3.4 EnterParamU32, EnterParamU16, EnterParamU8

This macro writes the supplied 32-bit, 16-bit or 8-bit word into the macro parameter space, checking for overflow.

Declaration

```
#define EnterParamU32(wordvalue)     { ... }
#define EnterParamU16(halfwordvalue) { ... }
#define EnterParamU8(bytevalue)      { ... }
```

Call the macro with a word-sized parameter. It is copied to a local temporary variable, so the macro parameter can have side-effects. Between copying the parameter to the temporary and copying the temporary to the buffer, the parameter buffer is checked for overflow. If the copy would overflow the buffer, the macro **return**s from the enclosing function with the error **IErr_NotEnoughMacroArgumentSpace**.

Preconditions

You must have called `InitParams` before calling any of these macros.

The value of the parameter must be known when the TAPOp macro is defined.

Example

Create a macro that places the TAP controller in SCAN_N state.

```
#define MAC_SCAN     1 /* number must be unique to connection */
TAPOp_Error writeScanN()
{
    int      ValPtr;
    unsigned8 Values[MACRO_ARGUMENT_AREA_SIZE]; //default decl: in this instance
                                                //array size could be just 3.
    InitParams;
    EnterParamU16(SCAN_N);     //TDIbits
    EnterParamU8(0);           //TDIrev
    EnterParamU8(0);           //nClks
    TAPCheck(TAPOp_DefineMacro(cId, MAC_SCAN, "ARMTAP_AccessIR_nClks:123", 1,
                               Values, ValPtr));
    return TAPOp_NoError;
}
```

2.3.5 HAS_ONCRPC_BUILTIN

This macro defines whether the supplied ONC RPC library is used or whether the host native implementation of this library is used.

Declaration

```
#define HAS_ONCRPC_BUILTIN 1
```

The macro is defined if any of the symbols __hppa, __svr4__ or __SVR4__ is defined.

2.3.6 InitParams

This macro initializes the parameter insertion point, ValPtr, ready for another parameter list.

Declaration

```
#define InitParams    ValPtr = 0
```

Define the integer ValPtr in the scope of functions that use the TAPOp parameter macros. Call this macro before creating a new parameter list with the EnterParam*x* macro calls.

2.3.7 MACRO_ARGUMENT_AREA_SIZE

This macro controls the size of an **extern** array that is used to construct TAPOp macro definition argument lists. It is measured in bytes. The default size is large to enable macros to be constructed that download large quantities of data to target memory.

Declaration

```
#define MACRO_ARGUMENT_AREA_SIZE  131100
extern unsigned8 Values[MACRO_ARGUMENT_AREA_SIZE;
```

You must declare the variable `Values` in a suitable location in your source files.

2.3.8 MACRO_RESULT_SIZE

This macro controls the size of an **extern** array that is used to receive the results of running a TAPOp macro on the server. It is measured in bytes.

Declaration

```
#define MACRO_RESULT_SIZE   8192
extern unsigned8 Results[MACRO_RESULT_SIZE];
```

You must declare the variable `Results` in a suitable location in your source files.

2.3.9 NREnterParamBytes

This macro writes the supplied array of byte values into the macro parameter space, checking for overflow.

Declaration

```
#define NREnterParamBytes(bytearray, nbytes)    { ... }
```

Call this macro with the start address of a byte array and an integer number of bytes. The buffer is checked for possible overflow and, if copying would overflow the buffer, the macro prints an error message to the `stderr` stream by calling `fprintf()`.

2.3.10 NREnterParamU32, NREnterParamU16, NREnterParamU8

This macro writes the supplied 32-bit, 16-bit or 8-bit word into the macro parameter space, checking for overflow.

Declaration

```
#define NREnterParamU32(wordvalue)     { ... }
#define NREnterParamU16(halfwordvalue) { ... }
#define NREnterParamU8(bytevalue)      { ... }
```

Call the macro with a word-sized parameter. It is copied to a local temporary variable, so the macro parameter can have side-effects. Between copying the parameter to the temporary and copying the temporary to the buffer, the parameter buffer is checked for overflow. If the copy would overflow the buffer, the macro prints an error message to the `stderr` stream by calling `fprintf()`.

Preconditions

You must have called `InitParams` before calling any of these macros.

The value of the parameter must be known when the TAPOp macro is defined.

Example

Create a macro that places the TAP controller in `SCAN_N` state.

```
#define MAC_SCAN     1 /* number must be unique to connection */
int      ValPtr;
unsigned8 Values[MACRO_ARGUMENT_AREA_SIZE]; //default decl: in this instance
                                            //array size could be just 3.
InitParams;
NREnterParamU16(SCAN_N);        //TDIbits
NREnterParamU8(0);              //TDIrev
TAPOp_DefineMacro(cId, MAC_SCAN, "ARMTAP_AccessIR:12", 1, Values, ValPtr);
```

2.3.11 TAPCheck macro

This macro is wrapped around all TAPOp API calls so that a `TAPOp_UnableToSelect` error can be retried and so that other errors are detected and dealt with.

Note

The implementation of `TAPCheck` supplied with the source is only an example, and application requirements might require modifications to, for example, use an application specific error handler.

Declaration

```
#define TAPCheck(OPERATION) \
TAPOp_Error t_err; \
do { \
    unsigned8 flags; \
    TAPOp_Error t2_err; \
    t_err = OPERATION; \
    /* First see if all was OK */  \
    if (t_err == TAPOp_NoError) {\
        break; \
    }\
    /* Try again if we could not connect */  \
    if (t_err == TAPOp_UnableToSelect) continue; \
    /* Check for power being off or reset being asserted */ \
    t2_err=TAPOp_ReadMICEFlags(connectId, &flags); \
    if (t2_err == TAPOp_NoError) { \
        if (flags & TAPOp_FL_TargetPowerOffNow) \
            fprintf(stderr,"ERROR: Target power is off\n"); \
        else if (flags & TAPOp_FL_InResetNow) \
            fprintf(stderr,"ERROR: Target is in reset state (nSRST low)\n"); \
        else if (flags & TAPOp_FL_TargetPowerHasBeenOff) \
            fprintf(stderr,"ERROR: Target power has been switched off\n"); \
        else if (flags & TAPOp_FL_TargetHasBeenReset) \
            fprintf(stderr,"ERROR: Target has been reset\n"); \
        else \
            /* Otherwise power & reset ok - we have a specific TAPOp error */ \
            fprintf(stderr,"ERROR: TAP Operation call failed, errno=%d " \
                           "(line %i, file %s)\n", t_err, __LINE__, __FILE__); \
    } \
    else \
        /* Otherwise we have an unknown error - just report the error code */ \
        fprintf(stderr,"ERROR: TAP Operation call failed, errno=%d " \
                       "(line%i, file %s)\n", t_err, __LINE__, __FILE__); \
    do { \
        t_err=TAPOp_CloseConnection(connectId); \
    } while (t_err == TAPOp_UnableToSelect); \
    give_up(); \
} while (t_err == TAPOp_UnableToSelect); }
```

The required elements of this macro are:

1. The parameter to the macro is called as a function and the error return recorded.
2. If the error return code is the unable to select code, then the function call is retried.
3. If calling the function `TAPOp_ReadMICEFlags` returns success, the flags are tested to check for a target reset or power off as the cause of the initial error.

4. Errors that are detected cause the program to call an error handler (the function `give_up()` in the default implementation).

This macro can be simplified significantly if the error handling is moved into a C function (for example, called `TAPCheck_fail`) as shown below, although this implementation looses the file and line number error reporting of the original.

```
#define TAPCheck(OPERATION)  do {\
         unsigned c = 200; TAPOp_Error t_err; \
         while ( c-- > 0 && ( t_err = ( OPERATION )) == TAPOp_UnableToSelect) \
             Sleep(0);  \
         if (t_err != TAPOp_NoError) \
             return TAPCheck_fail(t_err); \
         } while(0)
```

This version also includes a call to `Sleep()` that enables the operating system scheduler to run another task when an error occurs. In a multitasking environment this might enable the task which has selected the interface to complete without waiting for our task to reach the end of its timeslice.

Usage

A macro that detects the `TAPOp_UnableToSelect` error and retries the operation must be wrapped around every TAPOp API call that connects to the Multi-ICE server. The default macro never returns this error to the environment, and so it is not necessary to wrap any other function in the same way.

2.3.12 TAPOp_Error

This is an enumeration of the error codes for the TAPOp API functions listed in *TAPOp function alphabetic reference* on page 2-26.

Note

If a TAPOp call returns an error code other than `TAPOp_NoError` or `TAPOp_UnableToSelect`, the TAPOp connection is automatically deselected. This is so that a failing client does not block the Multi-ICE server.

Declaration

```
typedef enum TAPOp_Error
```

where the enumerators are:

`TAPOp_NoError` No error. The operation completed successfully.

`TAPOp_OutOfStore` Ran out of memory. This is a serious error.

`TAPOp_UnableToSelect`

Unable to select a connection for this TAP controllerbecause another TAP controller is being accessed. Try again later.

`TAPOp_TAPNotPresent`

The specified TAP controller is not present.

`TAPOp_NotInitialised`

The Multi-ICE server has not yet been configured.

`TAPOp_TooManyConnections`

The Multi-ICE server has no free connections.

`TAPOp_ClientsStillConnected`

The Multi-ICE server cannot finalize or reconfigure while clients are attached.

`TAPOp_NoSuchConnection`

Invalid connection ID was presented.

`TAPOp_InBadTAPState`

TAP controllers in unknown or incorrect state so Multi-ICE cannot perform the request.

`TAPOp_BadParameter`

Invalid parameter value specified.

`TAPOp_ConnectionStillSelected`

Connection still connected.

`TAPOp_IRSCTooLong` The combined length of all IRs is too great for this version of Multi-ICE.

`TAPOp_SCSRTooLong` One or more of the Scan Chain Select registers is too long.

`TAPOp_ScanChainAlreadyClaimed`

Connection cannot be opened because one of the required scan chains is claimed by another connection.

`TAPOp_BadConfigurationData`

The configuration data is unsuitable for this implementation.

`TAPOp_DriverLimitExceeded`

On a call to `TAPOp_GetDriverDetails`, the array size allocation was not big enough to return details of all the drivers found by Multi-ICE.

`TAPOp_UnknownDriverName`

On a call to `TAPOp_OpenConnection`, the driver name was not one that was passed back to the client by `TAPOp_GetDriverDetails`.

`TAPOp_CouldNotOpenPort`

Could not open the requested port.

`TAPOp_ParameterConflicts`

A parameter conflicts with the configuration data.

`TAPOp_RPC_Connection_Fail`

RPC connection failure during a call.

`TAPOp_UndefinedMacro`

Tried to run a macro that has not been defined.

`TAPOp_TooManyMacros`

Tried to create more macros than `MAX_MACROS` allows.

`TAPOp_TooManyMacroLines`

Tried to add more lines to a macro than `MAX_MACRO_LINES` allows.

`TAPOp_BadFixedParamNo`

When defining a macro line, reference to a nonexistent parameter was given in the list of fixed parameters.

`TAPOp_OutOfMacroResultSpace`

Ran out of macro result space.

`TAPOp_MaskAndTestFailed`

The Mask and Test operation did not match.

`TAPOp_NotAllocatedToThisConnection`

Resource is not allocated to this connection.

`TAPOp_CannotEnableLogging`

The log file is not set up.

`TAPOp_TooManyProcessors`

Maximum number of processors has been exceeded.

`TAPOp_IncompatibleModel`

The hardware connected is not the correct model version.

`TAPOp_CouldNotOpenMulFile`

A `.mul` file cannot be opened.

`TAPOp_BadlyFormattedMulFile`

A `.mul` file is corrupt.

`TAPOp_AnySeqUsedBadPath`

An Exit2-IR/DR -> Shift-IR/DR transition was requested.

`TAPOp_AnySeqWrongIRLength`

The wrong number of Shift-IR **TCK**s was detected.

`TAPOp_UnknownProcedureName`

Unknown procedure name in `TAPOp_DefineMacro` call.

`TAPOp_CantUseProcInMacro`

Procedure specified in `TAPOp_DefineMacro` call cannot be run in a macro.

`TAPOp_CouldNotBuildCompleteParameterList`

While attempting to run a macro, the server was unable to build a complete parameter list from the parameters supplied when it was defined and when it was executed.

`TAPOp_HardwareNotLicensedForSystem`

An attempt was made to use unlicensed hardware.

`TAPOp_ServerTooOld` A more recent version of the Multi-ICE server is required for this operation.

`TAPOp_WaitTooLong` A period of more than one second was passed to `TAPOp_Wait`.

`TAPOp_MultiICEHWNotPoweredUp`

The server cannot connect to the Multi-ICE hardware. It is possibly not powered up.

`TAPOp_ParallelInterfaceTimeout`

Parallel port interface timeout occurred.

2.3.13 WAIT_FOR_FIRST_CONNECTION

This constant is the initial connection timeout for the RPC communications subsystem.

Declaration

```
#define WAIT_FOR_FIRST_CONNECTION 10
```

The macro is only defined in the file `rpcclient.c`. This timeout is used when a connection attempt is being made to the Multi-ICE server, and is shorter than the normal timeout so that connection attempts to machines that are not running a server are rejected quickly.

2.3.14 WAIT_FOR_NORMAL_OPERATIONS

This constant is the initial packet timeout for the RPC communications subsystem.

Declaration

```
#define WAIT_FOR_NORMAL_OPERATIONS 66
```

The macro is only defined in the supplied C source file `rpcclient.c`. You compile this file and the other TAPOp library files and link it with your application to create the TAPOp client library.

2.4 TAPOp inter-client communication flags

This section describes the flags that are maintained on the Multi-ICE server for the benefit of the Multi-ICE clients. It contains the following sections:

- *Private flags*
- *Flags used by TAPOp_SetControlMacros* on page 2-24
- *Flags returned by TAPOp_ReadMICEFlags* on page 2-25.

2.4.1 Private flags

These flags are private to each TAP controller (processor).

Declaration

```
#define TAPOp_ProcRunning               0x1
#define TAPOp_ProcHasStopped            0x2
#define TAPOp_ProcStoppedByServer       0x4
#define TAPOp_DownloadingCode           0x8
#define TAPOp_ProcStartREQ             0x10
#define TAPOp_ProcStartACK             0x20
#define TAPOp_UserWantsSyncStart       0x40
#define TAPOp_UserWantsSyncStop        0x80
#define TAPOp_AcknowledgeReset        0x100
#define TAPOp_PendingServerStop       0x200
#define TAPOp_PendingServerACK        0x400
#define TAPOp_TestLogicReset          0x800
#define TAPOp_TestLogicResetACK      0x1000
```

where:

`TAPOp_ProcRunning`

A TAPOp client must set this flag when the processor starts executing code. It must be cleared when the processor halts. This can be used by the Multi-ICE server to indicate whether or not other processors must be stopped, according to your requirements. Setting this bit causes the processor state display to change to `R`. This is a write-only flag for the TAPOp client.

`TAPOp_ProcHasStopped`, `TAPOp_ProcStoppedByServer`

These two flags are used to determine if and why a processor has stopped. A client must poll the `TAPOp_ProcHasStopped` flag when the processor is running. If it is set, then if:

`TAPOp_ProcStoppedByServer` **is set**

The Multi-ICE server has stopped the processor because another processor has stopped and a synchronized stop condition was set up.

`TAPOp_ProcStoppedByServer` **is not set**

The processor has stopped of its own accord, for example, because it hit a breakpoint.

While `TAPOp_ProcHasStopped` is not set, then the value of `TAPOp_ProcStoppedByServer` is not defined.

These flags are read-only for a client.

`TAPOp_DownloadingCode`

A debugger must set this flag immediately before starting to download code to the target processor. This enables you to set an output bit when this occurs. This can be useful on a system that can switch between very slow and very fast clocks, because fast clocking speeds up download considerably. Similarly, when the download has completed, this bit must be cleared.

Setting this bit causes the processor state display to change to `D`. If the checkbox **Set on Download** has been checked for user output bit 1, setting the downloading code flag also asserts user output bit 1. This flag is read-only for the server.

