Universal Debugger Interface (UDI) Specification

Version 1.4, Revision 3

Universal Debugger Interface (UDI) Specification
Version 1.4, Revision 3
Last Update: August 17, 1995

Advanced Micro Devices, Inc.
5204 E. Ben White Blvd.
Austin, TX 78741-7399

Contents

About the UDI Specification

Chapter 1 Introduction to the Universal Debugger Interface (UDI)

Chapter 2 UDI Services Overview

Chapter 3 UDI Services

Chapter 4 UDI IPC Methods for DOS Hosts

Chapter 5 UDI IPC Methods for UNIX Hosts

Chapter 6 UDI Developer's Toolkit

Appendix A UDI Error Numbers

Appendix B UDI Configuration Files

Appendix C 29K Family UDI Resource Spaces

Appendix D Compatibility of UDI 1.4, 1.3, and 1.2 DFEs and TIPs

About the UDI Specification

The Universal Debugger Interface (UDI) was created to provide an interface between a debugger and a target which allows the two to be developed, implemented, maintained, and shipped separately. In this specification, we refer to these two separately built pieces as the Debugger Front End (DFE) and the Target Interface Process (TIP). UDI allows any DFE and TIP to communicate using a set of functions called UDI services. Figure 0–1 illustrates the communication which occurs between DFEs and TIPs through UDI.

The UDI specification defines:

- A C language procedural interface for each of the UDI services and provides a description of the semantics of the service and each of its parameters

- Inter–Process Communication (IPC) methods which provide the message format for each UDI service and the message passing mechanism for a particular host or operating system environment

The procedural interface and the semantics of both the services and the parameters are the same for all IPC methods. The procedural interface is defined in Chapters 2 and 3 of this specification.

UDI IPC methods have been defined for both DOS and UNIX hosts. These IPC methods are described in Chapters 4 and 5 of this specification.

UDI Documentation

This documentation is written for programmers using UDI to create portable debuggers and target interface processes (TIPs). Anyone using UDI should read Chapter 1 through Chapter 3.

AMD supplies sample IPC implementation code which maps the UDI procedural interface to the IPC mechanism for the following host environments: DOS IPC on DOS real–mode hosts, UNIX IPC on UNIX big–endian hosts. Read Chapter 6 concerning the UDI Developer's Toolkit if you intend to use the IPC sample code provided. (If you plan to use the sample code without modifying it, there is no need to read Chapters 4 and 5.)

If you want to modify the sample IPC code or port it to a different environment, or if you simply want to understand more about IPC methods for DOS hosts, read Chapter 4. For information on IPC methods for UNIX hosts, read Chapter 5.

About This Specification

- Chapter 1: "Introduction" describes the fundamental concepts behind the development of UDI.
- Chapter 2: "Services Overview" describes the services available with UDI.
- Chapter 3: "UDI Services" documents all UDI services.
- Chapter 4: "UDI IPC Methods for DOS Hosts" specifies how UDI DFEs and TIPs communicate using the DOS IPC mechanism.
- Chapter 5: "UDI IPC Methods for UNIX Hosts" specifies how UDI DFEs and TIPs communicate using the socket–based IPC mechanism.
- Chapter 6: "UDI Developer's Toolkit" describes the toolkit available for development of UDI–compliant DFEs or TIPs.
- Appendix A: "UDI Errors" lists the error codes that UDI–conforming services may return.
- Appendix B: "UDI Configuration Files" outlines the format of the UDI configuration files for MS–DOS and UNIX hosts.

- Appendix C: “29K Family UDI Resource Spaces” describes the resource spaces that are predefined when UDI is applied to targets using the AMD 29K Family of microprocessors.

- Appendix D: “Compatibility of UDI 1.4, 1.3, and 1.2 DFEs and TIPs” lists the precautions and restrictions involved with interoperations between TIPs and DFEs from different versions of UDI.

Suggested Reference Material

The following AMD documents may be of interest:

- *Am29000*t *and Am29005*t *User's Manual and Data Sheet*
  Advanced Micro Devices, order number 16914A.

- Am29030t and Am29035t Microprocessors User's Manual and Data Sheet
  Advanced Micro Devices, order number 15723B

- *Am29050*t *Microprocessor User's Manual*
  Advanced Micro Devices, order number 14778A

- *Am29050*t *Data Sheet*
  Advanced Micro Devices, order number 15039A.

- Am29200t RISC Microcontroller User's Manual and Data Sheet
  Advanced Micro Devices, order number 16362B

- *Am29205*t *RISC Microcontroller Data Sheet*
  Advanced Micro Devices, order number 17198A

- Am29240t, Am29245t, and Am29243t RISC Microcontrollers User's Manual and Data Sheet
  Advanced Micro Devices, order number 17741A

Documentation Conventions

The *Universal Debugger Interface (UDI) Specification* uses the conventions shown in Table 0–1 (unless otherwise noted). These same conventions are used in all the 29K Family support product manuals.

Symbol	Usage
Boldface	Indicates that characters must be entered exactly as shown, except that the alphabetic case is only significant when indicated.
Italic	Indicates a descriptive term to be replaced with a user–specified term.
`Typewriter face`	Indicates computer text input or output in an example or listing.
[ ]	Encloses an optional argument. To include the information described within the brackets, type only the arguments, not the brackets themselves.
{ }	Encloses a required argument. To include the information described within the braces, type only the arguments, not the braces themselves.
..	Indicates an inclusive range.
...	Indicates that a term can be repeated.
|	Separates alternate choices in a list — only one of the choices can be entered.
:=	Indicates that the terms on either side of the sign are equivalent.

Table 1. Notational Conventions

Chapter 1

Introduction to the Universal Debugger Interface (UDI)

This chapter describes the terms used in this specification to describe various parts of the overall solution the Universal Debugger Interface (UDI) provides, the problems that UDI attempts to solve, and the fundamental concepts used to solve those problems.

UDI Terms

This specification refers to the debugger as the Debugger Front End (DFE). Early conversations about UDI revolved around splitting the debugger into a front–end (user interface) and the target interface (execution interface). This target interface later became known as the target interface process (TIP). Referring to it as a process implied that the TIP was not linked with the DFE into a single executable. UDI exclusively specifies the interface between the DFE and the TIP. The term *target* refers to the actual execution vehicle that runs the program under control of the DFE, via the TIP. Only the TIP knows how to control the target.

Host refers to the computer system on which the DFE resides. Usually, the TIP also resides on the host, but the communications method defined for some hosts may allow the TIP to reside on a different computer system. Throughout this document, we assume the TIP and DFE run on the same computer (i.e., the host).

Problems UDI Solves

In many cases, a company provides multiple debuggers, targets, or both. The problem such a company faces is that any time an update is made to a complicated debugger, it must be rebuilt with the code that allows it to communicate with each of the possible targets. All debugger/target combinations must be retested and updates supplied to all affected customers.

Additionally, end customers of embedded systems inherently want to use debuggers with their custom hardware. While in–circuit emulators have been one solution to this problem in the lab, many customers would like to attach a debugger to their hardware without an in–circuit emulator. Often, that debugger does not support the only communications path to the hardware.

Finally, sometimes it is desirable to mix debuggers from one company with targets from another. For example, an emulator company may not be able to justify supporting a little–used debugger (or vice versa), but a customer may decide that such a configuration is best.

The fundamental goal of UDI is to provide an interface between a debugger and its target so that the two can be developed, implemented, maintained, and shipped separately. It should be possible to use any UDI–compliant debugger with any UDI–compliant target. Also, end users should be able to develop either custom debuggers for use with standard targets or, more probably, custom targets for use with standard debuggers.

UDI Concepts

Most of the companies involved in specifying UDI are concerned with cross–debugging, that is, using a computer of one type to debug a system of a different type. Cross–debugging involves communication between the computer that the debugger runs on (the *host*) and the system that the program the debugger is debugging runs on (the *target*). Unfortunately, there can be no strict standards for this communications path. For example, some targets communicate via the host computer's bus, some via RS–232, some via Ethernet, some via SCSI, and some have no communications path at all (simulators).

UDI works by providing interprocess communication (IPC) methods, which allow separately built DFEs and TIPs to communicate. The IPC method defines the basic communication method and the message format for each UDI service.

This specification describes a procedural interface for each of the UDI services and defines the semantics of the service and each of its parameters. The procedural interface and the semantics of the services are the same for all IPC methods.

UDI IPC methods have been defined for the following hosts:

- DOS
- UNIX sockets

The UDI IPC methods are defined in Chapters 4 and 5.

AMD provides sample code that maps the procedural interface defined in this specification to either of the two UDI IPC methods listed above. This sample code is described in Chapter 6.

UDI attempts to solve some problems that at first may not be obvious, for example:

- Using a single debugger to debug multiple processes running on a single target processor
- Using a single debugger to debug processes running on multiple target processors
- Using a target capable of supporting multiple debuggers

UDI should support a debugger that attaches to a running target. In–circuit emulators usually support advanced logic analysis features and an attempt is made to abstract a few of these capabilities. Finally, UDI should be usable in a more conventional non–cross, non–remote environment.

One issue that UDI does not address is target access to host resources (other than simple terminal access implemented in the programmed I/O service group). Because the TIP resides on the same host as the DFE, the TIP is charged with managing host resource access from the target. Consequently, UDI does not have to abstract wide operating system services across the interface.

Chapter 2

UDI Services Overview

This chapter provides an overview of the services available with UDI. Technically, the TIP and the DFE execute simultaneously. However, debugging is usually an interactive process in which the DFE is in control of the execution. Because of this, the majority of the UDI services are functions that are available to the DFE and implemented by the TIP. Starting with UDI 1.3, a few new services were added in which the TIP calls back to the DFE. These callback services can only be called by the TIP while the TIP is in the process of servicing a UDI request from the DFE, not while the UDI connection is quiet. In addition, the DFE, while servicing a UDI DFE callback service, may occasionally need to call another UDI service to the TIP, thus nesting one level further. Further nesting is theoretically possible, but is unlikely.

All services, whether DFE to TIP or TIP to DFE, return as soon as the requested information is available or the services are performed (except the special case of **UDIWait**). Not all services or variations of parameters are supported by all TIPs, while some functions are supported by every TIP. See Chapter 3 for implementation requirements and options.

UDI Session Management

Session management establishes DFE–to–TIP connections. The methods support one–to–one, one–to–many, many–to–one, and many–to–many DFE–to–TIP relationships. Some TIP configuration issues are handled here as well.