`TAPOp_ProcStartREQ`, `TAPOp_ProcStartACK`

These two flags control synchronized starting of processors. If `TAPOp_UserWantsSyncStart` is set, the debugger must set `TAPOp_ProcStartREQ` to request the server to start the processor.

When all the debuggers have set their `TAPOp_ProcStartREQ` flags, the server starts all processors together, and sets the `TAPOp_ProcStartACK` flag. `TAPOp_ProcStartREQ` is read-only for the server. `TAPOp_ProcStartACK` is read-only for a client.

`TAPOp_UserWantsSyncStart`, `TAPOp_UserWantsSyncStop`

These two flags are read-only, and are set by the server if you have selected synchronous start or stop from the dialog. These flags are read-only for a client.

`TAPOp_PendingServerStop`, `TAPOp_PendingServerACK`

These flags are never looked at by the client. They allow the server to keep track of synchronous stop events that occur during the synchronous start acknowledgement sequence.

`TAPOp_TestLogicReset`, `TAPOp_TestLogicResetACK`

These two flags signal and acknowledge a Test Logic Reset occurring on a connection as a result of the actions of another connection. If a connection to a TAP controller causes the TAP controllers to leave `Test-Logic Reset`, the Multi-ICE server sets the `TAPOp_TestLogicReset` flag for every other connection. Most TAPOp calls made to these other connections fail, returning `TAPOp_InBadTAPState` until the `TAPOp_TestLogicResetACK` is set for that connection.

2.4.2 Flags used by TAPOp_SetControlMacros

These flags are used by `TAPOp_SetControlMacros`:

Declaration

```
#define TAPOp_SyncStopSupported       0x1
#define TAPOp_SyncStartSupported      0x2
#define TAPOp_PreExecMacroUsed        0x4
#define TAPOp_PostExecMacroUsed       0x8
```

where:

`TAPOp_SyncStopSupported`

Set this flag to indicate that the client supports synchronized stopping. The server then runs the event macro periodically, and events cause the stop macros to be run as described in *TAPOp_SetControlMacros* on page 2-120.

Only set this flag if the client has defined a suitable event macro, stop macro, `eventMask`, and `eventXOR`.

`TAPOp_SyncStartSupported`

Set this flag to indicate that the client supports synchronized starting. The server then waits for the `TAPOp_ProcStartREQ` private flag to be asserted before starting the processor using the execute macro.

Only set this flag if the client has defined a suitable execute macro.

`TAPOp_PreExecMacroUsed`

Set this flag if a `PreExec` macro is required.

`TAPOp_PostExecMacroUsed`

Set this flag if a `PostExec` macro is required.

2.4.3 Flags returned by TAPOp_ReadMICEFlags

These flags are returned by `TAPOp_ReadMICEFlags`.

Declaration

```
#define TAPOp_FL_TargetPowerHasBeenOff   0x1
#define TAPOp_FL_TargetPowerOffNow       0x2
#define TAPOp_FL_UserOut1                0x4
#define TAPOp_FL_UserIn1                 0x8
#define TAPOp_FL_UserIn2                0x10
#define TAPOp_FL_TargetHasBeenReset     0x20
#define TAPOp_FL_InResetNow             0x40
#define TAPOp_FL_UserOut2               0x80
```

`TAPOp_FL_TargetPowerOffNow`

The target power is off. This is an error condition.

`TAPOp_FL_TargetPowerHasBeenOff`

The target power has been off since the last `TAPOp_OpenConnection` call was made.

`TAPOp_FL_InResetNow`

The target reset signal is currently asserted. This is an error condition.

`TAPOp_FL_TargetHasBeenReset`

The target has been reset since the last `TAPOp_OpenConnection` call was made. This is usually an error condition.

`TAPOp_FL_UserIn1`

The state of the user-defined input signal 1 from Multi-ICE.

`TAPOp_FL_UserIn2`

The state of the user-defined input signal 2 from Multi-ICE.

`TAPOp_FL_UserOut1`

Current state of user-defined output 1 from Multi-ICE.

`TAPOp_FL_UserOut2`

Current state of user-defined output 2 from Multi-ICE.

2.5 TAPOp function alphabetic reference

This section lists all available TAPOp functions. The prototypes for these functions are held in the file named in the usage section.

The TAPOp functions described are:

2.5.1 ARMTAP_AccessDR_1Clk_W

Write a value to the data register (scan chain) of an ARM TAP controller and clock the ARM core once.

Note

This function is deprecated. The first use by a connection causes a warning message to appear in the Multi-ICE server log window. It might not be supported in future releases. Use `ARMTAP_AccessDR_W` instead.

Syntax

```
#include “tapop.h”

extern TAPOp_ErrorARMTAP_AccessDR_1Clk_W(unsigned8 connectId,
                  ScanData40 *TDIbits, unsigned8 TDIrev, unsigned8 len,
                  unsigned8 WRoffset, ScanData40 *WRmask, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

TDIbits A pointer to up to 40 bits of data to be written to the scan chain data register. This pointer must not be NULL.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of `TDIbits` first (normal mode).

TRUE Write MSB of `TDIbits` first (reversed mode).

To write less than 40 bits of data using reversed mode you must set `WRmask` with bits set in the high-order end of the mask, not the low-order end.

len The length of the selected data register (scan chain).

WRoffset Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated so that whatever data appears on **TDO** is put back into **TDI** during the scan.

To write less than 40 bits, specify zero here and use `WRmask`.

WRmask A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan:

- if 1 then the bit is written
- if 0 then the bit is recirculated, so that whatever comes out of the **TDO** pin is put back into **TDI** during the scan.

To write all 40 bits, you can set WRmask to NULL when using direct TAPOp calls. When defining macros, you must always specify the mask in full.

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

TAPOp_NoError

No error.

TAPOp_UnableToSelect

Connection could not be made. You must try again later.

TAPOp_NoSuchConnection

The connectId was not recognized.

TAPOp_InBadTAPState

The TAP controller was reset or is not in Select-DR-Scan.

TAPOp_BadParameter

Failed because of one of the following:

- TDIbits = NULL
- len = 0
- WRoffset >= len.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

The data to write to the core is read from TDIbits and sent with the write mask and offset to the Multi-ICE interface unit. It is transferred to the scan chain that was selected by the last call to ARMTAP_AccessIR_nClks. The TAP state machine is then passed through the state Run-Test/Idle, which causes the JTAG logic in the processor to generate a single DCLK for the CPU. This causes the data in the scan chain to be read into the core.

The TAP controller must be in JTAG state Select-DR-Scan before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, or TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to ARMTAP_AccessDR_1Clk_W are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *ARMTAP_AccessDR_W* on page 2-42
- *ARMTAP_AccessIR_nClks* on page 2-49
- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_RunMacro* on page 2-113.

2.5.2 ARMTAP_AccessDR_NoClk_W

Write a value to the data register (scan chain) of an ARM TAP controller.

Note

This function is deprecated. The first use by a connection causes a warning message to appear in the Multi-ICE server log window. It might not be supported in future releases. Use `ARMTAP_AccessDR_W` instead.

Syntax

```
#include “tapop.h”

extern TAPOp_ErrorARMTAP_AccessDR_NoClk_W(unsigned8 connectId,
                  ScanData40 *TDIbits, unsigned8 TDIrev, unsigned8 len,
                  unsigned8 WRoffset, ScanData40 *WRmask, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

TDIbits A pointer to up to 40 bits of data to be written to the scan chain data register. This pointer must not be NULL.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of `TDIbits` first (normal mode).

TRUE Write MSB of `TDIbits` first (reversed mode).

To write less than 40 bits of data using reversed mode you must set `WRmask` with bits set in the high-order end of the mask, not the low-order end.

len The length of the selected data register (scan chain).

WRoffset Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated so that whatever data appears on **TDO** is put back into **TDI** during the scan.

To write less than 40 bits, specify zero here and use `WRmask`.

WRmask A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan:

- if 1 then the bit is written
- if 0 then the bit is recirculated, so that whatever comes out of the **TDO** pin is put back into **TDI** during the scan.

To write all 40 bits, you can set WRmask to NULL when using direct TAPOp calls. When defining macros, you must always specify the mask in full.

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

TAPOp_NoError

No error.

TAPOp_UnableToSelect

Connection could not be made. You must try again later.

TAPOp_NoSuchConnection

The connectId was not recognized.

TAPOp_InBadTAPState

The TAP controller was reset or is not in Select-DR-Scan.

TAPOp_BadParameter

Failed because of one of the following:

- TDIbits = NULL
- len = 0
- WRoffset >= len.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

The data to write to the core is read from TDIbits and sent with the write mask and offset to the Multi-ICE interface unit. It is transferred to the scan chain that was selected by the last call to ARMTAP_AccessIR_nClks. The data written to the scan chain is not clocked into the core by this call.

The TAP controller must be in JTAG state Select-DR-Scan before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, or TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to ARMTAP_AccessDR_NoClk_W are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *ARMTAP_AccessDR_W* on page 2-42
- *ARMTAP_AccessIR_nClks* on page 2-49
- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_RunMacro* on page 2-113.

2.5.3 ARMTAP_AccessDR_RW

Writes and reads data to and from a TAP data register (scan chain) and then clock the ARM core `nClks` times.

Syntax

```
#include "tapop.h"

extern TAPOp_Error ARMTAP_AccessDR_RW(unsigned8 connectId, ScanData40 *TDIbits,
                  unsigned8 TDIrev, ScanData40 *TDObits, unsigned8 TDOrev,
                  unsigned8 len, unsigned8 WRoffset, ScanData40 *WRmask,
                  unsigned8 RDoffset, unsigned8 nClks, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

TDIbits A pointer to up to 40 bits of data to be writtento the scan chain data register. This must not be NULL.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of `TDIbits` first (normal mode).

TRUE Write MSB of `TDIbits` first (reversed mode).

To write less than 40 bits of data using reversed mode you must set `WRmask` with bits set in the high-order end of the mask, not the low-order end.

TDObits A pointer to a 40-bit buffer that receives the data that is read from the scan chain data register. This must not be NULL.

TDOrev A flag determining the bit order of the data:

FALSE Read data into LSB of `TDObits` first (normal mode).

TRUE Read data into MSB of `TDObits` first (reversed mode).

To read less than 40 bits of data using reversed mode you must mask the returned `TDObits` with bits set in the high-order end of the mask, not the low-order end.

len Length of the selected data register (scan chain).

WRoffset Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated so that whatever data appears on **TDO** is written to **TDI** during the scan.

To write all 40 bits, you can set `WRmask` to NULL when using direct TAPOp calls. When defining macros, you must always specify the mask in full.

WRmask A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan:

- if 1 then the bit is written
- if 0 then the bit is recirculated, so that whatever comes out of the **TDO** pin is put back into **TDI** during the scan.

To write all 40 bits, you can set `WRmask` to NULL when using direct TAPOp calls.

RDoffset Offset of the selected data register (scan chain) to start reading data from.

nClks The number of ARM processor **DCLK**s to be generated. This is the number of transitions out of `Run-Test/Idle` that are made. If `0`, `Run-Test/Idle` is avoided altogether. The value of `nClks` is range checked:

0 <= `nClks` <= 31

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made. You must try again later.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Select-DR-Scan`.

`TAPOp_BadParameter`

Failed because of one of the following:

- `TDIbits` = `NULL`
- `TDObits` = `NULL`
- `len` = 0
- `WRoffset` >= `len`

- RDoffset >= len
- nClks > 31 or nClks < 0.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

The data to write to the core is read from TDIbits and sent with the write mask and offset to the Multi-ICE interface unit. It is transferred to the scan chain that was selected by the last call to ARMTAP_AccessIR_nClks. The data written to the scan chain is clocked into the core if the value of nClks is non-zero, and the previous value of the register is shifted out of the scan chain and stored in TDObits.

The TAP controller must be in Select-DR-Scan state before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, or TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to ARMTAP_AccessDR_RW are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

Example

This example is taken from example1.c in the source/examples subdirectory of the Multi-ICE installation. It returns the value in the IDcode register of an ARM processor system coprocessor.

```
TAPOp_Error idcode(unsigned32 *idcode)
{
    ScanData40  dummyTDIdata, TDOdata;
    /* First put the IDCODE instruction in the IR */
    TAPCheck(ARMTAP_AccessIR_nClks(connectId,  /* The connection ID we got from TAPOp_OpenConnection */
                                   IDCODE,     /* Put the IDCODE instruction in the IR */
                                   0,          /* The IR is not reversed */
                                   0,          /* No clock ticks */
                                   1));        /* Deselect. We have completed this operation and another
                                                * client can access the server without disturbing the
                                                * state of our session. */
    /* Now read the IDCODE scan chain */
    TAPCheck(ARMTAP_AccessDR_RW(connectId,     /* The connection ID we got from TAPOp_OpenConnection */
```



```
                                &dummyTDIdata, /* This is a read-only register therefore we don't care
                                                * about the data we put into it.
                                                */
                                0,             /* The register is not reversed for writing */
                                &TDOdata,      /* Put the result here (what comes out of TDO) */
                                0,             /* The register is not reversed for reading */
                                32,            /* ID code scan chain is 32 bits long */
                                0,             /* Start writing at bit 0 */
                                NULL,          /* Write all the bits */
                                0,             /* Start reading at bit 0 */
                                0,             /* Don't issue any processor clocks to the ARM */
                                1));           /* Deselect (as described above) */
    /* The bottom 32 bits of the 40-bit scan data structure contain the 32 bit value we need */
    *idcode = TDOdata.low32;
    return TAPOp_NoError;
}
```

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW_And_Test* on page 2-38
- *ARMTAP_AccessDR_W* on page 2-42
- *ARMTAP_AccessIR_nClks* on page 2-49
- *TAPOp_AccessDR_RW* on page 2-56
- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_RunMacro* on page 2-113.

2.5.4 ARMTAP_AccessDR_RW_And_Test

Writes and reads data to and from a TAP data register (scan chain) and test the result until either the data read from the register matches a pattern, or a specified maximum number of tries is exceeded.

Syntax

```
#include “tapop.h”

extern TAPOp_Error ARMTAP_AccessDR_RW_And_Test(unsigned8 connectId,
                   ScanData40 *TDIbits, unsigned8 TDIrev, unsigned8 TDOrev,
                   unsigned8 len, ScanData40 *maskBits, ScanData40 *resBits,
                   unsigned8 WRoffset, ScanData40 *WRmask, unsigned8 RDoffset,
                   unsigned32 nTries, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by TAPOp_OpenConnection.

TDIbits A pointer to up to 40 bits of data to be writtento the scan chain data register. This must not be NULL.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of TDIbits first (normal mode).

TRUE Write MSB of TDIbits first (reversed mode).

To write less than 40 bits of data using reversed mode you must set WRmask with bits set in the high-order end of the mask, not the low-order end.

TDOrev A flag determining the bit order of the data:

FALSE Read data into LSB of TDObits first (normal mode).

TRUE Read data into MSB of TDObits first (reversed mode).

To read less than 40 bits of data using reversed mode you must mask the returned TDObits with bits set in the high-order end of the mask, not the low-order end.

len Length of the selected data register (scan chain).

maskBits A pointer to up to 40 bits of data used as a mask for the data read from the device.

resBits A pointer to up to 40 bits of data used as a match pattern for the data read from the device.

WRoffset Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated so that whatever data appears on **TDO** is written to **TDI** during the scan.

To write all 40 bits, you can set `WRmask` to NULL when using direct TAPOp calls. When defining macros, you must always specify the mask in full.

WRmask A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan:

- if 1 then the bit is written
- if 0 then the bit is recirculated, so that whatever comes out of the **TDO** pin is put back into **TDI** during the scan.

To write all 40 bits, you can set `WRmask` to NULL when using direct TAPOp calls.

RDoffset Offset of the selected data register (scan chain) to start reading data from.

nTries The number of unsuccessful attempts allowed before returning an error.

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made. You must try again later.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Select-DR-Scan`.

`TAPOp_BadParameter`

Failed because of one of the following conditions was TRUE:

- `TDIbits` = `NULL`

- maskBits = NULL
- resBits = NULL
- ntries = 0
- len = 0
- WRoffset >= len
- RDoffset >= len.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

TAPOp_MaskAndTestFailed

After nTries, mask and test still do not match resBits.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

The operation performs the following algorithm within the Multi-ICE server:

```
ScanData40 TDObits;
bool match;
do {
    ARMTAP_AccessDR_RW(connectid, TDIBits, TDIrev, TDObits, len, WRoffset,
                       WRmask, RDoffset, 0, TRUE);
} while ((--nTries >= 0) || (match = ((TDObits & maskBits) != resBits)));
return (match) ? TAPOp_NoError : TAPOp_MaskAndTestFailed;
```

Data from the read from the scan chain that was selected by the last call to ARMTAP_AccessIR_nClks is masked with maskBits and compared with resBits. If a match is found, TAPOp_NoError is returned. If no match is found, the write and read is performed up to nTries times (in total) before returning TAPOp_MaskAndTestFailed.

The operation is indivisible, so once the connection is selected, it remains selected until a match is found or nTries reads have been performed. The TAP controller must be in Select-DR-Scan state before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, or TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to ARMTAP_AccessDR_RW_And_Test are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

Example

The example below shows how to request an ARM processor to enter debug state and then wait for the ARM EmbeddedICE logic to signal it. The algorithm is:

1. Select the EmbeddedICE logic scan chain.
2. Write the enter debug request to the EmbeddedICE logic control register.
3. Poll for the processor to signal debug state.

In the example, the structure pointer `TDefs` points to processor-dependent information.

```
#define INTEST  0xC
#define IB_Status  1
#define IBS_DbgAck  0x1
#define IBS_nMrq  0x8
ScanData40 statusBits, dbgStateBits;
statusBits.low32 = 0;
statusBits.high8 = IB_Status;
dbgStateBits.low32 = IBS_DbgAck | IBS_nMrq;
dbgStateBits.high8 = 0;
/* select the EmbeddedICE logic scan chain */
TAPCheck(ARMTAP_AccessIR_nClks(connectId, SCAN_N, 0, 0, FALSE));
TAPCheck(ARMTAP_AccessDR_W(connectId, SC_ICEbreaker, 0, TDefs->SCSRlen, 0,
                           TDefs->SCSRMask, 0, FALSE));
/* write the enter debug request */
TAPCheck(ARMTAP_AccessIR_nClks(connectId, INTEST, 0, 0, FALSE));
TAPCheck(ARMTAP_AccessDR_W(connectId, statusBits, TDefs->ICEBRev,
                           TDefs->ChainLengths[SC_ICEbreaker], 0,
                           TDefs->ICEBMask, 0, FALSE));
/* poll for the processor */
TAPCheck(ARMTAP_AccessDR_W_And_Test(connectId, statusBits, TDefs->ICEBRev,
                           TDefs->ICEBRev, TDefs->ChainLengths[SC_ICEbreaker],
                           dbgStateBits, dbgStateBits, 0, TDefs->ICEBMask,
                           0, NUM_TRIES, FALSE));
```

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *ARMTAP_AccessDR_W* on page 2-42
- *ARMTAP_AccessIR_nClks* on page 2-49
- *TAPOp_AccessDR_RW* on page 2-56
- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_RunMacro* on page 2-113.

2.5.5 ARMTAP_AccessDR_W

Writes data to a TAP data register (scan chain) and then clock the ARM core `nClks` times.

Syntax

```
#include “tapop.h”

extern TAPOp_Error ARMTAP_AccessDR_W(unsigned8 connectId, ScanData40 *TDIbits,
                   unsigned8 TDIrev, unsigned8 len, unsigned8 WRoffset,
                   ScanData40 *WRmask, unsigned8 nClks, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

TDIbits A pointer to up to 40 bits of data to be writtento the scan chain data register. This must not be NULL.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of `TDIbits` first (normal mode).

TRUE Write MSB of `TDIbits` first (reversed mode).

To write less than 40 bits of data using reversed mode you must set `WRmask` with bits set in the high-order end of the mask, not the low-order end.

len Length of the selected data register (scan chain).

WRoffset Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated so that whatever data appears on **TDO** is written to **TDI** during the scan.

To write all 40 bits, you can set `WRmask` to NULL when using direct TAPOp calls. When defining macros, you must always specify the mask in full.

WRmask A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan:

- if 1 then the bit is written
- if 0 then the bit is recirculated, so that whatever comes out of the **TDO** pin is put back into **TDI** during the scan.

To write all 40 bits, you can set `WRmask` to NULL when using direct TAPOp calls.

nClks The number of ARM processor **DCLK**s to be generated. This is the number of transitions out of `Run-Test/Idle` that are made. If `0`, `Run-Test/Idle` is avoided altogether. The value of `nClks` is range checked:

0 <= `nClks` <= 31

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made. You must try again later.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Select-DR-Scan`.

`TAPOp_BadParameter`

Failed because of one of the following:

- `TDIbits` = NULL
- `len` = 0
- `WRoffset` >= `len`
- `nClks` > 31.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The data to write to the core is read from TDIbits and sent with the write mask and offset to the Multi-ICE interface unit. It is transferred to the scan chain that was selected by the last call to ARMTAP_AccessIR_nClks. If the value of nClks is nonzero, data written to the scan chain is clocked into the core with one or more cycles of **DCLK**. Data shifted out of the scan chain is always discarded.

The TAP controller must be in JTAG state Select-DR-Scan before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, or TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to ARMTAP_AccessDR_W are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *ARMTAP_AccessDR_RW_And_Test* on page 2-38
- *ARMTAP_AccessIR_nClks* on page 2-49
- *TAPOp_AccessDR_RW* on page 2-56
- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_RunMacro* on page 2-113.

2.5.6 ARMTAP_AccessIR

Writes data to the TAP controller instruction register (IR).

Note

This function is deprecated. The first use by a connection causes a warning message to appear in the Multi-ICE server log window. It might not be supported in future releases. Use `ARMTAP_AccessIR_nClks` instead.

Syntax

```
#include “tapop.h”

extern TAPOp_ErrorARMTAP_AccessIR(unsigned8 connectId, unsigned16 TDIbits,
                  unsigned8 TDIrev, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

TDIbits Up to 16 bits of data to be written to the TAP controller instruction register. This must not be NULL and you cannot mask the value.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of `TDIbits` first (normal mode).

TRUE Write MSB of `TDIbits` first (reversed mode).

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made. You must try again later.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

TAPOp_BadParameter

Failed because of one of the following:

- TDIbits = NULL
- len = 0
- WRoffset >= len
- nClks > 31.

TAPOp_InBadTAPState

The TAP controller was reset or is not in Select-DR-Scan.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

The data to write to the TAP controller is read from TDIbits and sent to the Multi-ICE interface unit. The IR length is part of the device configuration data. Refer to the TAP controller documentation for details on the IR instructions that are supported by your device.

The TAP does not go through Run-Test/Idle, so if an ARM data bus scan chain is selected no **DCLK** happens. The TAP controller must be in Select-DR-Scan state before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, or TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to ARMTAP_AccessIR are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *ARMTAP_AccessIR_nClks* on page 2-49
- *TAPOp_AccessIR* on page 2-62.

2.5.7 ARMTAP_AccessIR_1Clk

Writes data to the TAP controller instruction register (IR) and clocks an ARM processor core once.

Note

This function is deprecated. The first use by a connection causes a warning message to appear in the Multi-ICE server log window. It might not be supported in future releases. Use `ARMTAP_AccessIR_nClks` instead.

Syntax

```
#include "tapop.h"

extern TAPOp_Error ARMTAP_AccessIR_1Clk(unsigned8 connectId, unsigned16 TDIbits,
                   unsigned8 TDIrev, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

TDIbits Up to 16 bits of data to be written to the TAP controllerinstruction register. This must not be NULL and you cannot mask the value.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of `TDIbits` first (normal mode).

TRUE Write MSB of `TDIbits` first (reversed mode).

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

TAPOp_NoSuchConnection

The connectId was not recognized.

TAPOp_InBadTAPState

The TAP controller was reset or is not in Select-DR-Scan.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

The data to write to the TAP controller is read from TDIbits and sent to the Multi-ICE interface unit. The IR length is part of the device configuration data. Refer to the TAP controller documentation for details on the IR instructions that are supported by your device.

The TAP controller is placed in INTEST and passed through the state Run-Test/Idle once to generate a clock pulse for the ARM processor core. The TAP controller must be in Select-DR-Scan state before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, or TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to ARMTAP_AccessIR_1Clk are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *ARMTAP_AccessIR_nClks* on page 2-49
- *TAPOp_AccessIR* on page 2-62.

2.5.8 ARMTAP_AccessIR_nClks

Writes data to the TAP controller instruction register (IR) and clocks an ARM processor core the specified number of times.

Note

The functions `ARMTAP_AccessIR` and `ARMTAP_AccessIR_1Clk` are deprecated. The first use of either by a connection causes a warning message to appear in the Multi-ICE server log window and they might not be supported in future releases. Use this function instead.

Syntax

```
#include “tapop.h”

extern TAPOp_Error ARMTAP_AccessIR_nClks(unsigned8 connectId,
                   unsigned16 TDIbits, unsigned8 TDIrev, unsigned8 nClks,
                   unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

TDIbits Up to 16 bits of data to be written to the TAP controllerinstruction register. This must not be NULL and you cannot mask the value.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of `TDIbits` first (normal mode).

TRUE Write MSB of `TDIbits` first (reversed mode).

nClks The number of ARM processor **DCLK**s to be generated. This is the number of transitions out of `Run-Test/Idle` that are made. If `0`, `Run-Test/Idle` is avoided altogether. The value of `nClks` is range checked:

0 <= `nClks` <= 31

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_BadParameter`

Failed because:

- `nClks` > `31`.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Select-DR-Scan`.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The data to write to the TAP controller is read from `TDIbits` and sent to the Multi-ICE interface unit. The IR length is part of the device configuration data. Refer to the TAP controller documentation for details on the IR instructions that are supported by your device.

The TAP controller is placed in `INTEST` and passed through the state `Run-Test/Idle` once to generate a clock pulse for the ARM processor core. The TAP controller must be in `Select-DR-Scan` state before the function is used, and is left in this state after the function has been performed.

The number of clocks to the ARM processor core can be specified as:

- zero (`0`), so the call behaves like the deprecated function `ARMTAP_AccessIR`
- one (`1`), so the call behaves like the deprecated function `ARMTAP_AccessIR_1Clk`

- more than one, and less than 31.

The length of the IR is already known by the Multi-ICE server as this is part of the configuration data for each processor.

If another connection on the same Multi-ICE server resets the TAP controllers by calling `TAPOp_AnySequence_W`, or `TAPOp_AnySequence_RW`, or `TAPOp_TestResetSignal`, all subsequent calls to `ARMTAP_AccessIR_nClks` are rejected with the error `TAPOp_InBadTAPState` until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

Example

The following example shows how to place an ARM scan chain in RESTART mode, that causes an ARM processor to resynchronize to the system clock:

```
#define RESTART 0x4
    ...
    /* write instr RESTART, not reversed, 1 clock */
    TAPCheck(ARMTAP_AccessIR_nClks(connectId, RESTART, 0, 1, FALSE));
    ...
```

Compatibility

This function first appeared in Multi-ICE Version 2.1.

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *ARMTAP_AccessIR* on page 2-45
- *TAPOp_AccessIR* on page 2-62.

2.5.9 ARMTAP_ClockARM

Clock an ARM processor core a specified number of times.

Note

This function is deprecated. The first use by a connection causes a warning message to appear in the Multi-ICE server log window. It might not be supported in future releases. Typically, you can use the `nClks` parameter of another `ARMTAP` API function for the same effect.

Syntax

```
#include “tapop.h”

extern TAPOp_Error ARMTAP_ClockARM(unsigned8 connectId, unsigned8 nClks,
                   unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

nClks The number of ARM processor core clocks to generate:

0 <= `nClks` <= 31

deselect If 0, the connection to this TAP controller remainsselected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_BadParameter`

Failed because:

- `nClks` > 31.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Select-DR-Scan`.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

This function writes `INTEST` into the IR as it has to go through either the DR or IR states to get to `Run-Test/Idle`. It is usually more efficient to call one of the `ARMTAP_AccessDR` functions that combines **DCLK**s with the DR access. The TAP controller must be in `Select-DR-Scan` state before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling `TAPOp_AnySequence_W`, or `TAPOp_AnySequence_RW`, or `TAPOp_TestResetSignal`, all subsequent calls to `ARMTAP_ClockARM` are rejected with the error `TAPOp_InBadTAPState` until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *ARMTAP_AccessDR_W* on page 2-42
- *TAPOp_AnySequence_W* on page 2-68.

2.5.10 GetServerName

User supplied callback that must return the name or IP address of the Multi-ICE server workstation as a null-terminated string.

Syntax

```
#include “tapop.h”

extern int GetServerName(char **server)
```

where:

server A pointer to a buffer containing the name of themachine running the Multi-ICE server.

Return

The function returns:

0 No error.

Nonzero Failure (user-supplied values).

Usage

The function is called to query the name of the Multi-ICE server by the functions `TAPOp_RPC_Initialise` and `TAPOp_GetDriverDetails`. The supplied function must write through the `server` pointer the address of a buffer containing the name. The name is not copied into TAPOp-local storage.

The hostname is resolved using the standard library function `gethostbyname()`. For example it can be `pc25`, `itlilien.acme.org`, or `172.15.1.1`.

Note

If you have multiple connections managed by code in the same process address space, one definition of GetServerName must supply server name information for all connections.

Example

This is an example definition of the function:

```
static char servername[] = “itlilien.acme.org”;
int GetServerName(char **server)
{
```