Services in this group and their functions are:

UDIConnect	Establishes initial connection.
UDIDisconnect	Tears down a connection and frees the TIP.
UDISetCurrentConnection	Used only by DFEs to identify multiple connections. All other UDI services are performed against the currently connected TIP.
UDICapabilities	Provides information between the DFE and the TIP.
UDIEnumerateTIPs	Used by DFEs to obtain a list of TIPs from the TIP ID file to present to the user.

TIP configuration issues such as the name of the TIP program and TIP startup parameters are specified in the configuration string passed to **UDIConnect** by the DFE. The interpretation of the configuration string is specific to a particular IPC method.

Process Management

Process management in UDI accommodates a wide range of DFE and TIP design goals. Some TIPs are best thought of as raw machine debuggers, providing complete access to all of the target's resources. Simulators and emulators are typically raw machine debuggers. Other TIPs provide access only to resources created explicitly for a process or distinguish between raw mode and a program debug mode via some other means. Similarly, some DFEs are designed to be raw machine debuggers and others are better thought of as program debuggers.

UDI uses the services in this section to allow access to either a single process or to the raw machine. The distinction is made by the value of the current process. When the current process is the special value, **UDIProcessProcessor**, then the resources accessed will be those of the raw machine. Note also that a **UDIInitializeProcess** request, when the current process is **UDIProcessProcessor**, is a request to reset the entire target system (or come as close as possible to a reset). Whether the TIP allows access to the raw machine is, of course, something the TIP decides.

Most debugging occurs in the context of a process. DFEs can create, initialize, and destroy processes. Processes can be executed and stepped, and breakpoints can be set within them. The services available to DFEs to control processes are:

UDICreateProcess	Is called when a DFE starts up a new connection. **UDICreateProcess** also allows multi–tasking TIPs to create a new process context.
UDISetCurrentProcess	Identifies to the TIP which of several possible processes the rest of the UDI services should apply against.
UDIDestroyProcess	Indicates that debugging of the current process is finished.
UDIInitializeProcess	Restarts a process already established. Any information the TIP maintains about the process (such as pass counts remaining on breakpoints) should be re–initialized when this service is invoked.
UDIExecute	Continues execution of the current process and returns when execution has been started. (It does not wait until execution is finished.) Execution is concurrent with DFE execution.
UDIStep	Executes one or more single steps of the current process, possibly excluding calls to other functions and/or traps.
UDIStop	Stops execution of the current process regardless of where it is.
UDIWait	Is called when the DFE requests the current state of TIP execution.
UDISetBreakpoint	Establishes a breakpoint in the TIP.
UDIClearBreakpoint	Clears a breakpoint. If Breakpoint ID = 0, clears all breakpoints.
UDIQueryBreakpoint	Determines the currently active breakpoints.

To provide the broadest possible range of connectivity between various DFEs and TIPs, UDI defines a specific set of rules for process management.

When a DFE starts up a new connection, it should always call **UDICreateProcess**. TIPs that support only raw machine debugging (type 0 TIPs) return a process ID (PId) of **UDIProcessProcessor**. TIPs that support only program debugging (type 1 TIPs) must return a different PId. TIPs that differentiate between program and machine resources (type 2 TIPs) should also return a PId other than **UDIProcessProcessor**. TIPs that return process **UDIProcessProcessor** in response to a **UDICreate** call can still support OS services. Note, also, that **UDICreateProcess** has the side effect of setting the current process as the one created.

In most cases, the DFE downloads the program to be debugged. In all cases, the program to be debugged is downloaded into a process space compatible with the TIP. If the current process is **UDIProcessProcessor**, the DFE simply sets the PC to the downloaded program's entry point and is ready to execute. (A **UDIInitializeProcess** when the process is **UDIProcessProcessor** performs a target reset).

If the current process is not **UDIProcessProcessor**, then the DFE should call **UDIInitializeProcess**. The TIP initializes the process so that the next instruction to be executed is the first instruction in the process (i.e., the entry point). The TIP determines what happens to any other **UDIInitializeProcess** parameters.

DFEs that debug programs rather than the raw machine can now inspect the PC (using UDI PC space). If the PC is not at the entry point of the program, the DFE sets a breakpoint at the entry point and executes up to it. Otherwise, stepping the TIP may result in executing one or more instructions that the DFE did not download, and this may confuse the DFE.

As stated earlier, **UDICreateProcess** also allows multi–tasking TIPs to create a new process context. For TIPs that allow multiple processes, but do not allow run–time creation of processes, this call would still be used to determine whether a process can be debugged. A process is executed in the context of an operating environment. When **UDIInitializeProcess** is called, command line arguments for the process are passed. The TIP determines what to do with the command line arguments.

Because **UDIStep** may possibly exclude calls to other functions and/or traps, stepping may take a significant amount of time. As a result, **UDIStep**, like **UDIExecute**, returns as soon as stepping has begun. Both **UDIStep** and **UDIExecute** stop when a breakpoint is hit, when the TIP can no longer continue executing the process, or when **UDIStop** is called.

For some TIPs on some hosts (for example, a DOS machine with a simulator for a TIP), all execution occurs when the DFE calls **UDIWait**. Consequently, a DFE may not assume that any progress has been made by the sequence: **UDIExecute**, long delay, **UDIStop** (and a DFE must call **UDIWait** to ensure progress). If the process is stopped, **UDIWait** returns the reason that the process stopped.

Resource Access

The services in this group provide read and write access to target resources. Services are available to move from host to target, to move from target to host, to move from target to target, and to search target memory. A simple search of target memory capability is also available.

The resource access functions are:

UDIRead and **UDIWrite**	Provide access to TIP resources[1], including memory and registers.
UDICopy	Produces target copies between resources. This can be used to copy or fill memory by use of the direction parameter.
UDIFind	Tells the TIP to report the occurrences of a specified pattern in a given range of memory or other resource. A pattern mask can be used to further qualify the search.

Programmed I/O

Services in this group allow the user to communicate directly with a program being debugged. Because the DFE ideally has complete control of the user interface, while the TIP provides all target operating system services, the UDI services in this group allow the DFE to manage the user interface for the process.

[1] Some TIPs may include additional resources, such as queues or semaphores associated with an operating system, or special hardware features associated with an emulator. A DFE must be TIP–aware to use these additional resources.

The programmed I/O functions are:

UDIGetStdout	Is invoked when **UDIWait** returns and indicates that access to stdout is needed. The DFE then performs I/O for the TIP.
UDIGetStderr	Is invoked when **UDIWait** returns and indicates that access to stderr is needed. The DFE then performs I/O for the TIP.
UDIPutStdin	Is invoked when **UDIWait** returns and indicates that access to stdin is needed. The DFE then performs I/O for the TIP.
UDIStdinMode	Changes the mode by which characters are fetched from the user.

When **UDIWait** returns, it may indicate that access to standard input, standard output, or standard error is needed. The DFE then invokes the appropriate UDI service and performs the I/O for the TIP. Once the I/O is completed, the TIP automatically continues execution (if it has stopped) and the DFE should again call **UDIWait**.

UDIStdinMode changes the mode by which characters are fetched from the user. Echoed and non–echoed input are supported. Line buffered and unbuffered input are also supported. In the absence of a TIP requesting a change, (which can occur only through **UDIWait**), the mode of input reverts to line buffered and echoed each time **UDIInitializeProcess** is called.

Since programmed I/O provides support for standard I/O services, I/O redirection can be performed by the DFE. Note, however, that this does not prevent the TIP from performing I/O redirection. In general, a DFE should support redirection through its user interface, rather than by interpreting the command line of the program being debugged. This helps avoid placing unexpected constraints on the command line of a target operating system.

Transparent Mode

This group of services allows DFEs and TIPs to support more functionality than the other groups of services provide. For example, if a target supports profiling in one of its many forms, none of which are directly supported by other UDI service groups, transparent mode can be used for communication.

DFEs can provide access to these extended TIP services using a simple terminal emulator written using transparent mode services. This set of functions is arranged so that the TIP can manage the I/O process with the user.

In general, a DFE may invoke transparent mode in one of two ways:

- interactive transparent mode (during which the TIP's prompt is displayed and the user can enter any number of commands),

- immediate transparent mode (in which a single command is presented to the TIP and the command's output is presented to the user).

The distinctions between interactive mode and immediate mode are handled entirely on the DFE side. This is possible because the DFE can distinguish the TIP's prompt output from the other TIP output. The code examples at the end of this chapter show examples for both interactive transparent mode and immediate transparent mode.

When a DFE leaves transparent mode, the DFE cannot know what changes have occurred in the target state and thus should flush any cached values that it has read from the target, poll the target state, query breakpoints, etc.

The transparent mode functions are:

UDIGetTrans

The DFE calls **UDIGetTrans** to find out if the TIP has any output and the TIP also uses special returns from **UDIGetTrans** to indicate:

- when the TIP requires input but is not at the prompt (**UDIErrorTransInputNeeded**)
- when the TIP is returning the prompt (**UDIErrorTransPrompt**). A TIP is not required to have a prompt but if it has one it may only be returned
- with **UDIErrorTransPrompt**.
- when the TIP is at the prompt waiting for input (**UDIErrorTransDone**)
- when the TIP has parsed the transparent mode "exit" command. (**UDIErrorTransExit**).
- when the TIP requires the DFE to change input mode (**UDIErrorTransModeX**).

In general, the DFE must continue calling **UDIGetTrans** until the TIP indicates (through the **UDIGetTrans** return code) that the DFE call another transparent function.

The DFE can allow the user to exit from the transparent mode session only when the TIP returns **UDIErrorTransDone** or **UDIErrorTransExit**. A DFE may call **UDIStop** during transparent mode and the TIP, on receiving **UDIStop**, should try to clean up and return **UDIErrorTransDone** as soon as possible, but the DFE is required to continue calling **UDIGetTrans** until **UDIErrorTransDone** or **UDIErrorTransExit** is returned.

UDIPutTrans

If the TIP returns **UDIErrorTransInputNeeded**, the DFE is required to acquire input from the user and call **UDIPutTrans**, sending one block of data to the TIP. Then the DFE should resume calling **UDIGetTrans**.

If the TIP returns **UDIErrorTransDone**, the TIP is at the prompt and the DFE is allowed to leave transparent

mode. If the DFE does not wish to leave transparent mode, it must acquire input from the user and call **UDIPutTrans**, as described above.