```
    *server = servername;
    return 0;
}
```

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_GetDriverDetails* on page 2-88
- *TAPOp_OpenConnection* on page 2-94
- *TAPOp_RPC_Initialise* on page 2-106.

2.5.11 TAPOp_AccessDR_RW

Writes data to a TAP data register (scan chain) and read the previous value.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_AccessDR_RW(unsigned8 connectId, ScanData40 *TDIbits,
                  unsigned8 TDIrev, ScanData40 *TDObits, unsigned8 TDOrev,
                  unsigned8 len, unsigned8 WRoffset, ScanData40 *WRmask,
                  unsigned8 RDoffset, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by TAPOp_OpenConnection.

TDIbits A pointer to up to 40 bits of data to be writtento the scan chain data register. This must not be NULL.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of TDIbits first (normal mode).

TRUE Write MSB of TDIbits first (reversed mode).

To write less than 40 bits of data using reversed mode you must set WRmask with bits set in the high-order end of the mask, not the low-order end.

TDObits A pointer to a 40-bit buffer that receives the data that is read from the scan chain data register. This must not be NULL.

TDOrev A flag determining the bit order of the data:

FALSE Read data into LSB of TDObits first (normal mode).

TRUE Read data into MSB of TDObits first (reversed mode).

To read less than 40 bits of data using reversed mode you must mask the returned TDObits with bits set in the high-order end of the mask, not the low-order end.

len Length of the selected data register (scan chain).

WRoffset Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated so that whatever data appears on **TDO** is written to **TDI** during the scan.

To write all 40 bits, you can set WRmask to NULL when using direct TAPOp calls. When defining macros, you must always specify the mask in full.

WRmask A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan:

- if 1 then the bit is written
- if 0 then the bit is recirculated, so that whatever comes out of the **TDO** pin is put back into **TDI** during the scan.

To write all 40 bits, you can set `WRmask` to NULL when using direct TAPOp calls.

RDoffset Offset of the selected data register (scan chain) where you start reading data.

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Run-Test/Idle`.

`TAPOp_BadParameter`

Failed for one of the following reasons:

- `TDIbits` = NULL
- `TDObits` = NULL
- `len` = 0
- `RDoffset` >= `len`
- `WRoffset` >= `len`.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The data to write to the core is read from `TDIbits` and sent with the write mask and offset to the Multi-ICE interface unit. It is transferred to the scan chain that was selected by the last call to `TAPOp_AccessIR`. The previous value of the register is shifted out of the scan chain and stored in `TDObits`.

The TAP controller must be in `Run-Test/Idle` state before this function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling `TAPOp_AnySequence_W`, or `TAPOp_AnySequence_RW`, or `TAPOp_TestResetSignal`, all subsequent calls to `TAPOp_AccessDR_RW` are rejected with the error `TAPOp_InBadTAPState` until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessDR_RW* on page 2-34
- *TAPOp_AccessDR_W* on page 2-59
- *TAPOp_AnySequence_RW* on page 2-65.

2.5.12 TAPOp_AccessDR_W

Writes data to a TAP data register (scan chain).

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_AccessDR_W(unsigned8 connectId, ScanData40 *TDIbits,
                   unsigned8 TDIrev, unsigned8 len, unsigned8 WRoffset,
                   ScanData40 *WRmask, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by TAPOp_OpenConnection.

TDIbits A pointer to up to 40 bits of data to be writtento the scan chain data register. This must not be NULL.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of TDIbits first (normal mode).

TRUE Write MSB of TDIbits first (reversed mode).

To write less than 40 bits of data using reversed mode you must set WRmask with bits set in the high-order end of the mask, not the low-order end.

len Length of the selected data register (scan chain).

WRoffset Offset of the selected data register (scan chain) where you start writing data. The rest of the scan chain is recirculated so that whatever data appears on **TDO** is written to **TDI** during the scan.

To write all 40 bits, you can set WRmask to NULL when using direct TAPOp calls. When defining macros, you must always specify the mask in full.

WRmask A 40-bit mask that determines which bits are written to the data register (scan chain) during a scan:

- if 1 then the bit is written
- if 0 then the bit is recirculated, so that whatever comes out of the **TDO** pin is put back into **TDI** during the scan.

To write all 40 bits, you can set WRmask to NULL when using direct TAPOp calls.

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Run-Test/Idle`.

`TAPOp_BadParameter`

Failed for one of the following reasons:

- `TDIbit` = `NULL`
- `WRoffset` >= `len`.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The data to write to the core is read from `TDIbits` and sent with the write mask and offset to the Multi-ICE interface unit. It is transferred to the scan chain that was selected by the last call to `TAPOp_AccessIR`. Data shifted out of the scan chain is discarded.

The TAP controller must be in `Run-Test/Idle` state before this function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling `TAPOp_AnySequence_W`, or `TAPOp_AnySequence_RW`, or `TAPOp_TestResetSignal`, all subsequent calls to `TAPOp_AccessDR_W` are rejected with the error `TAPOp_InBadTAPState` until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_AccessDR_RW* on page 2-56
- *TAPOp_AccessIR* on page 2-62
- *TAPOp_AnySequence_RW* on page 2-65.

2.5.13 TAPOp_AccessIR

Writes data to the TAP IR. Data coming out of **TDO** is discarded.

The length of the IR is already known by the Multi-ICE server as this is part of the configuration data for each processor.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_AccessIR(unsigned8 connectId, unsigned16 TDIbits,
                  unsigned8 TDIrev, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

TDIbits Up to 16 bits of data to be written to the TAP controllerinstruction register. This must not be NULL and you cannot mask the value.

TDIrev A flag determining the bit order of the data:

FALSE Write LSB of `TDIbits` first (normal mode).

TRUE Write MSB of `TDIbits` first (reversed mode).

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Run-Test/Idle`.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The data to write to the TAP controller is read from `TDIbits` and sent to the Multi-ICE interface unit. The IR length is part of the device configuration data. Refer to the TAP controller documentation for details on the IR instructions that are supported by your device.

The TAP controller must be in `Run-Test/Idle` state before the function is used, and is left in this state after the function has been performed.

If another connection on the same Multi-ICE server resets the TAP controllers by calling `TAPOp_AnySequence_W`, or `TAPOp_AnySequence_RW`, or `TAPOp_TestResetSignal`, all subsequent calls to `TAPOp_AccessIR` are rejected with the error `TAPOp_InBadTAPState` until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

Example

This example shows a flash memory device being put into `IDCODE` state. It is correct to use the `TAPOp_` variants here because the idle state for the flash memory is `Run-Test/Idle`, not the `Select-DR-Scan` used for ARM processors.

In this example it is correct to pass through `Test-Logic Reset` because the purpose of this code is to reset the controllers. However you must be careful when programming state changes because if you take any one TAP controller through `Test-Logic Reset` you reset every TAP controller.

```
{
    ... declarations
    /* Reset the TAP controller by passing through Test-Logic Reset. */
    /* Then enter Run-Test-Idle */
    TMSbits = 0x1F;
    TDIbits = 0x00;
    TAPCheck(TAPOp_AnySequence_W(connectId, 6, &TDIbits, &TMSbits, 0));
    /* Load in IDCODE instruction */
    TAPCheck(TAPOp_AccessIR(connectId, 0x6, 0, 1));
```

See also

These TAPOp API functions provide similar or related functionality:

- *ARMTAP_AccessIR_nClks* on page 2-49
- *TAPOp_AnySequence_W* on page 2-68.

2.5.14 TAPOp_AnySequence_RW

Perform a sequence of **TCK**s, with explicitly specified **TMS** and **TDI** inputs. **TDO** values are read out.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_AnySequence_RW(unsigned8 connectId, unsigned8 numTCKs,
                   unsigned32 *TDIbits, unsigned32 *TMSbits,
                   unsigned32 *TDObits, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

numTCKs The number of **TCK**s to perform.

0 <= `numTCKs` <= 255

TDIbits A pointer to up to 256 bits of TDI data to be writtento the TAP controller. This pointer must not be NULL.

TMSbits A pointer to up to 256 bits of TMS data to be written to the TAP controller. This pointer must not be NULL.

TDObits A pointer to a buffer of up to 256 bits to store TDO data from the TAP controller. This pointer must not be NULL.

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

TAPOp_InBadTAPState

The TAP controller was reset or is not in Select-DR-Scan or Run-Test/Idle.

TAPOp_BadParameter

Failed for one of the following reasons:

- TDIbits = NULL
- TDObits = NULL
- TMSbits = NULL
- numTCKs = 0.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

TAPOp_AnySeqUsedBadPath

An Exit2-IR/DR to Shift-IR or Shift-DR transition was requested.

TAPOp_AnySeqWrongIRLength

The wrong number of Shift-IR **TCKs** was detected.

Usage

This function provides an interface to the virtual TAP controller in the Multi-ICE server. If there are actually several TAP controllers connected in series, the server transparently inserts extra bits required into the IR and DR scan chains. For this to work, however, some restrictions are imposed on what sequences can be performed. These restrictions are described in *Restrictions* on page 2-67.

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, or TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to TAPOp_AnySequence_RW are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. See *TAPOp_AnySequence_W* on page 2-68 for more details.

Note

When the connection is deselected, the TAP controller must be left in one of the states Select-DR-Scan or Run-Test/Idle. If this is not done, an error results.

Restrictions

The following restrictions apply:

- When shifting data into an IR or DR register, all the shifts (performed when the TAP controller start state is `Shift-IR` or `Shift-DR`) must occur together. After performing these shifts, `Shift-IR` or `Shift-DR` must not be left and then re-entered without going through `Capture-IR/DR`. If you do this, the server returns an error indicating that the transition `Exit2-IR/DR` to `Shift-IR` or `Shift-DR` was requested. You must code the client to avoid this case.
- The Multi-ICE server knows the length of the IR register. If the number of shifts into this register is not the same as the length according to the server, the server ignores the data shifted into IR, and instead puts in a `BYPASS` instruction, causing an error to be returned. This ensures that other TAP controllers cannot be accidentally put into strange states by a bug in the client of this TAP controller.
- As a consequence of IR length checking, the data to be shifted into the IR must be in a single `TAPOp_AnySequence_RW` call. It must not be split over two calls. This is necessary so that the call can ensure that the correct number of IR shifts are performed before processing any of them.
- The number of shifts performed in `Shift-DR` must be the same as the length of the scan chain. If it is not, then in a multi-processor system the operation is undefined due to the insertion of extra **TCK**s by the server to bypass other TAP controllers.
- If used in a macro, all 256 bits must be passed. Use fixed or variable `unsigned32` TAPOp macro parameters. The value of `numTCKs` determine how many of the supplied bits are actually used. For a single `TAPOp_AnySequence_RW` call, a pointer to an array is passed.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_AnySequence_W* on page 2-68.

2.5.15 TAPOp_AnySequence_W

Performs a sequence of **TCK**s, with explicitly specified **TMS** and **TDI** inputs. **TDO** is ignored.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_AnySequence_W(unsigned8 connectId, unsigned8 numTCKs,
                  unsigned32 *TDIbits, unsigned32 *TMSbits, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

numTCKs The number of **TCK**s to perform (maximum255).

TDIbits **TDI** data is read from here. This must not be NULL.

TMSbits **TMS** data is read from here. This must not be NULL.

deselect 0 indicates that the connection to this TAP controller remains selected (excluding access to other TAP controller connections). Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_InBadTAPState`

The TAP controller was reset or is not in `Select-DR-Scan` or `Run-Test/Idle`.

TAPOp_BadParameter

Failed for one of the following reasons:

- TDIbits = NULL
- TMSbits = NULL
- numTCKs = 0.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

TAPOp_AnySeqUsedBadPath

An Exit2-IR/DR to Shift-IR/DR transition was requested.

TAPOp_AnySeqWrongIRLength

The wrong number of Shift-IR **TCK**s was detected.

Usage

This function provides an interface to the virtual TAP controller in the Multi-ICE server. If there are actually several TAP controllers connected in series, the server deals with having to insert extra bits into the IR and DR scan chains transparently to the caller of this function. For this to work, however, some restrictions are imposed on what sequences can be performed. These restrictions are described in *Restrictions* on page 2-71.

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

Note

When the connection is deselected, the TAP controller must be left in one of the states Select-DR-Scan or Run-Test/Idle. If this is not done:

- this call to TAPOp_AnySequence_W returns the error TAPOp_InBadTAPState
- the TAP controllers are reset and left in the state Run-Test/Idle
- other connections on the same scan chain are forced to acknowledge the reset as if the call to TAPOp_AnySequence_W had passed through Test-Logic Reset (see below).

If another connection on the same Multi-ICE server resets the TAP controllers by calling TAPOp_AnySequence_W, TAPOp_AnySequence_RW, or TAPOp_TestResetSignal, all subsequent calls to TAPOp_AnySequence_W are rejected with the error TAPOp_InBadTAPState until the reset is acknowledged. As soon as the TAP controllers move from Test-Logic Reset to Run-Test/Idle the controllers are reset, and then:

1. The Multi-ICE server places every TAP controller in the bypass mode selected by the last call to *TAPOp_SetAutoBypassInstruction* on page 2-118, or in the default bypass mode if no such call has been made.
2. The SCSR for the connection that caused the reset is restored to select the chain that was active before the reset took place.
3. The IR and SCSR for all the other connections are restored to their previous values as each connection is selected.
4. The remainder of the control bits from a TAPOp_AnySequence call, or the next TAPOp call, are carried out.

After a TAP controller reset, API calls to other connections will fail if the function called is one of the following (either called directly or embedded in a macro):

- ARMTAP_AccessDR_1Clk_W
- ARMTAP_AccessDR_NoClk_W
- ARMTAP_AccessDR_RW
- ARMTAP_AccessDR_RW_And_Test
- ARMTAP_AccessDR_W
- ARMTAP_AccessIR
- ARMTAP_AccessIR_1Clk
- ARMTAP_AccessIR_nClks
- ARMTAP_ClockARM
- TAPOp_AccessDR_RW
- TAPOp_AccessDR_W
- TAPOp_AccessIR
- TAPOp_AnySequence_RW
- TAPOp_AnySequence_W

On detecting the error TAPOp_InBadTAPState, each connection must perform the following actions:

1. Call TAPOp_ReadPrivateFlags. If the TAPOp_TestLogicReset flag is not set, the bad state error was caused by another problem. If it is set, the TAP controller was reset.

2. Set TAPOp_TestLogicResetACK in the read flags and call TAPOp_WritePrivateFlags to acknowledge the reset.

3. Restore any required state in the TAP controller logic. For example, on an ARM9 device, you must restore any breakpoints that were set.

4. Retry the failed TAPOp call.

Example

This example demonstrates the use of TAPOp_AnySequence_W in changing the TAP state from Select-DR-Scan to Run-Test/Idle.

```
{
    ... declarations
    TMSbits = 0x6;   /* 0110b: Capture-DR, Exit-DR, Update-DR, Run-Test/Idle */
    TDIbits = 0;
    TAPCheck(TAPOp_AnySequence_W(connectId, 4, &TDIbits, &TMSbits, 1));
```

Restrictions

The following restrictions apply:

- When shifting data into an IR or DR register, all the shifts (performed when the TAP controller start state is Shift-IR/DR) must occur together. After performing these shifts, Shift-IR/DR must not be left and then re-entered without going through Capture-IR/DR. If you do this, the server returns an error indicating that the transition Exit2-IR/DR to Shift-IR/DR was requested. You must code the client code to avoid this case.

- The Multi-ICE server knows the length of the IR register. If the number of shifts into this register is not the same as the length according to the server, the server ignores the data shifted into IR, and instead puts in a BYPASS instruction, causing an error to be returned. This ensures that other TAP controllers cannot be accidentally put into strange states by a bug in this TAP controller client.

- As a consequence of IR length checking, the data to be shifted into the IR must be in a single TAPOp_AnySequence_W call. It must not be split over two calls. This is necessary so that the call can ensure that the correct number of IR shifts are performed before processing any of them.

- The number of shifts performed in Shift-DR must be the same as the length of the scan chain. If it is not, then in a multi-processor system the operation is undefined due to the insertion of extra **TCK**s by the server to bypass other TAP controllers.

- If used in a macro, all 256 bits must be passed. Use fixed or variable `unsigned32`s. The value of `numTCKs` determine how many are actually used. For a single `TAPOp_AnySequence_W` call, a pointer to an array is passed.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_AnySequence_RW* on page 2-65.

2.5.16 TAPOp_CloseConnection

Close a TAPOp connection.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_CloseConnection(unsigned8 connectId)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

Returns

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing thisrequest.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The TAPOp connection is closed and the resources associated with it are released for use by another client. After this call, TAPOp functions must not use this connection ID. `TAPOp_DeleteAllMacros` is called to remove the macro definitions defined for this connection.

Note

Do not confuse closure of a connection with deselecting a connection, which enables the client to tell the server at what points the hardware can be shared with another client. Deselection occurs when a TAPOp function is called and the deselect parameter is nonzero.

Example

This example demonstrates closing a connection and then closing the base RPC interface in a program that only opens one connection. The `TAPCheck` macro is not used in this case because the error handling associated with it is inappropriate. Instead, an explicit loop is used, with a simple timeout counter.

```
{
    ... declarations
    int timeout = 20;
    // close TAP connection
    do {
        t_err = TAPOp_CloseConnection(connectId);
    } while (t_err == TAPOp_UnableToSelect && timeout-- > 0);
    // close TCP/IP connection
    TAPOp_RPC_Finalise();
```

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_DeleteAllMacros* on page 2-79
- *TAPOp_GetDriverDetails* on page 2-88
- *TAPOp_OpenConnection* on page 2-94
- *TAPOp_RPC_Initialise* on page 2-106
- *TAPOp_RPC_Finalise* on page 2-105.

2.5.17 TAPOp_DefineMacro

Adds a single TAPOp or ARMTAP function call to a macro.

Syntax

```
#include "tapmacro.h"

extern TAPOp_Error TAPOp_DefineMacro((unsigned8 connectId, unsigned8 macroNo,
                  char *callDetails, unsigned8 nTimes, void *fixedParamValues,
                  int paramSize)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

macroNo The number of the macro. This is a client selectedinteger that identifies the macro to append to. Macros are not shared between connections.

callDetails This is a string that is interpreted according to the following rules:

- *fn-name* Is used if there are no fixed parameters for this invocation.
- *fn-name*:*list* Is used if there are fixed parameters.

fn-name must be the name of a TAPOp or ARMTAP function. For example, `ARMTAP_AccessDR_W`.