A special usage of **UDIPutTrans** (called with a Count=0) tells the TIP that the DFE wants to get the prompt from the TIP on the next **UDIGetTrans** call. If the TIP has a prompt, it returns **UDIErrorTransPrompt** to **UDIPutTrans** and then returns the actual prompt on the next **UDIGetTrans** call (again with the return **UDIErrorTransPrompt**). If the TIP has no prompt, it returns **UDIErrorTransDone** to the **UDIPutTrans** (Count=0) call. If the TIP is not in a state where a prompt is possible, i.e., if the TIP is not between commands, etc., then it returns **UDIErrorCantAccept** to the **UDIPutTrans** (Count=0) call.

UDITransMode If **UDIGetTrans** requests a call to **UDITransMode**, the DFE must call **UDITransMode**, the TIP will return the new mode that it wishes the DFE to use and then the DFE must resume calling **UDIGetTrans**.

Using UDIDFE calls in Transparent Mode

A TIP is allowed to use **UDIDFEGetInput** and **UDIDFEPutOutput** with IOType **UDIIOTypeTip***xxx* when servicing transparent mode calls with the exception that the prompt, if any, can only be returned by **UDIGetTrans**. Other **UDIDFE***xxx* calls may be used by the TIP without restriction during transparent mode.

Because the TIP may mix **UDIDFEGetInput** and **UDIDFEPutOutput** with IOType **UDIIOTypeTip***xxx* with transparent mode handling, the DFE is required to treat transparent mode I/O and **UDIIOTypeTip***xxx* I/O in a similar manner.

Symbolic Mapping

The functions in this group, new with version 1.3 of UDI, are implemented by the DFE and called by the TIP. Generally, these functions would be used by a TIP that is operating in transparent mode.

It may be necessary for a TIP to allow a user to include symbolic expressions in the transparent mode commands. The TIP can then ask the DFE (which controls the symbol table) to map an expression to a value or an address. Alternatively, the TIP, instead of displaying a raw address to the user in the output of a transparent mode command, may need to map that raw address to a symbolic expression. Again, only the DFE can provide this mapping. The TIP asks the DFE to perform these services by "calling back" to the DFE.

The DFE may require TIP services (such as **UDIRead**) during its evaluation and it is legal for the DFE to make any UDI call before returning the answer to the TIP, as long as further transparent mode calls are avoided.

The symbolic mapping functions are:

UDIDFEEvalExpression	Evaluates a symbolic expression returning either a value or a resource address.
UDIDFEEvalResource	Maps a resource address to an ASCII string of the form "symbolname" or "symbolname + offset".

TIP Access to DFE Screen I/O

This set of functions, new with version 1.3 of UDI, is implemented by the DFE and called by the TIP. As with all UDI DFE callback functions, the TIP is only allowed to call these functions while in the process of handling a request from the DFE.

The DFE screen I/O services are:

UDIDFEPutOutput	Directs output to the DFE screen. An *IOType* parameter indicates whether the output is from the TIP itself or from a running target program.
UDIDFEGetInput	Gets input from the DFE, and again, an *IOType* parameter indicates whether the input is for the TIP itself or for the target program.
UDIDFEEndTIPIO	Provides a way for the TIP to indicate a logical boundary for a set of TIP–directed (as opposed to target–directed) I/O requests before returning from the original UDI request made by the DFE.

Possible uses of the functions is this group are:

- The TIP can present information to and get a response from the user to determine how to proceed while handling a UDI request. Some parts of a transparent mode session can be handled by using the **UDIDFEPutOutput** and **UDIDFEGetInput** functions with IOType = IOTypeTIPxxx. (The section on Transparent mode describes when **UDIDFEPutOutput** and **UDIDFEGetInput** calls are allowed to be used during transparent mode).

- The TIP can feed output from a running target program to the DFE or get input from the DFE on behalf of a running target program. In this manner, these calls are an alternative to returning **UDIStdOutReady** or **UDIStdInNeeded** from **UDIWait** and waiting for the DFE to call **UDIGetStdout**, **UDIGetStderr**, or **UDIPutStdin**.

Future Groups

This version of UDI supports basic debugging. Not all of the functionality of some TIPs will be apparent in the current UDI specification. The IPC mechanisms defined for each host are designed to support additional services without invalidating the existing set.

Recommended Usage

The following is an outline of some important sections of code present in most DFEs. It is intended to help implementors understand the relationship between various services. (Error handling code is not shown.) The code shown is for a simple, single–process program debugger.

Startup:

```
UDIConnect()              /* Establish connection with the TIP. */
UDIWait()                 /* See if the connection is a
reconnection to
                           * a running target. */
UDICreateProcess()        /* Simple TIPs return
UDIProcessProcessor,
                           * but the DFE does not care. */
UDIWrite()/UDICopy()      /* As necessary to download the program,
if
                           * required. */
/* At this point control can be given to the user to examine or
 * modify memory, set breakpoints, and so on. */
```

To begin or restart execution from the entry point:

```
if (currentprocess == UDIProcessProcessor) {
    UDIWrite(PC);                /* set PC manually */
}
else {
    UDIInitializeProcess()       /* To pass any arguments to the
program,
                                  * establish the entry point,
etc. */
    UDIRead(PC);                 /* to check where PC is after
                                  * UDIInitializeProcess */

    if (PC not at EntryPoint) {
           UDISetBreakPoint (EntryPoint);
           UDIExecute();
           UDIWait();            /* Until breakpoint is hit */
    }
}

UDIExecute()/UDIStep()    /* As required by the user's command. */
```

While executing:

After issuing a **UDIExecute**() or **UDIStep**() request, the DFE should execute **UDIWait()**.

```
switch( StopReason){
    case UDIStdOutReady:
    case UDIStderrReady:
    case UDIStdInNeeded:
    case UDIStdinModeX:    /* Perform requested I/O operation,
                            * then loop back to calling UDIWait().
*/
    default:               /* Report reason for stopping to user. Do
                            * whatever the user wants. */
}
```

Shutdown:

```
UDIDestroyProcess()       /* Ignore the error code if ProcessID
                           * (returned from CreateProcess) is
                           * UDIProcessProcessor. */
UDIDisconnect()           /* Frees resources & disconnects from TIP
*/
```

Transparent mode Code Examples:

The following show code examples for both an interactive transparent mode session, during which the TIP's prompt is displayed and the user can enter any number of commands, and an immediate transparent mode session, in which a single command is presented to the TIP and the command's output is presented to the user.

```
Interactive_Transparent_Session()

// Note: The TIP may have pending unsolicited transparent mode
output.
//       Thus, the DFE must call GetTrans before calling
UDIPutTrans.
{
  while (1) {
    switch (UDIGetTrans) {
        case UDIErrorNoError :
        display output;
        break;

    case  UDIErrorTransInputNeeded :
        Get a line of input from user;
        Call UDIPutTrans();
        break;

    case UDITransModeX :
        Call UDITransMode;
        Change Input Mode as required;
        break;

    case UDIErrorTransDone :
        Call UDIPutTrans with Count=0;  // part of protocol for
getting prompt
        Call UDIGetTrans() until status = UDIErrorTransDone;  // to
get prompt
        Display prompt to user.
            // DFE is allowed to break out of loop now (via DFE
exit method)
        if (not breaking out of loop) {
            Get a line of input;
            Call UDIPutTrans with new transparent mode command.
        }
             break;

    case UDIErrorTransExit :
        // DFE must break out of loop.

    } /*switch*/
  } /*while*/
}
```

The immediate session is identical to the interactive session loop except for the way that UDIErrorTransDone is handled. As in interactive mode, the TIP may have pending transparent mode output. Since this is immediate mode, the user may only expect to see the output from the immediate mode command, but the DFE could mark this "pending" output, if any, to show that it is not the output from the immediate mode command.

```
Immediate_Transparent_Session(command)
char *command;     // assumes immediate command is being passed as
a parameter
{
    command_has_been_sent = false;
    while (1) {
            switch (UDIGetTrans) {
                    case UDIErrorNoError:
                            display output;
                            break;

                    case  UDIErrorTransInputNeeded :
                            Get a line of input from user;
                            Call UDIPutTrans();
                            break;

                    case UDITransModeX :
                            Call UDITransMode;
                            Change Input Mode as required;
                            break;

                    case UDIErrorTransDone :
                            if (!command_has_been_sent) {
                                    // ready to send immediate
command.
                                    PutTrans(command);
                                    command_has_been_sent = true;
                            }
                            else {
                                    // command has been sent
&completed.
                                    exit loop;
                             }
                            break;
                    case UDIErrorTransExit :
                            // DFE must break out of loop.
            } /*switch*/
    }*while*/
}
```

Chapter 3

UDI Services

This chapter documents all UDI services. The services are defined in terms of C language function prototypes, known as the UDI Procedural interface (UDIP). These prototypes are independent of the underlying Inter–Process Communication (IPC) mechanisms, which are defined in Chapters 4 and 5. We recommend you use UDIP for creating DFEs and TIPs rather than using the IPC mechanisms directly. By using UDIP, a DFE and TIP can be linked together directly, allowing target–specific versions of debuggers to be created in special situations (in some environments, this makes debugging the DFE or TIP easier). Using UDIP also allows some changes to occur in the underlying IPC mechanisms without causing major problems. Finally, using UDIP provides a host–independent interface to UDI, thereby allowing one set of sources to be used on hosts with different IPC mechanisms.

Return Codes

In the following detailed descriptions of the UDI services, the return codes that are shown at the end of each call are those that are of interest for that call and do not generally include those return codes that could be returned by any UDI call. For more information on the meaning of error codes, see “Appendix B: UDI Error Codes.”

Procedure Names

The procedural interfaces for the following functions changed from UDI 1.3 to UDI 1.4: **UDISetBreakpoint**, **UDIQueryBreakpoint**, and **UDIConnect**. In the descriptions of these functions in this chapter, we present both the old and new procedural interfaces, differentiating them by appending either **_13** or **_14** to the above names. Of course, in the UDI 1.3 procedural interface, these functions were called simply **UDISetBreakpoint**, **UDIQueryBreakpoint**, and **UDIConnect.**

Since the **_14** versions are supersets of the **_13** versions, it is expected that DFEs and TIPs will want to use the new **_14** procedural interfaces. The old **_13** procedural interface is presented for the benefit of DFE and TIP implementors who had used the 1.3 (or earlier) version of the UDI spec. For backwards compatibility with existing 1.3 code, the sample procedural interface implementation from AMD points the unqualified names at the **_13** versions. These unqualified names can be modified to point to the **_14** versions via a compile-time flag.