The *list* defines the parameters of the call that are fixed, and their values must be specified in `fixedParamValues`. Parameters that are not listed in `callDetails` are variable parameters. These are specified as arguments to the function `TAPOp_RunMacro`.

A `callDetails` specification consists of a sequence of digits and uppercase letters in increasing order, digits first. Each digit or letters refers to a structure element in the corresponding `MAC_xxx` structure defined in `macstruct.h`, starting at 1.

nTimes The number of times the line of this macro is executed. Each time it is executed, the same fixed parameters are used, but a new set of variable parameters is used, and results are added to the result block.

fixedParamValues

A block of data holding the fixed parameters for this invocation of this macro.

paramSize The size of the block of fixed parameter data. This is required so the data can be sent easily over RPC.

Returns

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_InBadTAPState`

TAP controller is not in `Run-Test/Idle`.

`TAPOp_BadParameter`

Unrecognized *`fn-name`* in `callDetails`.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

`TAPOp_TooManyMacros`

Maximum number of macros exceeded. The maximum is 250.

`TAPOp_TooManyMacroLines`

Too many lines in the macro. The maximum is 1000.

`TAPOp_BadFixedParamNo`

A bad parameter number was specified.

`TAPOp_UnknownProcedureName`

Unknown procedure name in `TAPOp_DefineMacro` call.

`TAPOp_CantUseProcInMacro`

Procedure specified in `TAPOp_DefineMacro` call cannot be run in a macro.

Usage

`TAPOp_DefineMacro` adds a line (one TAPOp or ARMTAP function call) to a TAPOp macro definition, creating the macro as required. Macro definitions are stored in the Multi-ICE server and are private to each connection. Using macros instead of direct calls enables you to speed up the execution of clients, because the overhead of communicating with the server is reduced significantly.

This function must be called for each line of a macro. If a macro with the same ID number is already defined for this connection, the new line is added to the existing macro. If there is no current definition, a new macro is created with this line. You can display a summary of the contents of a macro by calling TAPOp_DisplayMacro.

You can specify that a line added by TAPOp_DefineMacro is executed a number of times, and a parameter to TAPOp_RunMacro enables you to specify that the whole macro is run a number of times. This allows the server to optimize some operations further, providing better performance.

If you run a macro or a macro line more than once by using the nTimes argument of TAPOp_DefineMacro or TAPOp_RunMacro, the same set of fixed parameter data is used each time the line is executed. However, each time a macro line is executed, a new set of variable data (passed in at runtime) is used and new results are output.

Example

This example demonstrates defining a macro to select a scan chain on an ARM processor.

There are three TAPOp calls in this macro. The fixed parameters are defined using the InitParams and NREnterParamUx C macros, described in *TAPOp constant and macro definitions* on page 2-11. There is one variable parameter, TDIbits1 of the second API call, that enables the scan chain to be specified when the macro is run.

See also the examples in *Using TAPOp macros* on page 1-16.

```
#define SELCHAIN   1   /* macro number for select scanchain macro */
{
    ... declarations
    InitParams;
    NREnterParamU16(SCAN_N);    //TDIbits
    NREnterParamU8(0);          //TDIrev
    NREnterParamU8(0);          //nClks
    TAPCheck(TAPOp_DefineMacro(cId, SELCHAIN, "ARMTAP_AccessIR_nClks:123", 1,
                                        Values, ValPtr));
    InitParams;
    NREnterParamU8(0);          //TDIbits2
    NREnterParamU8(0);          //TDIrev
    NREnterParamU8(TDefs->SCSRlen); //len
    NREnterParamU8(0);          //WRoffset
    NREnterParamU32(TDefs->SCSRMask.low32); //WRmask1
    NREnterParamU8(TDefs->SCSRMask.high8);  //WRmask2
    NREnterParamU8(0);          //nClks
    TAPCheck(TAPOp_DefineMacro(cId, SELCHAIN, "ARMTAP_AccessDR_W:2345678", 1,
                                        Values, ValPtr));
    InitParams;
    NREnterParamU16(INTEST);    //TDIbits
```

```
NREnterParamU8(0);          //TDIrev
NREnterParamU8(0);          //nClks
TAPCheck(TAPOp_DefineMacro(cId, SELCHAIN, "ARMTAP_AccessIR_nClks:123", 1,
                                        Values, ValPtr));
```

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_DeleteAllMacros* on page 2-79
- *TAPOp_DeleteMacro* on page 2-80
- *TAPOp_DisplayMacro* on page 2-82
- *TAPOp_RunBufferedMacro* on page 2-109
- *TAPOp_RunMacro* on page 2-113.

2.5.18 TAPOp_DeleteAllMacros

Delete all macros for a particular connection.

Syntax

```
#include “tapmacro.h”

extern TAPOp_Error TAPOp_DeleteAllMacros(unsigned8 connectId)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing thisrequest.

Usage

The call deletes all macros for a particular connection. You do not normally have to do this, as the macros for a connection are automatically deleted when the connection is closed.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_CloseConnection* on page 2-73
- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_DeleteMacro* on page 2-80
- *TAPOp_DisplayMacro* on page 2-82
- *TAPOp_RunBufferedMacro* on page 2-109
- *TAPOp_RunMacro* on page 2-113.

2.5.19 TAPOp_DeleteMacro

Delete a macro for a particular connection.

Syntax

```
#include “tapmacro.h”

extern TAPOp_Error TAPOp_DeleteMacro(unsigned8 connectId, unsigned8 macroNo)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

macroNo The number of the macro to delete.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing thisrequest.

`TAPOp_UndefinedMacro`

The `macroNo` is not defined for that `connectId`.

Usage

This call deletes any current definition of the TAPOp macro with the supplied number.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_CloseConnection* on page 2-73
- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_DeleteMacro*
- *TAPOp_DisplayMacro* on page 2-82
- *TAPOp_RunBufferedMacro* on page 2-109

- *TAPOp_RunMacro* on page 2-113.

2.5.20 TAPOp_DisplayMacro

Display a summary of the lines of a macro in the server log window.

Syntax

```
#include “tapmacro.h”

extern TAPOp_Error TAPOp_DisplayMacro(unsigned8 connectId, unsigned8 macroNo)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

macroNo The number of the macro to display.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing thisrequest.

Usage

If a macro is defined, a summary of its contents is displayed in the format described in *Output format*. If the macro has not been defined, this is displayed instead. No error is returned.

Output format

The macro information is displayed on the Multi-ICE server in the following format:

line_no : *TAPOp/ARMTAP_call*, *fixed_params*, *macro_line_executions*

For example:

```
------------------------- Macro no. 1 ----------------------
0: ARMTAP_AccessDR_W , plist=0x1F , nTimes=1
1: ARMTAP_AccessDR_W , plist=0x1E , nTimes=14
```


In line 0:

```
plist=0x1F = b00011111
```

shows that parameters 1, 2, 3, 4, and 5 have been fixed.

In line 1:

```
plist=0x1E = b00011110
```

shows parameters 2, 3, 4, and 5 have been fixed.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_RunBufferedMacro* on page 2-109
- *TAPOp_RunMacro* on page 2-113
- *TAPOp_DeleteMacro* on page 2-80.

2.5.21 TAPOp_FillMacroBuffer

Loads a buffer in the server with the variable parameters used by a subsequent call to *TAPOp_RunBufferedMacro* on page 2-109.

Syntax

```
#include "tapmacro.h"

extern TAPOp_Error TAPOp_FillMacroBuffer(unsigned8 connectId,
                  unsigned8 bufferNo, void *variableParamValues, int paramSize)
```

where:

connectId Connection ID, as returned by TAPOp_OpenConnection.

bufferNo The number of the buffer to load with the variabledata block. This must be 0 or 1.

variableParamValues

A block of data holding the variable parameters to be written to the buffer.

paramSize The size of the block of variable parameter data. This is required for data to be sent over RPC.

Return

The function returns:

TAPOp_NoError

No error.

TAPOp_NoSuchConnection

The connectId was not recognized.

TAPOp_BadParameter

The bufferNo is not 0 or 1.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

Usage

The parameter data is passed to the server and stored in the indicated macro buffer. Existing data is overwritten, truncating the buffer as required. The data in the buffer is not referenced again until a call to TAPOp_RunBufferedMacro is made.

Note

The call does not have to select the connection and so never returns the error `TAPOp_UnableToSelect`.

Example

This example demonstrates filling server macro buffer number 1 with the contents of an array, `data`. Once this has happened a macro that writes this data to the device can be run a number of times from this buffer.

```
... declarations
InitParams;
for (int i = 0; i < 32; i++)
    NREnterParamU32(data[i]);
}
TAPCheck(TAPOp_FillMacroBuffer(cId, 1, Values, ValPtr));
```

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_DeleteMacro* on page 2-80
- *TAPOp_RunBufferedMacro* on page 2-109
- *TAPOp_RunMacro* on page 2-113.

2.5.22 TAPOp_FlushScanQueue

Flushes the scan queue of JTAG primitives waiting to be issued by the server.

Syntax

```
#include “tapop.h”

extern TAPOp_ErrorTAPOp_FlushScanQueue(unsigned8 connectId)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

Usage

This function flushes the queue of JTAG primitives waiting to be issued by the server to the Multi-ICE hardware. It ensures that all the JTAG writes have been issued to the hardware.

The scan queue is flushed by the server when a request to read data from the device is made (for example, using `TAPOp_AccessDR_RW`) and after a timeout has expired between the data being placed in the buffer and the next TAPOp call that affects the connection.

It is not usually necessary to explicitly flush the scan queue. An example of requiring a flush is to ensure that a device write that initiates some hardware action is made promptly.

Compatibility

This function first appeared in Multi-ICE Version 1.4.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_AccessDR_RW* on page 2-56
- *TAPOp_AccessDR_W* on page 2-59.

2.5.23 TAPOp_GetDriverDetails

Get from the server an array of the devices that are connected to the server.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_GetDriverDetails(MultiICE_DriverDetails *driverDets,
                  unsigned32 *numDrivers, unsigned32 *versionNo)
```

where:

driverDets Pointer to the structure containing the device details.See the structure definition in *MultiICE_DriverDetails* on page 2-8.

numDrivers The number of drivers in the structure pointed to by DriverDets. Before you call this procedure, set numDrivers to the number of MultiICE_DriverDets structures in the array driverDets. The procedure returns the actual number in numDrivers.

versionNo The version number of the Multi-ICE server.

Return

The function returns:

TAPOp_NoError

No error.

TAPOp_NotInitialised

The Multi-ICE server is not configured.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

Usage

Before a debugger opens a connection to a device on the Multi-ICE server, it must find out from the server the devices that are available. You use TAPOp_GetDriverDetails to do this, and use the information returned to select a driver to connect to.

Note

TAPOp_GetDriverDetails uses the user-supplied function GetServerName to return the name of the computer to query. You do not require an open connection.

You must call TAPOp_RPC_Initialise before calling TAPOp_GetDriverDetails.

Example

The example demonstrates getting driver and device details from the Multi-ICE server and scanning them to find the device at a specific TAP position. In overview:

1. TAPOp_RPC_Initialise() is called to set up an RPC connection to the server.
2. TAPOp_RPC_Initialise() calls GetServerName() (not shown here) to get the hostname of the Multi-ICE server.
3. TAPOp_GetDriverDetails() is called to query the server.
4. The returned array is scanned to find the required device.
5. The information in driverDets is then be used to make a call to TAPOp_OpenConnection() (not shown here).

```
#define arraySize(arr) (sizeof(arr)/sizeof(arr[0]))
... definition of GetServerName().
{
    unsigned32              i, versionNo;
    MultiICE_DriverDetails  driverDetails[10];
    int                     driverNo;
    unsigned32              driverCount;
    /* Try to connect to the network service */
    if (TAPOp_RPC_Initialise() != 0) {
        fprintf(stderr, "ERROR: Can't connect to rpc service\n");
        exit (3);
    }
    /* Ask the server what drivers (processors) it has */
    driverCount = arraySize(driverDetails);
    err = TAPOp_GetDriverDetails(driverDetails, &driverCount, &versionNo);
    if (err != TAPOp_NoError) {
        fprintf(stderr,"ERROR: GetDriverDetails failed, error %d\n", err);
        TAPOp_RPC_Finalise();
        exit(3);
    }
    /* Search the list for the one matching TAP_pos */
    for (driverNo = -1, i=0; i < driverCount; i++) {
        if (driverDetails[i].TAP_pos == TAP_pos) {
            driverNo = i;
            break;
        }
    }
    if (driverNo == -1) {
        fprintf(stderr,"ERROR: TAP %d doesn't exist on server %s\n",
                        TAP_pos, hostname);
```

```
    TAPOp_RPC_Finalise();
    exit (3);
}
... use members of driverDetails[driverNo] to call TAPOp_OpenConnection
```

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_OpenConnection* on page 2-94
- *TAPOp_RPC_Initialise* on page 2-106
- *TAPOp_GetModelDetails* on page 2-91.

2.5.24 TAPOp_GetModelDetails

Reads details of the Multi-ICE hardware, returning them in an information block.

Syntax

```
#include “tapop.h”

extern TAPOp_ErrorTAPOp_GetModelDetails (unsigned8 connectId,
                  unsigned32 *dataBlk, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

dataBlk 4 words of model details information:

word 0, byte 0 = model number

word 0, byte 1 = model level

word 1, byte 0 = number of TAPs configured.

All other bytes reserved.

deselect If 0 then the connection to this TAP controller remains selected, so excluding access by other TAP controller connections. Otherwise the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

This function reads details of the Multi-ICE hardware, returning the details in the `dataBlk` information block.

Example

The example demonstrates getting Multi-ICE interface unit model number:

```
unsigned32    datablk[4];    unsigned8     model_number;
/* Read the Multi-ICE model info from the server */
TAP_Check(TAPOp_GetModelDetails(connectId, datablk, 1 /*deselect*/));
model_number = (unsigned8) (datablk[0] & 0xFF);
```

Compatibility

This function first appeared in Multi-ICE Release 1.4.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_GetDriverDetails* on page 2-88
- *TAPOp_OpenConnection* on page 2-94
- *TAPOp_RPC_Initialise* on page 2-106.

2.5.25 TAPOp_LogString

This function inserts a user-supplied string into the log file.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_LogString(unsigned8 connectId, char *message)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

message The string to output to the log file. The stringlength is limited to 255 characters.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

This call does not require a selected connection to a TAP controller, because it is just providing debug output. Display of the message is conditional on logging being enabled, either using the Multi-ICE server menu commands, or by calling `TAPOp_SetLogging`.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_DisplayMacro* on page 2-82
- *TAPOp_SetLogging* on page 2-124.