In general, throughout the rest of this specification, when we use the unqualified names **UDISetBreakpoint**, **UDIQueryBreakpoint**, and **UDIConnect,** we mean either the **_13** or the **_14** version.

Note that the new **_14** procedural interface can be used by a DFE, even if the TIP happens to be an older TIP. It is the responsibility of the IPC code that maps the procedural interfaces to messages to detect the version of the TIP, to map the new procedure to an old message, if possible, and if not, to return an error.

Note also that a 1.4 TIP must define both the **_13** and **_14** versions of these functions. This is because a 1.4 TIP may get connected to by a 1.3 DFE and such a DFE can only send 1.3 messages. However, the sample IPC implementation from AMD provides entry points for the **_13** versions which simply maps the **_13** requests to the superset **_14** functions and the TIP writer using this IPC implementation need only supply the **_14** functions.

Optional versus Required Services

The following list documents which of the UDI services are optional for the TIP and which of the UDI DFE callback services are optional for the DFE. All other services are required. Any TIP or DFE which does not support an optional service should return **UDIErrorUnsupportedService** if the service is invoked. Despite the optional nature of some services, TIP and DFE implementors should strive to provide all of the documented services.

- **UDIGetErrorMsg** is required only if the TIP returns TIP–defined (negative) error codes.
- **UDIGetTargetConfig** is not required.
- **UDICopy** is not required.

- **UDIStep** is required, although only calls with *StepType* equal to **UDIStepNatural** are guaranteed to work.

- **UDISetBreakpoint** is not required.

- **UDIQueryBreakpoint** and **UDIClearBreakpoint** must be supported if **UDISetBreakpoint** is supported.

- **UDIGetStdout** must be implemented if a TIP returns **UDIStdoutReady**.

- **UDIGetStderr**, **UDIPutStdin**, and **UDIStdinMode** behave in a similar manner.

- **UDIGetTrans** is optional. If **UDIGetTrans** is supported, **UDIPutTrans** and **UDITransMode** must be supported. In the case of **UDITransMode** and **UDIStdinMode**, only buffered, echoed, line–oriented input (the default mode) is guaranteed to be supported.

- **UDIFind** support in the TIP is optional. If **UDIFind** is supported, the sub–features of masking, "Stride != PatternSize, PatternCount != 1", reverse searches, and "WhereToLook.Space != <a memory space>" are optional. The error **UDIErrorUnsupportedServiceVariation** can be used to mean, "I have the service, but I can't do that particular variation of it."

- **UDIDFEEvalExpression** and **UDIDFEEvalResource** are optional in the DFE.

- In **UDIDFEGetInput**, only the buffered, echoed, line–oriented input (the default mode) is guaranteed to be supported.

- **UDIEnumerateTIPS**() is optional. Since this service is implemented entirely in the DFE, it is optional by definition.

In general, no UDI services can be called from signal handlers except **UDIStop**. TIPs and the IPC Layer are not required to be reentrant.

Values Reserved for Vendor–Specific Use

Except for those types that have been specifically defined to contain vendor–specific values, a function parameter cannot have a value other than those defined by UDI. Use of a value for any parameter other than those defined here will result in undefined behavior and may render the DFE incompatible with future UDI specifications. A summary of the parameters that have been defined to contain vendor–specific values are:

CPUSpace	Negative values are reserved for vendor–specific definition.
UDIStepType	Negative values are reserved for vendor–specific definition.
UDIBreakType	If the value is negative (i.e., the MSB is set) the definition of all other bits is vendor–specified.

UDI Type Definitions

UDI defines many different data types. This section documents these data types and provides specific values for the Am29000 microprocessor. Other processors will need some modification of values in some fields and some may even require new types.

UDISessionId is a handle that refers to a specific connection. It is returned from a **UDIConnect** call and is used in **UDISetCurrentConnection** and **UDIDisconnect** calls.

```
typedef    UDIInt  UDISessionId;
```

UDIPId is a handle for processes. The special **UDIPId** value, **UDIProcessProcessor**, refers to the raw CPU. See **UDICreateProcess** for more information about when **UDIProcessProcessor** can be used.

```
typedef    UDIInt  UDIPId;
```

UDIError is the type for the error code returned by each UDI service and for the parameter supplied to **UDIGetErrorMessage**.

```
typedef    UDIInt  UDIError;
```

Many UDI services access target resources. All resources are identified via the **UDIResource** structure, including target memory, target registers, and any resources the TIP may provide, such as trace buffers or profiling data. **UDIResource** consists of two members: a Space member, and an Offset member. The Offset member is usually the width of the target CPU's address, although it may be wider. For the 29K Family, *CPUOffset* is an unsigned 32–bit integer. The Space member, *CPUSpace*, is of signed integer type, and is generally a small integer. Negative values of *CPUSpace* are reserved for vendor–specific definition, i.e. a UDI specification cannot define a negative *CPUSpace* value.

Each CPU–specific implementation provides its own values for *CPUSpace* to access the various CPU–specific resources available. Note that CPU–specific generally means CPU–family specific. All processors within a family that use the same size addresses share space values and meanings.

Refer to Appendix C for 29K Family resource definitions.

The structure associated with a UDI resource is:

```
typedef struct
{
 CPUSpace  Space;
 CPUOffset Offset;
 } UDIResource;
```

Some services return or receive a range of resources sharing a common space. These services have defined **UDIMemoryRange** types.

```
typedef struct {
    CPUSpace       Space;
    CPUOffset      Offset;
    CPUSizeT       Size;
 } UDIMemoryRange;
```

When stepping the target, several options are available. You can step individual instructions or step over certain kinds of instructions (calls and traps). You can request stepping until the program is outside a specified range of instruction addresses. These various step types (shown below) can be ORed together. Each TIP offers a default step mode, and DFEs are encouraged to use **UDIStepNatural** to refer to the TIP when the user has not specifically requested a type of step. Some TIPs, such as emulators and simulators, naturally step into traps, while other TIPs do not. If you request a step type that the TIP cannot support, you get a **UDIErrorUnsupportedStepType** message. If the value of a UDIStepType parameter is negative (i.e., the MSB is set), the definition of all other bits is vendor-specified.

```
typedef    UDIInt  UDIStepType;
#define    UDIStepOverTraps      0x0001
#define    UDIStepOverCalls      0x0002
#define    UDIStepInRange        0x0004
#define    UDIStepNatural        0x0000
typedef    struct
{
    CPUOffset      Low;
    CPUOffset      High;
 } UDIRange;
```

The function UDIWait returns the current status of the target in a parameter of type UDIStopReason. It is defined as follows:

```
typedef UDIUInt32  UDIStopReason;
#define UDIGrossState     0xff
#define UDITrapped        0       /* Fine state - which trap */
#define UDINotExecuting   1
#define UDIRunning        2
#define UDIStopped        3
#define UDIWarned         4
#define UDIStepped        5
#define UDIWaiting        6
#define UDIHalted         7
#define UDIStdoutReady    8       /* fine state - size */
#define UDIStderrReady    9       /* fine state - size */
#define UDIStdinNeeded    10      /* fine state - size */
#define UDIStdinModeX     11      /* fine state - mode */
#define UDIBreak          12      /* Fine state - Breakpoint Id */
#define UDIExited         13      /* Fine state - exit code */
#define UDINotResponding        14
#define UDIOutOfControl         15
#define UDIReset                16
#define UDINoPower              17
#define UDINoClock              18
```

UDIBreakInfo is a structure that describes the characteristics of a breakpoint. It is an input for **UDISetBreakpoint** and an output for **UDIQueryBreakpoint**. It is defined as:

```
 struct    UDIBreakInfo {
    UDIBreakType_14 Type;  /* break type */
    UDIMemoryRange         Region; /* breakpoint region: start addr
& size */
    UDIInt32       PassCount;     /* reload pass count */
    UDIInt32       CntRemaining;  /* effective pass count */
    UDIUInt32      BufLen;        /* Length of the Buf field
                                    * used on vendor-specific bkpts
*/
    char           Buf[1];       /* Variable length Buffer for
                                     vendor-specific breakpoints */
};
```

Note that the *Buf* field is used only for vendor-specific breakpoints and its actual length is reflected in *Buflen*.

UDIBreakType_14 specifies the type of breakpoint to be applied in a **UDISetBreakpoint_14** call. If the value of a **UDIBreakType_14** parameter is negative (i.e., the MSB is set), the definition of all other bits is vendor–specified. The various BreakFlags shown below can be OR'ed together to make compound breaktypes.

```
typedef    UDIInt32       UDIBreakType_14;

#define    UDIBreakFlagExecute           0x0001
#define    UDIBreakFlagRead              0x0002
#define    UDIBreakFlagWrite             0x0004
#define    UDIBreakFlagFetch             0x0008
#define    UDIBreakFlagWidthByte         0x0010
#define    UDIBreakFlagWidthHalfWord     0x0020
#define    UDIBreakFlagWidthWord         0x0040
#define    UDIBreakFlagGenSyncPulse      0x0080
#define    UDIBreakFlagDoNotStopProcessor       0x0100
```

The semantics of these BreakType flags as well as the semantics of the other fields of BreakInfo are described under the **UDISetBreakpoint_14** call.

Note: the older call **UDISetBreakpoint_13** used a subset of these breaktype bits passed in a **UDIBreakType_13** type defined as follows:

```
typedef    UDIInt  UDIBreakType_13;

#define    UDIBreakFlagExecute           0x0001
#define    UDIBreakFlagRead              0x0002
#define    UDIBreakFlagWrite             0x0004
#define    UDIBreakFlagFetch             0x0008
```

The following are the legal return values for the *KindOfAnswer* parameter returned by **UDIDFEEvalExpression**:

```
#define    UDIAnswerKindNone             0
#define    UDIAnswerKindValue            1
#define    UDIAnswerKindResource         2
```

The *IOType* parameter is used with **UDIDFEPutOutput** and **UDIDFEGetInput** to specify whether the source of the I/O request (TIP or target) and, for output, whether the destination is standard output or standard error:

```
typedef    UDIUInt UDIIOType
#define    UDIIOTypeTIPStdout     0
#define    UDIIOTypeTIPStderr     1
#define    UDIIOTypeTIPStdin      2
#define    UDIIOTypeTargetStdout  3
#define    UDIIOTypeTargetStderr  4
#define    UDIIOTypeTargetStdin   5
```

UDIDFEEvalExpression can return a *Type* parameter (of type **UDIExprType**) if the expression evaluates to a value. This parameter is a simplified indicator of the C language type of the returned value. The following section from the **udiproc.h** file defines the possible return types.

```
typedef    UDIInt UDIExprType;
#define    UDITypeUnknown 0      /* type not applicable, or DFE
                                  * doesn't support types */
#define    UDITypeOther   1      /* type is known but does not
                                  * match one in the UDI list */
#define    UDITypeChar    2      /* an 8-bit ASCII character */
#define    UDITypeInt     3      /* an integral type with
                                  * Size, handles short, int,
                                  * long */
#define    UDITypeFloat   4      /* with Size, handles float,
                                  * double, long double */
```

The end–of–line (EOL) character for all strings used in Programmed I/O Services, Transparent Mode Services, and DFE Access to TIP I/O Services is LF (0ah). It is the responsibility of the DFE (or its libraries) to do any host mapping of this EOL to the host–specific line termination.