2.5.26 TAPOp_OpenConnection

Open a connection through a Multi-ICE server to one or more scan chains on a TAP controller.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_OpenConnection(unsigned8 TAPPos, unsigned8 *connectId,
                   unsigned8 numScanChains, unsigned8 *scanChains,
                   unsigned8 startState, unsigned8 IRlen, unsigned32 intest,
                   unsigned8 SCSRlen, unsigned32 scan_n,
                   unsigned8 allowAutoDisconnect, char *appName,
                   char *driverName)
```

where:

TAPPos The position in the scan chain of the TAP controllerto connect to. Position 0 is closest to **TDI**.

connectId The connection ID to be used when calling other TAPOp functions.

numScanChains

The number of scan chains claimed by this connection.

scanChains An array containing the numbers of the scan chains claimed by this connection.

startState The startup state of TAP controller for this connection:

- 1 starts the TAP controller(s) in `Select-DR-Scan`
- 0 starts the TAP controller(s) in `Run-Test/Idle`
- other values are invalid.

IRlen This is the length of the IR of the TAP being connected. This is compared with the length stored in `IRlength.arm` for the device.

intest `INTEST` instruction bit pattern for the TAP that this connection is associated with.

SCSRlen Length of the scan chain select register for the TAP being connected. If it is 0, the TAP does not support multiple scan chains.

scan_n `SCAN_N` instruction bit pattern for the TAP that this connection is associated with. It is only valid if `SCSRlen 0`.

allowAutoDisconnect

This flag is set if the server is allowed to disconnect this client due to another client attempting to connect. If this client implements the standard heartbeat mechanism (automatic for Win32 clients), this flag must be set. If this is a non-Win32 client and there is no special heartbeat set up, this flag must be 0.

appName A null-terminated string that gives the identity of the debugger.

driverName A null-terminated string that contains the driver name from `TAPOp_GetDriverDetails` for this connection.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_TAPNotPresent`

Bad TAP controller position.

`TAPOp_NotInitialised`

The Multi-ICE server is not configured.

`TAPOp_TooManyConnections`

There are no free connection IDs.

`TAPOp_BadParameter`

Bad parameter value passed.

`TAPOp_ScanChainAlreadyClaimed`

One of the required scan chains has already been claimed by another connection.

`TAPOp_ParameterConflicts`

A parameter conflicts with the configuration data in the server configuration file.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

At the beginning of a debug session, this function creates a connection to one or more scan chains on a TAP controller, and returns a connection ID to be used when calling other TAPOp functions. The parameters passed to the function enable the Multi-ICE server to manipulate the TAP controller itself, so that it can virtualize TAP connections.

The parameter, `allowAutoDisconnect`, is passed to `TAPOp_OpenConnection`. If it is defined as `1`, the server disconnects the client if the server does not receive a TAPOp request from the client within a predefined timeout. On Microsoft Windows systems, a separate thread is used that calls `TAPOp_PingServer` at regular intervals to keep the connection alive. This is not done for the UNIX clients, and so `allowAutoDisconnect` is set to `0`.

The details of the device passed to `TAPOp_OpenConnection` (for example, the length of the *Scan Chain Select Register* (SCSR), or the encoding of `INTEST`) must be known or derived by the client. The Multi-ICE server does not know anything about individual devices. You can however use the `DriverName` as a key to look this information up, because the `DriverName` is fixed for any particular device.

Note

Do not confuse this function with selecting a connection that occurs when a TAPOp function is called and a different connection was previously selected.

Example

The example demonstrates opening a connection to the Multi-ICE server. It follows on from the example presented for *TAPOp_GetDriverDetails* on page 2-88. In overview:

1. `TAPOp_RPC_Initialise()` (not shown here) is called to set up an RPC connection.
2. `TAPOp_RPC_Initialise()` calls `GetServerName()` (not shown here) to get the hostname of the Multi-ICE server.
3. `TAPOp_GetDriverDetails()` (not shown here) is called to query the server and returned array is scanned to find the required device.
4. An array is set up containing the scan chains that this connection is using. In calling `TAPOp_OpenConnection`, you are only connecting to the listed scan chains.
5. The information in `driverDets` is used to make a call to `TAPOp_OpenConnection()`.

```
{
    MultiICE_DriverDetails  driverDetails[10];
    int                     driverNo;
    unsigned8               TAP_ScanChainsClaimed[4];
    unsigned8               TAP_TotalClaimedChains;
```

```
    unsigned8               allowAutoDisconnect;
    unsigned32              driverCount;
    ... call TAPOp_RPC_Initialise
    ... call TAPOp_GetDriverDetails to get the available drivers
    ... search driverDets[] for the correct driver and set driverNo
    /* The scan chains we will try to claim - this assumes it is an ARM */
    TAP_ScanChainsClaimed[0] = 0;
    TAP_ScanChainsClaimed[1] = 1;
    TAP_ScanChainsClaimed[2] = 2;
    TAP_TotalClaimedChains = 3;
#ifdef WIN32
    allowAutoDisconnect = 1;
#else
    allowAutoDisconnect = 0;
#endif
    /* Try to connect to the device on the Multi-ICE server */
    err=TAPOp_OpenConnection(TAP_pos,  /* The TAP number */
              &connectId,              /* The connection ID is returned
                                        * here if a connection is made */
              TAP_TotalClaimedChains,  /* Number of scan chains to claim */
              TAP_ScanChainsClaimed,   /* List of scan chains claimed */
              1,                       /* Start in Select-DR-scan state */
              4,                       /* instruction register is 4 bits wide */
              INTEST,                  /* INTEST instruction encoding */
              4,                       /* SCS register is 4 bits wide */
              SCAN_N,                  /* SCAN_N instruction encoding */
              allowAutoDisconnect,     /* Allow server to disconnect us if
                                        * the connection appears to be dead */
              appName,                 /* A name identifying the connection */
              driverDetails[driverNo].DriverName);
    /* If the device is already connected we get this error back */
    if (err == TAPOp_ScanChainAlreadyClaimed) {
        ... handle error, for example by prompting the user to change TAP pos.
    }
```

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_CloseConnection* on page 2-73
- *TAPOp_GetDriverDetails* on page 2-88
- *TAPOp_RPC_Initialise* on page 2-106
- *TAPOp_RPC_Finalise* on page 2-105.

2.5.27 TAPOp_PingServer

This provides a heartbeat function.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_PingServer(unsigned8 connectId)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing thisrequest.

Usage

The function is used to poll the server so that the server knows the client is still connected during periods of inactivity. Although this function is part of the TAPOp public interface, it is not used in the user-written parts of Windows clients, because the code provided in `rpcclient.c` sets up a thread that calls this function once per second when a connection is active.

UNIX clients must call `TAPOp_PingServer` periodically if the autodisconnect behavior described in *TAPOp_OpenConnection* on page 2-94 is required.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_OpenConnection* on page 2-94
- *TAPOp_RPC_SetTimeout* on page 2-108.

2.5.28 TAPOp_ReadCommonData

Reads the data block that is common to all the clients connected to the server.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_ReadCommonData(unsigned8 connectId,
                   unsigned32 *commonBlk, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

commonBlk Four words of common data whose meaning is determinedby the client. This buffer is allocated by the caller.

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_BadParameter`

Failed because `commonBlk` is NULL.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The call returns in the memory referenced by `commonBlk` the four words of data allocated to TAPOp connections by the server. This data is shared between all connections, and so must be written with care.

To safely write to the common data block, use an atomic Read-Modify-Write sequence. You can make a call of `TAPOp_ReadCommonData` and `TAPOp_WriteCommonData` atomic by setting the deselect parameter of the `TAPOp_ReadCommonData` call to zero.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_ReadPrivateFlags* on page 2-103
- *TAPOp_WriteCommonData* on page 2-133
- *TAPOp_WritePrivateFlags* on page 2-139.

2.5.29 TAPOp_ReadMICEFlags

Read the Multi-ICE server flags word.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_ReadMICEFlags(unsigned8 connectId, unsigned8 *flags)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

flags A pointer to a variable into which the flags arewritten. The bits set are defined in *TAPOp inter-client communication flags* on page 2-22.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_BadParameter`

Failed because `flags == NULL`.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

Note

Do not use the standard `TAPCheck` macro to retry this call, because that macro also calls `TAPOp_ReadMICEFlags` to determine if the wrapped call failed because the target was reset or switched off.

The Multi-ICE server flags provide information relating to the state of the target power, target reset, the state of the user input bits and the state of the user output bits. There is a separate set of flags for each connection. The flags are not written directly by the client application, but maintained by the server. For example, the flag

TAPOp_FL_TargetPowerHasBeenOff is set by the server as soon as the server detects that the target power has been interrupted. You must call TAPOp_CloseConnection and then TAPOp_OpenConnection to recover from this state.

Example

The example shows the use of TAPOp_ReadMICEFlags to check if the Multi-ICE interface unit is switched on at the moment, or not.

```
{
    ... declarations
    unsigned8 miceFlags;
    TAPOp_ReadMICEFlags(connectId, &miceFlags);
    if (miceFlags & TAPOp_FL_TargetPowerOffNow) {
        printf("Multi-ICE power is off; please switch it on.\n");
    }
```

Note

Because the Multi-ICE interface unit can be independently powered, knowing that the power is on does not imply that the unit is also plugged into the target.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_ReadPrivateFlags* on page 2-103
- *TAPOp_WriteCommonData* on page 2-133
- *TAPOp_WritePrivateFlags* on page 2-139.

2.5.30 TAPOp_ReadPrivateFlags

Reads the private word of flags for this connection.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_ReadPrivateFlags(unsigned8 connectId,
                   unsigned32 *flags)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

flags Private word of flags. The flags are described in *TAPOp inter-client communication flags* on page 2-22.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_BadParameter`

Failed because `flags = NULL`.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing thisrequest.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The flags are private to each connection and are used by the Multi-ICE server to signal events to each client. For example, one of the flags indicates that the device has been reset. The description for each flag in *TAPOp inter-client communication flags* on page 2-22 indicates how each is used, and when it is appropriate to use `TAPOp_WritePrivateFlags` to change a flag.

Example

The example shows the use of `TAPOp_ReadPrivateFlags` to check whether the Multi-ICE server forced the processor to stop, for example as the result of being in a synchronous stop group.

```
{
    ... declarations
    unsigned32 privateFlags;
    TAPCheck(TAPOp_ReadPrivateFlags(connectId, &privateFlags));
    if (privateFlags & TAPOp_ProcStoppedByServer) {
        printf("TestForDebugState: Processor stopped by Server\n");
        halt_forced = 1;
    }
```

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_ReadCommonData* on page 2-99
- *TAPOp_WriteCommonData* on page 2-133
- *TAPOp_WritePrivateFlags* on page 2-139.

2.5.31 TAPOp_RPC_Finalise

Call this to close the connection to the Multi-ICE server.

Syntax

```
#include “tapop.h”

extern void TAPOp_RPC_Finalise(void)
```

Return

None.

Usage

Use this function to close the RPC connection to the Multi-ICEserver for this application and free up the resources allocated in `TAPOp_RPC_Initialise`. You must close each of the TAPOp client connections by calling `TAPOp_CloseConnection` for each open `connectId` before calling `TAPOp_RPC_Finalise`.

Compatibility

This function first appeared in Multi-ICE Version 2.1, replacing but providing the same functionality as `rpc_finalise`.

The Multi-ICE Version 2.1 TAPOp library implements `rpc_finalise` for backward compatibility. It is now deprecated and might be removed in a future release.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_CloseConnection* on page 2-73.
- *TAPOp_RPC_Initialise* on page 2-106

2.5.32 TAPOp_RPC_Initialise

Initialize the Sun RPC subsystem and the TAPOp network layer in preparation for contacting a Multi-ICE server.

Syntax

```
#include “tapop.h”

extern int TAPOp_RPC_Initialise(void)
```

Return

The function returns:

0 No error.

Nonzero Failed to open RPC connection.

Usage

Opens an RPC connection to the Multi-ICE server. The Sun RPC subsystem is initialized and a connection attempt made to the Multi-ICE server, using the host name returned from a call of `GetServerName`. If this is successful, the server is queried to discover it’s capabilities.

If the host name is `localhost`, the server version is used to work out whether a non-RPC (shared memory) communications link to the server can be used. If available, this speeds up the communication between client and server, but can only be used when the server and client are on the same workstation.

Note

You must call `TAPOp_RPC_Initialise` before making any other TAPOp calls.

Example

The example shows the use of TAPOp_ReadPrivateFlags to check whether the Multi-ICE server forced the processor to stop, for example as the result of being in a synchronous stop group.

```
{
    ... declarations
    unsigned32 privateFlags;
    TAPCheck(TAPOp_ReadPrivateFlags(connectId, &privateFlags));
    if (privateFlags & TAPOp_ProcStoppedByServer) {
```



```
        printf("TestForDebugState: Processor stopped by Server\n");
        halt_forced = 1;
    }
```

Compatibility

This function first appeared in Multi-ICE Version 2.1.

The function `rpc_initialise` is implemented in the Multi-ICE Version 2.1 TAPOp library for backward compatibility only. It is now deprecated. Do not use the name `rpc_initialise` in new code.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_OpenConnection* on page 2-94
- *TAPOp_CloseConnection* on page 2-73
- *TAPOp_GetDriverDetails* on page 2-88
- *TAPOp_RPC_Finalise* on page 2-105.

2.5.33 TAPOp_RPC_SetTimeout

Set the RPC timeout for all connections from this client.

Syntax

```
#include “tapop.h”

extern void TAPOp_RPC_SetTimeout(int seconds)
```

where:

seconds RPC timeout value to set.

Usage

This call sets the RPC communications timeout for all TAPOpconnections created by this client application. The initial timeout is the value of the constant `WAIT_FOR_NORMAL_OPERATIONS`, defined in `rpcclient.c`. You can increase this value if you are using a slow or unreliable connection to the server.

See also *Known problem with HP-UX clients* on page 1-13.

Return

None.

Compatibility

This function first appeared in Multi-ICE Version 2.1.

The now obsolete function `rpc_set_timeout` was implemented in Multi-ICE Version 1, and is implemented in the Multi-ICE Version 2.1 TAPOp library to maintain linker compatibility. It is now deprecated. Do not use the old function name in new code.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_RPC_Initialise* on page 2-106
- *TAPOp_RPC_Finalise* on page 2-105.

2.5.34 TAPOp_RunBufferedMacro

Run a predefined macro using variable parameter information from the Multi-ICE server macro buffer and then reading out any results.

Syntax

```
#include "tapmacro.h"

extern TAPOp_Error TAPOp_RunBufferedMacro(unsigned8 connectId,
                  unsigned8 macroNo, unsigned8 bufferNo, int *lineno_error,
                  int *loopno_error, void *resultValues, int *resultSize,
                  int nTimes, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

macroNo The number of the macro to run, local to this connection.

bufferNo The number of the buffer that contains the variable data block. This must be 0 or 1 and the data must have been loaded before this call.

lineno_error

The macro line number that caused the error. This is valid when `TAPOp_RunBufferedMacro` returns an error:

- `0` indicates that the call to `TAPOp_RunBufferedMacro` failed, for example, an undefined macro number was specified.
- `1,2,3...` indicate failure on statement 1, 2, or 3 of the macro.

loopno_error

Macro loop number that caused the error. This is only valid if `TAPOp_RunBufferedMacro` returns an error. For example, if a macro call has `nTimes = 4` and it fails on the third time the macro is run, `loopno_error` is three.

resultValues

A block of data into which the results of the functions called by this macro are placed. This is allocated by the caller.

resultSize On entry, this indicates the maximum amount of data that can fit in `resultValues`. On exit, the actual amount of data in `resultValues` is returned.

nTimes The number of times the macro is run (parameters are read and results are written cumulatively from or to the data arrays).

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

`TAPOp_CouldNotBuildCompleteParameterList`

Parameters were missing during either the macro define or execution.

Others Any other errors that the components of the macro can return.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

This function is very similar to `TAPOp_RunMacro` except that the variable data is not passed alongside the call. Instead, the variable data is written into one of the two available buffers in the Multi-ICE server using `TAPOp_FillMacroBuffer`. This is useful because:

- `TAPOp_FillMacroBuffer` cannot return the error `TAPOp_UnableToSelect`, so under normal circumstances it does not fail. This means that the variable data for a `TAPOp_RunMacro` call does not have to be resent if the server is busy. With large variable data blocks the time to transfer the variable data dominates the communication time, and so avoiding resending the data improves performance on multi-processor systems, where the server often returns `TAPOp_UnableToSelect`.

- There are cases, for example filling memory with a pattern, when the variable data in a macro is actually fixed for a few successive calls. Writing this data to a buffer and running the macro a number of times avoids sending it over the network and so improves performance.
- Because there are two buffers, a multi-threaded client can be loading one buffer while running a macro from the other. This overlap of the data transfer and execute portions of the `TAPOp_RunMacro` calls improves performance and ensures that the interface between client and server is fully utilized. This is particularly important on relatively slow interfaces such as 10Mbps Ethernet.

Like `TAPOp_RunMacro`, this call returns `TAPOp_UnableToSelect` when the server is busy.