Table 2. UDI Services in Alphabetical Order

Name	Description	Page
UDICapabilities	Allows TIPs and DFEs to be aware of the other's capabilities	3-11
UDIClearBreakpoint	Clears identified breakpoint	3-13
UDIConnect	Connects a TIP to a DFE and confirms communication	3-14
UDICopy	Duplicates a block of objects at the TIP	3-17
UDICreateProcess	Creates a process context	3-18
UDIDestroyProcess	Informs the TIP that no further debugging will occur	3-19
UDIDisconnect	Called when a DFE is finished with a connection	3-20
UDIEnumerateTIPs	Informs DFE of available TIPs	3-21
UDIExecute	Causes execution to continue from the current program counter location	3-22
UDIFind	Finds one or more occurrences of a specified pattern	3-23
UDIGetErrorMessage	Retrieves the text associated with a TIP–specific error	3-24
UDIGetStderr	Called when TIP needs to send output to the user via stderr	3-25
UDIGetStdout	Called when TIP needs to send output to the user via stdout	3-26
UDIGetTargetConfig	Provides configuration information about target	3-27
UDIGetTrans	Is the heart of transparent mode operation for DFEs	3-28
UDIInitializeProcess	Initializes or reinitializes a process	3-30
UDIPutStdin	Obtains input from the user	3-32
UDIPutTrans	Called during transparent mode to send characters from the DFE to the TIP	3-33

UDICapabilities

Call

```
UDIError UDICapabilities (
    UDIUInt32      *TIPId,              /* Out */
    UDIUInt32      *TargetId,           /* Out */
    UDIUInt32      DFEId,               /* In */
    UDIUInt32      DFE,                 /* In */
    UDIUInt32      *TIP,                /* Out */
    UDIUInt32      *DFEIPCId,           /* Out */
    UDIUInt32 *    TIPIPCId,            /* Out */
    char *         TIPString,           /* Out */
    UDISizeT       BufSize,             /* In */
    UDISizeT *     CountDone            /* Out */
    );
```

Description

UDICapabilities allows TIPs and DFEs to be aware of one another's capabilities. As of UDI 1.3, the DFE is required to call this function immediately after **UDIConnect**. A DFE that does not call **UDICapabilities** immediately after **UDIConnect** can be assumed to be a UDI 1.2 DFE.

The TIP, Target, DFE, DFEIPC, and TIPIPC Id fields are divided into three sections:

- The 16 most–significant bits are a company code, assigned by AMD.
- The next 4 bits are a product code, assigned by the company.
- The least–significant 12 bits are a version number assigned by the company. These twelve bits are broken into three 4-bit fields comprising a major version number, minor version number and point version number.

For common display purposes, the version number should be displayed as three four–bit values. Any component not present has a 0 Id. The TIP (always present) zeroes the IPC Id fields, and the IPC Layer fills them in after the TIP has finished.

- As a special case, bit 31 of the DFEIPCId and TIPIPCId parameters are used to indicate the endianness of the DFE or TIP. Bit 31 set indicates a little-endian DFE or TIP, Bit 31 clear indicates a big-endian DFE or TIP. At the procedural interface, a DFE or TIP usually need only be concerned with the endianness of the other side for vendor-specific extensions where non-byte quantities are being sent.

The *DFE* parameter indicates to the TIP what version of UDI the DFE prefers, i.e., the latest version number the DFE understands. The TIP examines the DFE's UDI version and determines whether it can support that version. If the TIP is capable of supporting the DFE at the DFE's preferred UDI level, the TIP returns that level in the TIP parameter. If the TIP's UDI version is lower than the DFE's, the TIP returns the TIP's version in the TIP parameter. If the TIP's UDI version is higher than the DFE's, but the TIP cannot support obsolete features of the DFE's version, the TIP returns 0 in the *TIP* parameter. Upon returning, the DFE should examine the TIP's UDI version. If the TIP's UDI version is the same as the DFE's, communication can take place without problems. If the TIP's UDI version is 0, the DFE should immediately disconnect because further communication is unlikely. If the TIP's UDI version is other than 0 or the DFE's version, the DFE can communicate with the TIP at the TIP's level.

The *TIPString* parameter is a buffer of *BufSize* characters that the TIP fills with information that may further modify the TIP identification information, setting *CountDone* to the number of bytes (including the null terminator) returned. A typical example of the kind of information returned in TIPString would be a human-readable listing of the TIP and/or target's configuration. If *CountDone* is equal to *BufSize*, the DFE should continue calling **UDICapabilities** to get the rest of the *TIPString*. It is recommended that DFEs provide a *TIPString* buffer of at least 1Kbyte to avoid multiple calls to the TIP. The TIP is allowed to have embedded end–of–line characters in the returned *TIPString*. It is recommended that DFEs display this information on a line by itself immediately following a line where the various *Id* fields have been displayed.

UDIClearBreakpoint

Call

```
UDIError UDIClearBreakpoint (
    UDIBreakId     BreakId              /* In */
    );
```

Description

UDIClearBreakpoint clears the identified breakpoint. If no such breakpoint exists, **UDIErrorInvalidBreakId** is returned. If Breakpoint ID = 0, **UDIClearBreakpoint** clears all breakpoints.

Return Codes

UDIErrorInvalidBreakId

UDIConnect

Call

```
UDIError UDIConnect_14 (
    char           *Configuration,       /* In */
    UDISessionId   *Session              /* Out */
    UDIUint32      DFE                   /* In */
    );

 UDIError UDIConnect_13 (
    char           *Configuration,       /* In */
    UDISessionId   *Session              /* Out */
    );
```

Description

UDIConnect connects a TIP to a DFE and confirms that the DFE and the TIP can communicate. No other action (such as initializing the target) is performed. If the TIP is not active at the time of the connection, the UDI DFE IPC Layer starts the TIP.

The DFE parameter has the same semantics as the DFE parameter in the UDICapabilities call; it indicates to the TIP what version of UDI the DFE prefers, i.e., the latest version number the DFE understands. Adding the parameter to the **UDIConnect_14** call allows the TIP to decide whether it can use some of the newer callback services during **UDIConnect_14**.

If the DFE parameter indicates a version that is older than what the TIP wishes to support, the TIP can either "step down" to that level or else can refuse to accept the connection. If the DFE parameter is newer than the TIP's the TIP must still accept the connection, because the DFE may be willing to "step down" to the TIP's level, which negotiation can't happen until the UDICapabilities exchange.

UDIConnect_13 is the older version of this call. It is identical to **UDIConnect_14** except that the DFE parameter is not passed. The rest of this description applies to either version.

The way the *Configuration* parameter is translated for DOS hosts and for UNIX hosts is described in detail in Appendix B. In general, IPC layers use a configuration file where each line contains the configuration name, the TIP program name, and other TIP–specific options. For example, a simulator TIP might have several lines in the file where each line represents different configurations of the simulator that the user has set up and named.

Session points to an object that will be used to identify this DFE–TIP connection in future session management UDI calls.

After a successful **UDIConnect**, the current connection is the one just connected and the current process for that connection is **UDIProcessProcessor**. If an error is returned, the connection was not established. If the error is negative, the DFE can call **UDIGetErrorMsg** and must call **UDIDisconnect**. No other service requests are permitted if an error is returned.

To solve certain problems, the IPC Layer (prior to spawning a new TIP and calling the new TIP's **UDIConnect** function) can call **UDIConnect** at each currently running TIP that has the same executable file name as the desired configuration. Because TIPs may need exclusive access to either TIP or target resources, TIPs may need to either prevent or force the IPC Layer to spawn a new TIP.

One such example is a TIP that communicates with a private target. The TIP should ensure that no other TIPs are spawned that might contend for the same target. Such TIPs, when called at **UDIConnect** for a configuration that would create the contention, should return **UDIErrorConnectionUnavailable**. The IPC Layer will recognize this error and return to the DFE immediately, rather than attempting to find or spawn another TIP to satisfy the request.

In another example, a TIP that can support multiple connections by spawning multiple TIPs, like a simulator, needs to return only **UDIErrorTryAnotherTIP**. This causes the IPC Layer to check other installed TIPs and, if necessary, attempt to spawn another instance of the same TIP.

Return Codes

```
UDIErrorTryAnotherTIP
UDIErrorNoSuchConfiguration
UDIErrorCantConnect
UDIErrorCantOpenConfigFile
UDIErrorCantStartTIP
UDIErrorConnectionUnavailable
UDIErrorTryAnotherTIP
UDIErrorExecutableNotTIP
UDIErrorInvalidTIPOption
```

UDICopy

Call

```
UDIError UDICopy (
    UDIResource          From,              /* In */
    UDIResource          To,                /* In */
    UDICount             Count,             /* In */
    UDISizeT             Size,              /* In */
    UDICount *           CountDone,         /* Out */
    UDIBool              Direction          /* In */
    );
```

Description

UDICopy duplicates a block of objects at the TIP. The *Direction* parameter indicates whether the copy occurs from lower–to–higher addresses or vice versa. The parameter is used regardless of whether the copy involves overlapping objects. A non–zero value indicates a lower–to–higher copy, while a 0 value specifies a higher–to–lower copy.

UDICopy can be used to fill an area with a pattern by writing the pattern once at the base of the fill area and then calling copy, specifying a lower–to–higher copy with the source as the base of the fill area and the destination as the address of the second copy. The TIP must guarantee that the copy is performed as if it were done one object at a time. The actual transfer width used is TIP–specific.

Unlike **UDIRead** and **UDIWrite**, there are no restrictions on *Size*. Like **UDIRead** and **UDIWrite**, if the count of objects requested cannot be performed, then *CountDone* indicates the number of objects completely copied and **UDICopy** returns **UDIErrorIncomplete**. The result of *To* pointing into the middle of the first *From* object is undefined. Not all memory spaces support all object sizes. An attempt to access a resource with an unsupported object size returns **UDIErrorInvalidSize**.

Return Codes

```
UDIErrorUnknownResourceSpace
UDIErrorInvalidResource
UDIErrorResourceNotWriteable
```

UDICreateProcess

Call

```
UDIError UDICreateProcess (
    UDIPId          *PId            /* Out */
    );
```

Description

UDICreateProcess creates a process context. See the Process Management section on page 2-2 for details on simple process management strategies. Note that each connection manages processes separately. This means that two sessions may return the same *PId* value. It also means that changing connections (using **UDISetCurrentConnection**) changes the "current" process, because each connection maintains its own current process state. After a successful **UDICreateProcess** call, the newly created process is the current process for that connection.

Return Codes

`UDIErrorCantCreateProcess`

UDIDestroyProcess

Call

```
UDIError UDIDestroyProcess (
    UDIPId         PId            /* In */
    );
```

Description

UDIDestroyProcess informs the TIP that no further debugging will occur against the indicated process. DFEs should call this service for all processes (except **UDIProcessProcessor**) before shutting down. If the *PId* passed to **UDIDestroyProcess** is the current *PId* for the connection, then **UDIProcessProcessor** becomes the current process.

Return Codes

`UDIErrorNoSuchProcess`

UDIDisconnect

Call

```
UDIError UDIDisconnect (
    UDISessionId          Session,              /* In */
    UDIBool               Terminate             /* In */
    );
```

Description

When a DFE is finished with a connection, it must call **UDIDisconnect**. There are two types of disconnection:

- The primary disconnection (*Terminate* = **UDITerminateSession**) terminates the TIP, possibly causing any target action to be lost.
- The second type of disconnection (*Terminate* = **UDIContinueSession**) allows subsequent reconnection (via **UDIConnect**).

A user might want to reconnect in order to switch DFEs at a certain point in a program, for example. Or, if the host cannot support a DFE resident with other applications that the user wants to run, the user may want to leave a target executing while running the other application.

The terminate type of **UDIDisconnect** allows a graceful shutdown of the connection, and, on some hosts, returns critical resources used by the IPC Layer.

The non–termination form of **UDIDisconnect** also permits a graceful shutdown of the connection, although in some cases the TIP may not be able to return some critical resources because they are needed to maintain execution of the target.

In the event that **UDIDisconnect** returns an error, the connection is still established (unless the error is **UDIErrorNoSuchConnection**, in which case the connection was never established).

Return Codes

`UDIErrorCantDisconnect`
`UDIErrorNoSuchConnection`

UDIEnumerateTIPs

Call

```
UDIError UDIEnumerateTIPs (
    UDIInt         (*UDIETCallback) (char *Configuration )
                   /* In                 In to callback */
    );
```

Description

If a DFE wants to know what TIPs are available, it should call **UDIEnumerateTIPs**. For each TIP configuration on the system, **UDIEnumerateTIPs** calls the callback function identified by the *UDIETCallback* parameter. That function receives a single parameter that is a pointer to a string containing the name of the configuration. If the DFE wants to preserve the name, it must copy it from the indicated area into a DFE–managed space. This is because **UDIEnumerateTIPs** can be implemented with only one buffer into which it points for each configuration.

In all circumstances, the callback function must return to **UDIEnumerateTIPs**. If the callback function does not want to be called for additional configurations, it can return the value **UDITerminateEnumeration**. Otherwise, it should return **UDIContinueEnumeration**.

Note: **UDIEnumerateTips** is not a service provided by the TIP, it is an optional service provided by the procedural interface and implemented entirely in the DFE side.

UDIExecute

Call

```
UDIError UDIExecute (
    void
    );
```

Description

UDIExecute causes execution to continue from the current program counter location. Progress is guaranteed by the TIP. The implementation of the guarantee is as if one single step occurs, then all breakpoints are installed, followed by unbounded execution. The single step that occurs is of the StepNatural variety (see **UDIStep**). This may occasionally cause execution to stop at the breakpoint again even though the instruction at the breakpoint has not yet been executed. This occurs on emulators when interrupts are pending. The natural step that takes place before breakpoints are installed does not step the current PC, but rather steps the first instruction of the interrupt handler.

UDIExecute returns as soon as execution has begun on the target, usually just after breakpoints are installed and the process is let go. For some TIPs on some hosts, all execution occurs when the DFE calls **UDIWait**. Consequently, a DFE may not assume that any progress has been made by the sequence: **UDIExecute**, long delay, **UDIStop** (and a DFE must call **UDIWait** to ensure progress).

Calling **UDIExecute** when the target is already running will cause the error **UDIErrorTargetAlreadyRunning**.

Return Codes

`UDIErrorTargetAlreadyRunning`

UDIFind

Call

```
UDIError UDIFind (
    UDIMemoryRange        WhereToLook,          /* In */
    UDIInt32              Stride,               /* In */
    UDIHostMemPtr         Pattern,              /* In */
    UDIHostMemPtr         PatternMask,          /* In */
    UDICount              PatternCount,         /* In */
    UDISizeT              PatternSize,          /* In */
    UDIBool               PatternHostEndian,    /* In */
    UDICount              MaxToFind,            /* In */
    UDICount *            CountFound,           /* Out */
    CPUOffset             FoundAtOffset[]       /* Out */
    UDIHostMemPtr         FoundValues[]         /* Out */
    );
```

Description

UDIFind is used to find one or more occurrences of a specified pattern in a specified *UDIMemoryRange*. The DFE specifies the memory range to search, a *Pattern*, and the maximum occurrences to report. The pattern consists of *PatternCount* objects, each of size *PatternSize* and an endian type of *PatternHostEndian*. (The pattern may be further qualified by a *PatternMask*, such that a match is found only if, for each object in the pattern: *TargetMem* & *PatternMask* == *Pattern* & *PatternMask*). A *PatternMask* of NULL indicates no masking, i.e., it is equivalent to a *PatternMask* whose bits are all ones.

The *Stride* parameter specifies how much to move the target offset pointer for each step of the search (this will usually be the same as *PatternSize*). When *Stride* is negative, a reverse search over the *WhereToLook* range is specified. The TIP always reports the number of matches found and the offsets at which they were found. If the *PatternMask* is not null, the TIP also reports the actual data that was found at the offsets. Using the *FoundAtOffset* and *FoundValues* arrays, the DFE is responsible for ensuring that the *FoundAtOffset* and *FoundValues* buffers are large enough to hold *MaxToFind* answers. The data returned in *FoundValues* if of endian type *PatternHostEndian*.

Return Codes

`UDIErrorUnknownResourceSpace`
`UDIErrorInvalidResource`

UDIGetErrorMessage

Call

```
UDIError UDIGetErrorMsg (
    UDIError              ErrorCode,            /* In */
    UDISizeT              MsgSize,              /* In */
    char *                Msg,                  /* Out */
    UDISizeT *            CountDone             /* Out */
    );
```

Description

UDIGetErrorMessage retrieves the text associated with a TIP–specific error. *MsgSize* indicates the size of the buffer the DFE has allocated to hold the error message. The DFE should keep calling **UDIGetErrorMessage** until *CountDone* is less than *MsgSize*.

UDIGetErrorMessage returns **UDIErrorUnknownError** if *ErrorCode* is not a TIP–specific error code.

Multiple calls to **UDIGetErrorMessage** with the same error code will not necessarily return the same string. Thus, negative error code strings should not be cached by the DFE.

UDIGetStderr

Call

```
UDIError UDIGetStderr (
    UDIHostMemPtr          Buf,                /* Out */
    UDISizeT               BufSize,            /* In */
    UDISizeT *             CountDone           /* Out */
    );
```

Description

UDIGetStderr is analogous to **UDIGetStdout** (on page 3-26) except that data comes from the TIP's standard error channel and **UDIWait** returns **UDIStderrReady**.

UDIGetStdout

Call

```
UDIError UDIGetStdout (
    UDIHostMemPtr          Buf,                 /* Out */
    UDISizeT               BufSize,             /* In */
    UDISizeT *             CountDone            /* Out */
    );
```

Description

When the TIP needs to send output to the user via the conventional standard output means, **UDIWait** will have returned a *StopReason* of **UDIStdoutReady** with a fine state of the number of characters available. The DFE should then call **UDIGetStdout** to acquire the data and then display it for the user by whatever means the DFE uses for program I/O.

The DFE should pass a *BufSize* as large as practical even though it knows how many characters were available when **UDIWait** returned. The program may have added characters to the standard output stream since then and **UDIGetStdout** can acquire those as well. In any case, the DFE should call **UDIGetStdout** until the returned *CountDone* is less than *BufSize*. The characters should be made immediately available to the user, although deferral until certain other conditions (such as **UDIStdinNeeded** or **UDIExited**) occur should work as well. After all output data have been obtained, the DFE should resume calling **UDIWait**.

In the event the DFE calls **UDIGetStdout** and no data are available from the TIP, it should simply return a *CountDone* of 0 rather than a UDI error.

UDIGetTargetConfig

Call

```
UDIError UDIGetTargetConfig (
    UDIMemoryRange        KnownMemory[],        /* Out */
    UDIInt  *             NumberOfRanges,       /* In/Out */
    UDIUInt32             ChipVersions[],       /* Out */
    UDIInt *              NumberOfChips         /* In/Out */
    );
```

Description

Call **UDIGetTargetConfig** to determine matters such as the size of various memory spaces and the type of CPU–related chips installed. The DFE passes the address of an array of *UDIMemoryRange* structures (using C semantics, a pointer to a single *UDIMemoryRange* structure). The size of the array is passed by the DFE at *NumberOfRanges*. The TIP fills in as many of the structures as necessary, returning the number filled in at *NumberOfRanges*. If the function returns **UDIErrorIncomplete**, the TIP tries to return more ranges than available space allows, and the DFE is encouraged to call **UDIGetTargetConfig** again with a larger array to retrieve all of the data.

The same array/count technique is used for various chips that may be encountered in the system. For each target CPU family type, specific elements of the array are defined to correspond to specific chips that may be present in the system. For the 29K Family, element 0 is the CPU PRL (the entire configuration register, since future members may expand the configuration register towards the LSB) and element 1 is the coprocessor (Am29027—PRL fetched from precision register). A chip not present in the system should have its element filled in with the CPU–family defined value **UDI*xxx*CPUChipNotPresent**. (*xxx* is CPU–family dependent. For the 29K Family, for example, the constant's name is **UDI29KChipNotPresent**.)