Example

This example shows a function that writes a register bank back to the processor (remember that ARM processor register numbers are processor mode dependent). The function does this by:

1. Formatting the data using the parameter macros.
2. Sending the formatted data to the server macro buffer.
3. Running the macro identified by the constant `WRITE_ALL_REGS`. The macro must have been defined elsewhere in the client.

The instructions that cause the data to end up in processor registers are hidden inside the macro itself, and a routine that wrote to target memory could look very similar.

```
#define WRITE_ALL_REGS 20
TAPOp_Error defineMacros(void)
{
    ... define macro WRITE_ALL_REGS
}
TAPOp_Error WriteRegisters(unsigned32 reg[])
{
    unsigned8 resultvalues[1];
    int lnerr, looperr, resultsize;
    int i;
    // adjust PC to account for the macro's instructions
    reg[15] -= 1 * 4;
    // Format the data to write
    InitParams;
    for (i = 0; i < 16; i++)
    {
        NREnterParamU32(reg[i]);
    }
    // Send the data to the server
```

```
    TAPCheck(TAPOp_FillMacroBuffer(connectId, 0, Values, ValPtr);
    // Call the macro...
    resultsize = 0;
    TAPCheck(TAPOp_RunBufferedMacro(connectId, WRITE_ALL_REGS, 0, &lnerr,
                                    &looperr, &resultvalues, &resultsize, 1, 0));

    return TAPOp_NoError;
}
```

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_ReadPrivateFlags* on page 2-103
- *TAPOp_WriteCommonData* on page 2-133
- *TAPOp_WritePrivateFlags* on page 2-139.

2.5.35 TAPOp_RunMacro

Runs a previously defined macro, passing in the variable data and reading out any results.

Syntax

```
#include “tapmacro.h”

extern TAPOp_Error TAPOp_RunMacro(unsigned8 connectId, unsigned8 macroNo,
                    void *variableParamValues, int paramSize, int *lineno_error,
                    int *loopno_error, void *resultValues, int *resultSize,
                    int nTimes, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by TAPOp_OpenConnection.

macroNo The number of the macro to run, local to this connection.

variableParamValues

A block of data holding the variable parametersfor this invocation of this macro.

paramSize The size of the block of variable parameter data. This is required so the data can be easily sent over RPC.

lineno_error

Gives the macro line number that caused the error. This is only valid if TAPOp_RunMacro returned an error. The numbers start from 0.

loopno_error

Macro loop number that caused the error. This is only valid if TAPOp_RunMacro actually returned an error. For example, if a macro call has nTimes=3 and it fails on the third time the macro is run, loopno_error is 3.

resultValues

A block of data where the results of the functions called by this macro are put. This is allocated by the caller.

resultSize On entry, this indicates the maximum amount of data that fits in resultValues. On exit, the actual amount of data in resultValues is returned.

nTimes The number of times the macro is run (parameters are read and results are written cumulatively to, or read cumulatively from, the data arrays).

deselect If 0, the connection to this TAP controller remains selected. This excludes access to this TAP controller from other Multi-ICE server connections. Otherwise the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

`TAPOp_CouldNotBuildCompleteParameterList`

Parameters were missing during either the macro define or execution.

Others Any other errors the components of the macro can return.

Usage

`TAPOp_RunMacro` executes a macro that has been stored in the Multi-ICE server using a sequence of calls to `TAPOp_DefineMacro`. Using a macro offers a number of advantages over calling the TAPOp functions directly:

- You can speed up the operation of your client, because:
  - there is no communication delay between the client and the server for macro requests
  - you can include a significant part of the data required in the macro definition, reducing the amount of data that must be formatted by the client and then transmitted.
- You can use the repeat counts available in `TAPOp_RunMacro` and `TAPOp_DefineMacro` to execute macro lines or whole macros a number of times improving the performance of both client and server.

- The Multi-ICE server macro buffer can be used to store variable data for use with a TAPOp_RunBufferedMacro call. This enables the client to avoid having to resend the data if a TAPOp call fails with an unable to select error, but also provides the opportunity to overlap data transmission to the server and macro execution, and to reuse commonly sent data blocks.

The disadvantage of using TAPOp macros is that it is harder to debug applications using them. It is therefore suggested that applications are developed using normal TAPOp API calls until the precise operation of the target device is sufficiently understood to convert to using macros.

A macro definition is a sequence of standard API function names, together with the values of some or all of the parameters required, known as the *fixed data*. Every time the macro is run the values of any missing parameters must be filled in with the remaining data, known as the *variable data*. If insufficient variable data is supplied with a run macro call, the call fails. If too much variable data is supplied, the excess is ignored.

Fixed data is supplied with the TAPOp_DefineMacro call, and the same values are always used for each API call in the macro, even if a macro or macro line is repeated. However, each item of the variable data supplied with a run macro call is used only once. For example, suppose a macro is defined with a call to ARMTAP_AccessDR_W that has every parameter except the first 32 bits of TDIbits fixed. It therefore requires 32 bits of variable data when it is run. If this macro call is repeated 10 times by setting nTimes to 10 in the TAPOp_RunMacro call, the TAPOp_RunMacro call expects 10 ° 32 bits of variable data. This property can be used to write large quantities of data to the device with relatively short macros.

Data that is read from a device by an API call, for example from the TDOBits parameter of ARMTAP_AccessDR_RW, is always written to the result parameter of the TAPOp_RunMacro call. You must ensure that the parameter resultSize is set before every call to TAPOp_RunMacro to indicate the size of the resultSize array. When the TAPOp_RunMacro call returns, resultSize contains the number of bytes actually written to the array and again, if a macro or macro line is repeated, the results are appended to the array. This feature can be used to read large volumes of data from the device using a short macro.

Note

It is important to remember that the data returned from ARMTAP_AccessDR_RW is formatted as a stream of bytes, and thus must be unpacked. It is suggested that functions or macros similar to those in Example 2-1 on page 2-116 are used to do this:

Example 2-1 Decoding resultValues

```
#ifdef BIG_END_CLIENT
__inline ScanData40 ToScanData40(unsigned8 *block)
{
    // Big end == high order bytes first.
    ScanData40 res;
    res.high8 = *block[0];
    res.low32 = (block[1] << 24) |(block[2] << 16) |(block[3] << 8) |(block[4]);
    return res;
}
#else
__inline ScanData40 ToScanData40(unsigned8 *block)
{
    // Little end == low order bytes first.
    ScanData40 res;
    res.high8 = block[4];
    res.low32 = (block[3] << 24) |(block[2] << 16) |(block[1] << 8) |(block[0]);
    return res;
}
#endif
```

Example

This example shows a function that writes a register bank back to the processor. The function does this by formatting the data using the parameter macros and then running the macro identified by the constant `WRITE_ALL_REGS`, defined elsewhere in the client.

The instructions that cause the data to end up in processor registers are hidden inside the macro itself, and a routine that wrote to target memory might look very similar.

```
#define WRITE_ALL_REGS 20
TAPOp_Error defineMacros(void)
{
    ... define macro WRITE_ALL_REGS
}
TAPOp_Error WriteRegisters(unsigned32 reg[])
{
    unsigned8 resultvalues[1];   // this macro does not return any data
    int lnerr, looperr, resultsize;
    int i;
    // adjust PC to account for the macro's instructions
    reg[15] -= 1 * 4;
    InitParams;
    for (i = 0; i < 16; i++)
```



```
    {
        NREnterParamU32(reg[i]);
    }
    // Call the macro, passing the variable parameters...
    resultsize = 0;    // not expecting any data back
    TAPCheck(TAPOp_RunMacro(connectId, WRITE_ALL_REGS, Values, ValPtr,
                            &lnerr, &looperr, &resultvalues, &resultsize, 1, 0));

    return TAPOp_NoError;
}
```

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_DefineMacro* on page 2-75
- *TAPOp_RunBufferedMacro* on page 2-109
- *TAPOp_SetControlMacros* on page 2-120
- *TAPOp_DeleteMacro* on page 2-80.

Macros are also described in *Using TAPOp macros* on page 1-16.

2.5.36 TAPOp_SetAutoBypassInstruction

Defines the instruction that is loaded into the IR when the server switches to another connection.

Syntax

```
#include “tapop.h”

extern TAPOp_ErrorTAPOp_SetAutoBypassInstruction (unsigned8 connectId,
                  unsigned32 auto_bypass)
```

where

connectId Connection ID, as returned by TAPOp_OpenConnection.

auto_bypass Instruction pattern to use when the device is auto-bypassed by the server.

Return

TAPOp_NoError

No error.

Usage

In a multi-device system, when the Multi-ICE server accesses a specific device, it automatically loads an instruction into the IR registers of the other devices. The default instruction it uses is the BYPASS instruction defined in the IEEE 1149.1 standard:

- This scheme works for multiple debugger connections where the debugger uses INTEST to change the internal state of the device.
- When EXTEST is being used to apply test vectors to the pins of the device, switching connections to another device causes BYPASS to be loaded. This affects the state of the external device pins.

  To get round this, some devices have a CLAMP instruction that latches the state of the pins when the device is in EXTEST mode.

You can use this function to define the instruction that is loaded into the IR when the Multi-ICE server switches to another connection. For example, instead of the default BYPASS instruction, you might load the CLAMP instruction for the device.

The selection status of the connection is irrelevant (this call cannot return TAPOp_UnableToSelect), and the selection status is not modified by this call.

Compatibility

This function first appeared in Multi-ICE Version 1.4.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_OpenConnection* on page 2-94.

2.5.37 TAPOp_SetControlMacros

Define the macro numbers that the Multi-ICE server uses to recognize device events and perform synchronized starting or stopping of processors.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_SetControlMacros(unsigned8 connectId, unsigned8 flags,
                   unsigned8 eventMacroNo, unsigned8 preExecMacroNo,
                   unsigned8 executeMacroNo, unsigned8 postExecMacroNo,
                   unsigned8 stopMacroNo, ScanData40 *eventMask,
                   ScanData40 *eventXOR)
```

where:

connectId Connection ID, as returned by TAPOp_OpenConnection.

flags Various flags (defined in *TAPOp inter-client communication flags* on page 2-22).

eventMacroNo The number of the macro to use as eventMacro.

preExecMacroNo

The number of the macro to use as preExecMacro.

executeMacroNo

The number of the macro to use as executeMacro.

postExecMacroNo

The number of the macro to use as postExecMacro.

stopMacroNo The number of the macro to use as stopMacro.

eventMask The mask for detecting an event (see the descriptionof eventMacro in *Usage* on page 2-121).

eventXOR XOR for detecting an event (see the description of eventMacro in *Usage* on page 2-121).

Return

The function returns:

TAPOp_NoError

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

`TAPOp_UndefinedMacro`

A macro number specified has not been defined.

Usage

Before calling this function, the client must define macros that perform the following functions:

eventMacro

This macro is periodically run by the server while the processor is running, indicated by the state of the private flags set by the debugger. The last line of this macro must be `TAPOp_AccessDR_RW` or `ARMTAP_AccessDR_RW`. The result of the data read from this function is ANDed with `eventMask` and XORed with `eventXOR`. If the result is zero, an event is recognized. When the server recognizes an event, it runs the `stopMacros` for other processors if marked for synchronized stopping on the server.

The typical use of `eventMacro` is to recognize a breakpoint condition. However, this is not the only possible use. Any event that can be recognized using a scan chain read and mask can be set up as a trigger for `stopMacro` (and `stopMacro` does not have to be written to stop the processor).

preExecMacro

If the `TAPOp_PreExecMacroRequired` flag is set, this macro is run before the `executeMacro` because it is used to set up the processor state before actually starting it. The server runs the supplied macros in the following order:

1. All the `preExec` macros are run one by one for each processor.
2. All the execute macros that can be merged are run in one pass of `Run-Test/Idle`.
3. Any other execute macros are run.
4. All the `postExec` macros are run one by one for each processor.

All of these steps are completed in one block. No other TAPOp operations take place during these steps. The sequence commences when all the processors are ready to start. This happens when the debuggers for all the processors that require sync start have set TAPOp_ProcStartREQ.

executeMacro

This macro is run by the server to start execution of the processor. If the last line of the macro is an IR write and the operation causes the TAP state machine to pass through Run-Test/Idle (as is the case on an ARM processor), the IR writes from all the processors are merged. The whole set of IR registers in the scan chain is written in one operation, and the resulting pass through Run-Test/Idle starts all the processors on the same **TCK**.

postExecMacro

If the TAPOp_PostExecMacroRequired flag is set, this macro is run by the server immediately after the executeMacro because it must be used to tidy up after executeMacro. Because the last line of executeMacro has some constraints (as described in the text for executeMacro), this extra macro can be used to sort out any loose ends. For example, reselecting scan chains and putting INTEST in the IR.

stopMacro

This macro is run to stop the processor. See the description of eventMacro in *Usage* on page 2-121.

Note

The event, stop, and combined execute macros must all start and stop in a known state, because they can be run in any order by the server.

Any TAPOp operations that a client makes while the processor is running with synchronous stopping enabled must also start and stop in this same state, and must also not deselect until these have finished (to ensure that running one of these macros cannot get in at an inappropriate point).

Example

This example gives an outline of the steps you must take to use the TAPOp_SetControlMacros function.

```
#define ARM9_EVENTMAC  21
#define ARM9_PREEXECMAC  22
#define ARM9_EXECMAC  23
#define ARM9_POSTEXECMAC  24
```



```
#define ARM9_STOPMAC  25
TAPOp_Error defineMacros(void)
{
    ... define ARM9_EVENTMAC
    ... define ARM9_PREEXECMAC
    ... define ARM9_EXECMAC
    ... define ARM9_POSTEXECMAC
    ... define ARM9_STOPMAC
    ... define any other macros for this device
}
TAPOp_Error initialiseTarget(void)
{
    ... make a connection to the target
    if (defineMacros() == TAPOp_NoError) {
        TAPCheck(TAPOp_SetControlMacros(connectId, flags,
                 ARM9_EVENTMAC, ARM9_PREEXECMAC, ARM9_EXECMAC,
                 ARM9_POSTEXECMAC, ARM9_STOPMAC, andMask, xorMask));
        ... other initializations
    }
}
```

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_ReadPrivateFlags* on page 2-103
- *TAPOp_WriteCommonData* on page 2-133
- *TAPOp_WritePrivateFlags* on page 2-139.

2.5.38 TAPOp_SetLogging

Switches debug logging for TAPOp functions on or off, for a particular TAP controller.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_SetLogging(unsigned8 connectId, unsigned32 flags)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

flags If 0, switch logging off, otherwise switch it on.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing thisrequest.

`TAPOp_CannotEnableLogging`

The server could not enable logging.

Usage

This call does not require a selected connection to a TAP controller. It changes the level of debugging output and returns.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_LogString* on page 2-93
- *TAPOp_DisplayMacro* on page 2-82.

2.5.39 TAPOp_SystemResetSignal

Sets or clears the **nSRST** signal.

Note

This call replaces the now obsolete `TAPOp_SetSysResetSignal`.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_SystemResetSignal(unsigned8 connectId, unsigned8 level,
                  unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

level Specifies whether to set or clear System Reset:

- 0 clears (deasserts) theSystem reset signal
- nonzero sets (asserts) the System reset signal.

deselect If 0, the connection to this TAP controller remains selected, so excluding access by other TAP controller connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The call enables a client to reset the target system logic, but not the TAP controller logic, by asserting the system reset signal (**nSRST)** on the JTAG connector. To do this, you must first assert **nSRST** by calling `TAPOp_SystemResetSignal` with `level = 1`, then delay for several milliseconds, then deassert **nSRST** by calling `TAPOp_SystemResetSignal` with `level = 0`.

When setting and then clearing the system reset signal, the Multi-ICE server automatically clears the sticky reset bit in the status register. This makes it possible for applications to perform a system reset and then continue with the same connection without the server forcing a disconnect and reconnect to take place.

If you want the server to take account of the system reset and force this (and other) connections to disconnect and reconnect, call `TAPOp_ReadMICEFlags` while system reset is asserted.