UDIGetTrans

Call

```
UDIError UDIGetTrans (
    UDIHostMemPtr          Buf,                 /* Out */
    UDISizeT               BufSize,             /* In */
    UDISizeT *             CountDone            /* Out */
    );
```

Description

UDIGetTrans is the heart of transparent mode operation for DFEs. DFEs generally call **UDIGetTrans** until **UDIGetTrans** requests (via a return value) that one of the other transparent mode functions be called. **UDIGetTrans** returns **UDINoError** when it has transparent mode data to give to the user. The amount of data is indicated by the *CountDone* return argument.

The TIP also uses special returns from **UDIGetTrans** to indicate:

- when the TIP requires input but is not at the prompt (**UDIErrorTransInputNeeded**)
- when the TIP is returning the prompt (**UDIErrorTransPrompt**). A TIP is not required to have a prompt but if it has one it may only be returned with **UDIErrorTransPrompt**.
- when the TIP is at the prompt waiting for input (**UDIErrorTransDone**)
- when the TIP has parsed the transparent mode "exit" command. (**UDIErrorTransExit**).
- when the TIP requires the DFE to change input mode (**UDIErrorTransModeX**).

TIPs that do not support transparent mode should return **UDIErrorUnsupportedService**. Note that TIPs that have a small transparent command set that does not produce any output must still implement **UDIGetTrans** if they implement **UDIPutTrans**. Such a TIP would always return **UDIErrorTransDone** when **UDIGetTrans** is called.

Return Codes

```
UDIErrorTransDone
UDIErrorTransInputNeeded
UDIErrorTransModeX
UDIErrorTransExit
UDIErrorTransPrompt
```

UDIInitializeProcess

Call

```
UDIError UDIInitializeProcess (
    UDIMemoryRange        ProcessMemory[],          /* In */
    UDIInt                NumberOfRanges,           /* In */
    UDIResource           EntryPoint,               /* In */
    CPUSizeT              StackSizes[],             /* In */
    UDIInt                NumberOfStacks,           /* In */
    char *                ArgString                 /* In */
    );
```

Description

UDIInitializeProcess initializes or reinitializes a process. If **UDIInitializeProcess** is performed against process **UDIProcessProcessor**, a target system reset is performed, if permitted. Against other processes, **UDIInitializeProcess** performs whatever steps are necessary to get the process to the stage where the next instruction to be executed is the entry point. This service should also initialize any files or other process–specific operating system state. After a **UDIInitializeProcess** on a 29K Family target, the state of the CPS and CFG registers is TIP–specific.

This call also identifies to the TIP the range of addresses used by the program image(s). Execution of the program may result in additional memory being allocated to the process. Since some processors (notably, the 29K) have more than one stack, we provide as many stacks as the processor family may need. Each processor family defines which element of the array of stack sizes corresponds to which stack. For the 29K, stack 0 is the register stack and stack 1 is the memory stack. Zero is used for any stack size where the default values should be used (default values are TIP–defined). The argument string is passed and can be parsed according to TIP operating system rules. If the DFE wishes to pass embedded whitespace in arguments, those arguments should be enclosed in quotes. If no arguments are supplied, a NULL can be passed.

When **UDIIntializeProcess** is called (including the case when it is performed against UDIProcessProcessor) all breakpoints remain enabled. If the target system is reset manually via a reset button, breakpoints must also persist although it is permitted that breakpoint passcounts be reset to their initial values.

After **UDIIntializeProcess** is called, DFEs that debug programs rather than the raw machine can now inspect the PC (using UDI PC space). If the PC is not at the entry point of the program, the DFE sets a breakpoint at the entry point and executes up to it. Otherwise, stepping the TIP may result in executing one or more instructions that the DFE did not download, and this may confuse the DFE.

Return Codes

```
UDIErrorNoSuchProcess
UDIErrorUnknownResourceSpace
UDIErrorInvalidResource
```

UDIPutStdin

Call

```
UDIError UDIPutStdin (
    UDIHostMemPtr          Buf,                 /* In */
    UDISizeT               Count,               /* In */
    UDISizeT *             CountDone            /* Out */
    );
```

Description

UDIPutStdin obtains input from the user. This can occur in either of two ways:

- If the TIP wants user input, it returns **UDIStdinNeeded** from **UDIWait**, and the number of characters it is prepared to receive.

OR

- The user supplies data for the program's I/O.

Note that even though the **UDIWait** returns **UDIStdinNeeded**, it does not necessarily imply that the process is no longer running. If the TIP's operating system supports asynchronous I/O, it may still be running.

The result of data sent to a TIP that does not support buffering of data for user programs is TIP–defined. If the TIP cannot accept the data, it should return **UDIErrorCantAccept**. The TIP is required to accept at least the number of characters requested by the *StopReason* from **UDIWait** if the most recent **UDIWait** call has returned **UDIStdinNeeded**. After providing the TIP with input data in response to a **UDIWait** *StopReason* of **UDIStdinNeeded**, the DFE should resume calling **UDIWait**.

Return Codes

`UDIErrorCantAccept`

UDIPutTrans

Call

```
UDIError UDIPutTrans (
    UDIHostMemPtr          Buf,                 /* In */
    UDISizeT               Count,               /* In */
    UDISizeT *             CountDone            /* Out */
    );
```

Description

The DFE can support transparent mode access to the TIP. If so, when the DFE wants to send characters to the TIP, it sends them via UDIPutTrans.

When **UDIGetTrans** returns **UDIErrorTransInputNeeded**, the DFE must send transparent mode input to the TIP. When **UDIGetTrans** returns **UDIErrorTransDone** (indicating it is at a logical break between commands), the DFE may present transparent mode input or may exit transparent mode. For any other returns from **UDIGetTrans**, the DFE may not call **UDIPutTrans**.

If **UDIPutTrans** is called with Count=0, this is a request for the TIP to return the prompt on the next **UDIGetTrans** call. If the TIP has a prompt, it returns **UDIErrorTransPrompt** to **UDIPutTrans** and then returns the actual prompt on the next **UDIGetTrans** call (again with the return **UDIErrorTransPrompt**). If the TIP has no prompt, it returns **UDIErrorTransDone** to the **UDIPutTrans** (Count=0) call. If the TIP is not in a state where a prompt is possible, i.e., if the TIP is not between commands, etc., then it returns **UDIErrorCantAccept** to the **UDIPutTrans** (Count=0) call.

TIPs that do not support transparent mode should return **UDIErrorUnsupportedService**.

Return Codes

`UDIErrorCantAccept`
`UDIErrorTransPrompt`

UDIQueryBreakpoint

Call

```
UDIError UDIQueryBreakpoint_14 (
    UDIBreakId            BreakId,             /* In */
    UDISizeT              BufSize,             /* In */
    UDIBreakInfo *        BreakInfo,           /* Out */
    UDIBreakId *          NextBreakId          /* Out */
    );

 UDIError UDIQueryBreakpoint_13 (
  UDIBreakId       BreakId,              /* In */
  UDIResource      *Addr,                /* Out */
  UDIInt32         *PassCount,           /* Out */
  UDIBreakType_13  *Type,                /* Out */
  UDIInt32         *CurrentCount         /* Out */
  ));
```

Description

UDIQueryBreakpoint_14 allows a DFE to obtain the state of breakpoints on the TIP. The DFE can do so even though it does not know which *BreakId* values are valid. This is useful when a DFE connects to an unterminated TIP. (See **UDIDisconnect** on page 3-20.)

Since the size of a returned breakinfo structure can vary (because of the variable length vendor-specific Buf field), the DFE uses the parameter BufSize to specify the maximum size of the BreakInfo.Buf that it is using to receive the BreakInfo structure. If this is not big enough for the breakpoint that is being queried, the TIP returns an error, UDIErrorIncomplete, and fills in BreakInfo.BufLen as to what the actual required BufLen is. The DFE can then requery with a larger BreakInfo.Buf if it desires.

To query all breakpoints, the DFE can start at BreakId 1 and continue until UDIQueryBreakpoint returns UDIErrorNoMoreBreakIds. If the DFE queries a BreakId values not currently in use, but with higher BreakId values in use, the TIP will instead return UDIErrorInvalidBreakId. In all cases, the TIP returns the next valid BreakId in NextBreakId and the DFE should use this for its next query. In this way, the DFE can acquire all of the breakpoints installed on the TIP without keeping any state locally.

Note that the CntRemaining field of the returned BreakInfo structure shows the number of passes remaining until the breakpoint triggers. That is, it is a downcounter which starts from the Passcount parameter passed in by UDISetBreakpoint.

UDIQueryBreakpoint_13 is the old version of this call.

Return Codes

```
UDIErrorInvalidBreakId
UDIErrorNoMoreBreakIds
UDIErrorIncomplete
```

UDIRead

Call

```
UDIError UDIRead (
    UDIResource          From,              /* In */
    UDIHostMemPtr        To,                /* Out */
    UDICount             Count,             /* In */
    UDISizeT             Size,              /* In */
    UDICount *           CountDone,         /* Out */
    UDIBool              HostEndian         /* In */
    );
```

Description

UDIRead transfers objects from the TIP to the DFE. The DFE specifies the source resource address (*From*) and its base object size (*Size*). *Count* objects are moved from the source to DFE memory at *To*. If the resource is not as large as the requested transfer requires, *CountDone* indicates how many objects were completely moved and **UDIErrorIncomplete** is returned.

If the DFE wants the TIP to ensure that the data is in host byte order, it should set *HostEndian* to a non–zero value. A 0 *HostEndian* argument causes the TIP to return the data in target order (as opposed to non–host order). It is assumed that both the DFE and the TIP know the target order. Target order will be either big endian (MSB first for all size objects) or little endian (LSB first for all size objects). The only valid object sizes at this time are 1, 2, 4, 8, 10, and 16. Not all memory spaces support all object sizes. An attempt to access a resource with an unsupported object size returns **UDIErrorInvalidSize**.