Note

- Resetting the TAP controller for any one device resets all TAP controllers on that scan chain. It is not possible to reset only one TAP controller in a chain.
- Some target hardware is wired, incorrectly, with the **nTRST** and **nSRST** JTAG signals connected together. If this is the way the hardware you are using is connected, you cannot reset the target system logic without resetting the TAP controller logic.

Example

This example shows a sequence of API calls that resets the TAP controllers on the scan chain.

```
{
    ... declarations
    /* assert nSRST, and keep connection selected */
    TAPCheck(TAPOp_SystemResetSignal(connectId, 1, 0));
    /* wait for hardware to respond */
    Sleep(20);                          /* Win32 function that waits 20ms */
    /* deassert nSRST, and enable the connection to be deselected */
    TAPCheck(TAPOp_SystemResetSignal(connectId, 0, 1));
```


See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_TestResetSignal* on page 2-128.

2.5.40 TAPOp_TestResetSignal

Assert and release the **nTRST** reset signal on the JTAG interface.

Note

This call replaces the now obsolete `TAPOp_SetNTRSTSignal`.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_TestResetSignal(unsigned8 connectId, unsigned8 level,
                  unsigned8 deselect)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

level Specifies whether to set or clear **nTRST** resetsignal:

- 0 clears the **nTRST** reset signal, taking the TAP controller logic out of RESET
- Nonzero sets the **nTRST** reset signal, putting the TAP controller logic into RESET.

deselect If 0, the connection to this TAP controller remains selected, so excluding access by other TAP controller connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The call enables a client to reset the TAP controller logic by asserting the TAP controller reset signal (**nTRST)** on the JTAG connector. To do this, you must first assert **nTRST** by calling `TAPOp_TestResetSignal` with `level = 1`, then delay for several milliseconds, then deassert **nTRST** by calling `TAPOp_TestResetSignal` with `level = 0`.

Note

- Resetting the TAP controller for any one device resets all TAP controllers on that scan chain. It is not possible to reset only one TAP controller in a chain.
- Some target hardware is wired, incorrectly, with the **nTRST** and **nSRST** JTAG signals connected together. If this is the way the hardware you are using is connected, you cannot use this function. Instead, you must use the `Test-Logic Reset` state in the TAP controller state machine.

If a client connection makes a call to `TAPOp_TestResetSignal` with `level = 1` but with `deselect = 1`, the Multi-ICE server automatically deasserts the reset and returns the error `TAPOp_BadParameter`. This avoids failures on other connections.

Note

Do not use the combination of `level = 1` and `deselect = 1` to avoid a second TAPOp call. The delay is unlikely to be long enough to reset the hardware properly.

When a call to `TAPOp_TestResetSignal` causes the TAP controller to come out of reset, the Multi-ICE server reinitializes all the other connections to this same scan chain and forces an acknowledgement of the reset, in the same way as when `TAPOp_AnySequence_W` resets the TAP controller. Additionally, after a reset caused by `TAPOp_TestResetSignal`, all TAP controllers are left in the `Run-Test/Idle` state. See *TAPOp_AnySequence_W* on page 2-68 for more details.

Example

This example shows a sequence of API calls that resets the TAP Controllers on the scan chain.

```
{
    ... declarations
    /* assert nTRST, and keep connection selected */
    TAPCheck(TAPOp_TestResetSignal(connectId, 1, 0));
```



```
/* wait for hardware to respond */
Sleep(20);                          /* Win32 function that waits 20ms */
/* deassert nTRST, and enable the connection to be deselected */
TAPCheck(TAPOp_TestResetSignal(connectId, 0, 1));
```

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_SystemResetSignal* on page 2-125.

2.5.41 TAPOp_Wait

A variable delay for use in macros.

Syntax

```
#include “tapop.h”

extern TAPOp_ErrorTAPOp_Wait(unsigned8 connectId, unsigned32 us_delay)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

us_delay Delay in microseconds.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_WaitTooLong`

`us_delay` was more than 1 second.

Usage

This function provides a variable delay for use in macros where time delays must be provided between scan chain accesses.

Before the time delay begins, the pending scan buffer is flushed. The delay that follows is never less than the given value. This ensures that enough time is given between accesses. The maximum allowed wait is one second. This is so that errors do not lock up the server for long.

The selection status of the connection is irrelevant (this call cannot return `TAPOp_UnableToSelect`), and the selection status is not modified by this call unless the wait is long enough to provoke an error.

Compatibility

This function first appeared in Multi-ICE Version 1.4.

See also

This TAPOp API function provides similar or related functionality:

- *TAPOp_DefineMacro* on page 2-75.

2.5.42 TAPOp_WriteCommonData

Writes to the data block that is common to all the clients connected to the server.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_WriteCommonData(unsigned8 connectId,
                   unsigned32 *commonBlk, unsigned8 deselect)
```

where:

connectId Connection ID, as returned by TAPOp_OpenConnection.

commonBlk Four words of common data, with a meaning definedby the client.

deselect If 0, the connection to this TAP controller remains selected, so excluding access by other TAP controller connections. Otherwise, the connection is deselected, giving other connections a chance to perform operations.

Returns

The function returns:

TAPOp_NoError

No error.

TAPOp_UnableToSelect

Connection could not be made.

TAPOp_NoSuchConnection

The connectId was not recognized.

TAPOp_RPC_Connection_Fail

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a TAPOp_UnableToSelect error.

To safely write to the common data block, use an atomic Read-Modify-Write sequence. You can make a call of TAPOp_ReadCommonData and TAPOp_WriteCommonData atomic by setting the deselect parameter of the TAPOp_ReadCommonData call to zero.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_ReadPrivateFlags* on page 2-103
- *TAPOp_WriteCommonData* on page 2-133
- *TAPOp_WritePrivateFlags* on page 2-139.

2.5.43 TAPOp_WriteMICEUser1

Sets the level of the user-defined Multi-ICE User 1 output bit.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_WriteMICEUser1(unsigned8 connectId, unsigned8 user1)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

user1 Bit 0 indicates the state to which the User 1 signalis set.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

`TAPOp_NotAllocatedToThisConnection`

The User Output bit is not allocated to this connection by the Multi-ICE server.

Usage

For this procedure to have an effect on the state of the output bits, you must choose the **Set by Driver** option in the server **User Output bit** settings dialog box. You must also choose the TAP position for the connection.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_WriteMICEUser2* on page 2-137
- *TAPOp_ReadMICEFlags* on page 2-101.

2.5.44 TAPOp_WriteMICEUser2

Sets the level of the user-defined Multi-ICE User 2 output bit.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_WriteMICEUser2(unsigned8 connectId, unsigned8 user2)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

user2 Bit 0 indicates the state to which the User 2 signalis set.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

`TAPOp_NotAllocatedToThisConnection`

The User Output bit is not allocated to this connection by the Multi-ICE server.

Usage

For this procedure to have an effect on the state of the output bits, you must choose the **Set by Driver** option in the server **User Output bit** settings dialog box. You must also choose the TAP position for the connection.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_WriteMICEUser1* on page 2-135
- *TAPOp_ReadMICEFlags* on page 2-101.

2.5.45 TAPOp_WritePrivateFlags

To write the private word of flags for this processor.

Syntax

```
#include “tapop.h”

extern TAPOp_Error TAPOp_WritePrivateFlags(unsigned8 connectId,
                   unsigned32 flags)
```

where:

connectId Connection ID, as returned by `TAPOp_OpenConnection`.

flags Private word of flags. The flag bits are describedin *TAPOp inter-client communication flags* on page 2-22.

Return

The function returns:

`TAPOp_NoError`

No error.

`TAPOp_UnableToSelect`

Connection could not be made.

`TAPOp_NoSuchConnection`

The `connectId` was not recognized.

`TAPOp_RPC_Connection_Fail`

The RPC connection was lost while processing this request.

Usage

The call attempts to select the connection. If this cannot be done (for example, because another TAP controller is being accessed), the call fails with a `TAPOp_UnableToSelect` error.

The flags are private to each connection and are used by the Multi-ICE server to signal events to each client. For example, one of the flags indicates that the device has been reset. The description for each flag in *TAPOp inter-client communication flags* on page 2-22 indicates how each is used, and when it is appropriate to use `TAPOp_WritePrivateFlags` to change a flag.

See also

These TAPOp API functions provide similar or related functionality:

- *TAPOp_ReadMICEFlags* on page 2-101
- *TAPOp_ReadPrivateFlags* on page 2-103
- *TAPOp_WriteCommonData* on page 2-133
- *TAPOp_WritePrivateFlags* on page 2-139.

Glossary

Adaptive Clocking A technique in which a clock signal is sent out by Multi-ICE and it waits for the returned clock before generating the next clock pulse. The technique allows the Multi-ICE interface unit to adapt to differing signal drive capabilities and differing cable lengths.

ADS See *ARM Developer Suite*.

ADU See *ARM Debugger for UNIX*.

ADW See *ARM Debugger for Windows*.

Angel Angel is a debug monitor that runs on an ARM-based target and enables you to debug applications.

ARM Debugger for UNIX *ARM Debugger for UNIX* (ADU) and *ARM Debugger for Windows* (ADW) are two versions of the same ARM debugger software, running under UNIX or Windows respectively.

ARM Debugger for Windows *ARM Debugger for Windows* (ADW) and *ARM Debugger for UNIX* (ADU) are two versions of the same ARM debugger software, running under Windows or UNIX respectively. This debugger was issued originally as part of the ARM Software Development Toolkit.

ARM Developer Suite A suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.

ARM eXtended Debugger
The *ARM eXtended Debugger* (AXD) is the latest debugger software from ARM that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. AXD is supplied in both Windows and UNIX versions.

armsd
The *ARM Symbolic Debugger* (armsd) is a command-line, interactive, source-level debugger providing debug support for C and assembly language programs.

ARMulator
ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.

AXD
See *ARM eXtended Debugger*.

Big-endian
Memory organization where the least significant byte of a word is at a higher address than the most significant byte. See *Little-endian*.

Coprocessor
An additional processor that is used for certain operations, for example, for floating-point math calculations, signal processing, or memory management.

Core Module
See Integrator

CPU
Central Processor Unit.

CPSR
Current Program Status Register. See *Program Status Register*.

DLL
See Dynamic Linked Library

Double word
A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Dynamic Linked Library
A collection of programs, any of which can be called when needed by an executing program. A small program that helps a larger program communicate with a device such as a printer or keyboard is often packaged as a DLL.

ECP
See Enhanced Capability Port.

Enhanced Capability Port
A standard for parallel ports which enables fast bidirectional data transfers over parallel ports.

See also EPP and IEEE1284.

Enhanced Parallel Port
A standard for parallel ports which enables fast bidirectional data transfers over parallel ports.

See also ECP and IEEE1284.

EPP
See Enhanced Parallel Port.

Environment
The actual hardware and operating system that an application will run on.

ETM
Embedded Trace Macrocell.

External Data Representation A specification defined by Sun Microsystems describing a way of transferring typed data between computer systems in a system independent manner. Used by Sun RPC.

Flash memory Nonvolatile memory that is often used to hold application code.

Halfword A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Heap The portion of computer memory that can be used for creating new variables.

Host A computer which provides data and other services to another computer. *Especially*, a computer providing debugging services to a target being debugged.

ICE *See* In Circuit Emulator.

ID Identifier.

IEEE 1149.1 The IEEE Standard which defines TAP. Commonly (but incorrectly) referred to as JTAG.

IEEE 1284 A standard for parallel port interfaces which encompasses ECP but extends it to enable semi-autonomous transfers.

Image An executable file that has been loaded onto a processor for execution.

In-Circuit Emulator A device enabling access to and modification of the signals of a circuit while that circuit is operating.

Instruction Register When referring to a TAP controller, a register that controls the operation of the TAP.

Integrator An ARM hardware development platform. Core Modules are available that contain the processor and local memory.

IR *See* Instruction Register.

Joint Test Action Group The name of the standards group which created the IEEE 1149.1 specification.

JTAG *See* Joint Test Action Group.

Little-endian Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also *Big-endian*.

Memory management unit Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.

MMU *See* Memory Management Unit.

Multi-ICE Multi-processor EmbeddedICE interface. ARM registered trademark.

nSRST Abbreviation of *System Reset*. The electronic signal which causes the target system other than the TAP controller to be reset. This signal is known as **nSYSRST** in some other manuals.

See also **nTRST**.

nTRST Abbreviation of *TAP Reset*. The electronic signal that causes the target system TAP controller to be reset. This signal is known as **nICERST** in some other manuals.

See also **nSRST**.

Open collector A signal that may be actively driven LOW by one or more drivers, and is otherwise passively pulled HIGH. Also known as a "wired AND" signal.

PID The ARM Platform-Independent Development card, now known as the ARM Development Board.

PIE A platform-independent evaluator card designed and supplied by ARM Limited.

Port mapper A process that enables RPC client processes to contact the RPC server process for a particular RPC service.

Processor Core The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.

Processor Status Register See *Program Status Register*.

Program image See Image.

Program Status Register Program Status Register (PSR), containing some information about the current program and some information about the current processor. Often, therefore, also referred to as *Processor Status Register*.

Is also referred to as *Current PSR* (CPSR), to emphasize the distinction between it and the *Saved PSR* (SPSR). The SPSR holds the value the PSR had when the current function was called, and which will be restored when control is returned.

RDI *See* Remote Debug Interface.

Remapping Changing the address of physical memory or devices after the application has started executing. This is typically done to allow RAM to replace ROM once the initialization has been done.

Remote_A Remote_A is a software protocol converter and configuration interface. It converts between the RDI 1.5 software interface of a debugger and the Angel Debug Protocol used by Angel targets. It can communicate over a serial or Ethernet interface.

Remote Debug Interface RDI is an open ARM standard procedural interface between a debugger and the debug agent. The widest possible adoption of this standard is encouraged.

Remote Procedure Call A call to a procedure in a different process. The calling procedure invokes a procedure in a different process that is usually running on a different processor.

RM RealMonitor.

RPC *See* Remote Procedure Call.

RTCK Returned **TCK**. The signal which enables Adaptive Clocking.

RTOS Real Time Operating System.

Scan Chain A group of one or more registers from one or more TAP controllers connected between TDI and TDO, through which test data is shifted.

Semihosting A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.

SPSR Saved Program Status Register. See *Program Status Register*.

SWI Software Interrupt. An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.

Synchronous starting Setting several processors to a particular program location and state, and starting them together.

Synchronous stopping Stopping several processors in such a way that they stop executing at the same instant.

Sun RPC A specific form of RPC defined as a standard by Sun Microsystems that uses the XDR standard and TCP/IP datagrams to communicate between networked computers.

TAP *See* Test Access Port.

TAP Controller Logic on a device which allows access to some or all of that device for test purposes. The circuit functionality is defined in IEEE1149.1.

See also TAP, IEEE1149.1.

TAP position An integer, starting at zero, that is used to identify a particular TAP controller on a given scan chain. It does not uniquely identify a device because it is possible for two devices to share a TAP controller. However, shared controllers (such as an ARM9TDMI and an ETM9) are usually distinguishable in other ways.

TAPOp The name of the interface API between the Multi-ICE Server and its clients.

Target The actual processor (real silicon or simulated) on which the application program is running.

TCK The electronic clock signal which times data on the TAP data lines TMS, TDI, and TDO.

TDI The electronic signal input to a TAP controller from the data source (upstream). Usually this is seen connecting the Multi-ICE Interface Unit to the first TAP controller.

TDO The electronic signal output from a TAP controller to the data sink (downstream). Usually this is seen connecting the last TAP controller to the Multi-ICE Interface Unit.

Test Access Port The port used to access a device's TAP Controller. Comprises TCK, TMS, TDI, TDO and nTRST (optional).

TTL Transistor-transistor logic. A type of logic design in which two bipolar transistors drive the logic output to one or zero. LSI and VLSI logic often used TTL with HIGH logic level approaching +5V and LOW approaching 0V.

Watchpoint A location within the image that will be monitored and that will cause execution to stop when it changes.

Word A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

XDR *See External Data Representation.*

Index

F

H

I

K

L

M

N

O

P

R

S

T

U

W

Numerics