Return Codes

`UDIErrorUnknownResourceSpace`
`UDIErrorInvalidResource`

UDISetBreakpoint

Call

```
UDIError UDISetBreakpoint_14 (
    UDIBreakInfo          *BreakInfo,          /* In */
    UDIBreakId            *BreakId             /* Out */
    );

 UDIError UDISetBreakpoint_13 (
  UDIResource             Addr,                /* In */
  UDIInt32                PassCount,           /* In */
  UDIBreakType_13         Type,                /* In */
  UDIBreakId              *BreakId             /* Out */
  );
```

Description

UDISetBreakpoint_14 sets a breakpoint in the current process. The characteristics of the breakpoint are defined in the BreakInfo structure.

The **UDIBreakType_14** Type field of the BreakInfo structure can contain one or more of the following flags:

The flags UDIBreakFlagExecute, UDIBreakFlagRead, UDIBreakFlagWrite and UDIBreakFlagFetch qualify the type of access to break on.

UDIBreakFlagExecute	Is the ordinary execution breakpoint.
UDIBreakFlagRead	Indicates that the process should break when the given address is read.
UDIBreakFlagWrite	Indicates that the process should break when the given address is written.
UDIBreakFlagFetch	Is used when fetching an address should cause the process to break. Fetching is distinguished from reading based on whether the CPU is seeking an instruction or a datum.

The flags **UDIBreakFlagWidthByte**, **UDIBreakFlagWidthHalfword** and **UDIBreakFlagWidthWord** qualify the width of an access (These are usually used with a Read, Write or Fetch access).

The flags **UDIBreakFlagGenSyncPulse** and **UDIBreakFlagDoNotStopProcessor** enable special actions that some processors support. **UDIBreakFlagGenSyncPulse** causes the processor to generate a sync pulse for external triggering when the breakpoint condition is hit. **UDIBreakFlagDoNotStopProcessor** indicates that the processor should not stop execution when this breakpoint is hit. Note that some processors support these two flags only in the combinations of both set or both clear.

In general, any combination of the BreakType flags can be OR'ed together to make a more complex breakpoint although some combinations may not be supported on a particular TIP or particular processor. All TIPs are guaranteed to support UDIBreakFlagExecute against instruction spaces that are writeable. All other breakpoint combinations may be invalid against a TIP. If any part of a breakpoint is invalid, the breakpoint is not set and UDIErrorCantSetBreakpoint is returned.

The other fields of Breakinfo have the following meaning:

The Region field qualifies the address range of the breakpoint. For example, a BreakType of UDIBreakFlagWrite will break when any address in the Region is written.

The CntRemaining field of the BreakInfo structure shows the number of passes remaining until the breakpoint next triggers.

The PassCount field specifies whether the CntRemaining should be reloaded after triggering as well as what value it should be reloaded with. If PassCount is greater than 0, the breakpoint is sticky. Each time the CntRemaining expires, the process stops and the CntRemaining for this breakpoint is reinitialized to the PassCount value. The breakpoint persists until UDIClearBreakpoint is called against the returned BreakId. If PassCount is 0, the breakpoint automatically clears when execution stops (UDIWait returns UDIBreak, UDIStepped, or UDIExited for the current process) after the next UDIStep or UDIExecute call. If PassCount is less than 0, the breakpoint is non-sticky. When CntRemaining has expired, the process stops and the breakpoint is removed. Of course, if execution stops earlier, the breakpoint persists unless UDIClearBreakpoint is called. In no event will a non-sticky breakpoint have its CntRemaining reinitialized.

The CntRemaining field can be set to any value by UDISetBreakpoint. The usual usage is to set it to the absolute value of PassCount. Note that by using the sequence

```
UDIQueryBreakpoint(id, &info);
UDIClearBreakpoint(id);
```

and then some time later

```
UDISetBreakpoint(info, &id);
```

a breakpoint can be disabled and then reenabled without changing its PassCount and current CntRemaining status.

The Buf field, whose length is set in the BufLen field, is not used for the standard breakpoints described in this section but can be optionally used for vendor specific breakpoints.

BreakId values returned by UDISetBreakpoint are required to be non-zero and are generally small positive integers. This makes it easier for pre-UDI1.4 TIPs to query the breakpoints (UDIQueryBreakpoint did not return a NextBreakId field before UDI 1.4).

The result of more than one breakpoint of the same type set against the same address is TIP–defined (that is, the TIP must document what it does). A TIP can limit the number of breakpoints it allows to be set. When any such limit is reached, **UDIErrorTooManyBreakpoints** is returned.

The mechanism used to implement breakpoints should be hidden from the DFE if possible. For example, if an execute breakpoint is implemented by replacing an instruction with a special breakpoint instruction, subsequent calls to **UDIRead** should continue to report the original instruction.

UDISetBreakpoint_13 is the old version of this call. It allowed a subset of the breakpoint types allowed by UDISetBreakpoint_14. See the description of **UDISetBreakType_13** at the beginning of Chapter 3.

Return Codes

```
UDIErrorCantSetBreakpoint
UDIErrorTooManyBreakpoints
UDIErrorUnknownResourceSpace
UDIErrorInvalidResource
```

UDISetCurrentConnection

Call

```
UDIError UDISetCurrentConnection (
    UDISessionId          Session        /* In */
    );
```

Description

DFEs that support multiple concurrent connections use **UDISetCurrentConnection** to switch between the various connected TIPs. DFEs that do not support multiple concurrent connections do not need to call **UDISetCurrentConnection**.

TIPs should be aware that this function may be called apparently without reason and more frequently than expected because of the way that multiple connections may be managed in some IPC implementations.

Return Codes

`UDIErrorNoSuchConnection`

UDISetCurrentProcess

Call

```
UDIError UDISetCurrentProcess (
    UDIPId  PId            /* In */
    );
```

Description

UDISetCurrentProcess is used by DFEs that can handle multiple processes. As pointed out in Chapter 2, even non–multitasking TIPs can support two different processes. DFEs that debug strictly raw machines or strictly programs do not need to be concerned with this call. All TIPs should validate the *PId* argument, even those TIPs that support only one process.

Return Codes

`UDIErrorNoSuchProcess`

UDIStdinMode

Call

```
UDIError UDIStdinMode (
    UDIMode *Mode          /* Out */
    );
```

Description

When the TIP wants to change the method used by the DFE for obtaining input from the user, it causes **UDIStdinMode** to be called by returning **UDIStdinModeX** from **UDIWait**. The default mode for input is line–buffered and echoed with editing. Other modes to be supported are unbuffered (vs. line–buffered), raw (vs. cooked), and non–echoing (vs. echoing). The TIP returns the desired mode in *Mode* after **UDIStdinMode** is called. The TIP also returns the desired mode in the **UDIWait** call as the fine state for the **UDIStdinModeX** gross state. DFEs should endeavor to support all the modes, but currently are required to support only the default mode. After calling **UDIStdinMode**, the DFE should resume calling **UDIWait**.

UDIStep

Call

```
UDIError UDIStep (
    UDIUInt32           Steps,      /* In */
    UDIStepType         StepType,   /* In */
    UDIRange            Range       /* In */
    );
```

Description

UDIStep causes execution to continue one instruction at a time. The number of instructions executed is specified by the *Steps* argument. That number of steps is executed unless a breakpoint is encountered first. Additionally, the *StepType* parameter can modify which instructions count and which do not. Specifically, **UDIStepOverCalls** causes instructions that are executed as a result of a call not to be counted. Similarly, **UDIStepOverTraps** causes instructions in interrupt and trap handlers not to be counted. Not all targets support all step types (monitors, for example, have problems with *not* stepping over traps) and when a requested step type is unavailable, **UDIErrorUnsupportedStepType** is returned.

An additional step type is **UDIStepInRange** which executes until the step count is exhausted, a breakpoint is hit, or the PC is outside *Range*. The last step type is **UDIStepNatural**, which allows the TIP to step naturally. It is the only *StepType* value that is guaranteed to be supported by all TIPs. Note that like **UDIExecute**, progress is guaranteed using the "same step, install breakpoints, more steps" process.

UDIStep returns as soon as the TIP is informed of the number of steps to be executed. Conceptually, no steps are performed until after control returns from **UDIStep**. As a practical matter, on non–multitasking hosts, **UDIStep** does the first step and installs breakpoints before returning.

If **UDIStep** is called when the target is already running, it should cause the target to stop and the next **UDIWait** call should return *StopReason*=**UDIStepped**.

Return Codes

`UDIErrorUnsupportedStepType`
`UDIErrorUnknownResourceSpace`
`UDIErrorInvalidResource`

UDIStop

Call

```
void UDIStop (
    void
    );
```

Description

UDIStop requests that the TIP should return as soon as possible. If the TIP is currently performing a transfer operation, it should be aborted and the TIP should return **UDIErrorAborted** to the DFE. If no transfer is in progress, but transparent mode data transfer is in progress, then the next **UDIGetTrans** call should return **UDIErrorTransDone**, if possible. If transparent mode can not be exited at the time of the **UDIStop** request, the next call to **UDIGetTrans** should return text specifying why transparent mode can not be exited and what can be done to put transparent mode in a state from which it can exit. If no transfer or transparent mode transfer is in progress, but the TIP is executing or stepping, execution should stop and the next **UDIWait** call should return **UDIStopped**. In all other cases, the TIP ignores a **UDIStop** request. It is safe to call **UDIStop** from a signal handler. **UDIStop** is unique in that it does not have a return code.

UDITransMode

Call

```
UDIError UDITransMode (
    UDIMode *Mode          /* Out */
    );
```

Description

If **UDIGetTrans** returns **UDIErrorTransModeX**, the DFE must call **UDITransMode**, and the TIP will return the new mode that it wishes the DFE to use. The DFE must then resume calling **UDIGetTrans**.

In all other respects, this call is identical to **UDIStdinMode** on page 3-42.

UDIWait

Call

```
UDIError UDIWait (
    UDIInt32            MaxTime,            /* In */
    UDIPId *            PId,                /* Out */
    UDIUInt32 *         StopReason          /* Out */
    );
```

Description

UDIWait returns when either the target's state changes or when approximately *MaxTime* milliseconds have elapsed. If *MaxTime* is the special value **UDIWaitForever**, then **UDIWait** returns only when the target's state changes. **UDIWait** returns the state of the target in *StopReason*. When **UDIWait** returns any *StopReason* other than **UDIRunning**, then the process that stopped is returned in *PId* and this process also becomes the "current process" for the connection.

The TIP must return immediately from **UDIWait** (without waiting for the expiration of the *MaxTime* parameter timer) for the following conditions:

- If the gross state is not **UDIRunning**.
- If the *StopReason* (combination of gross state and fine state) has changed since the last **UDIWait** return—with the exception that if the state change has been from (*not* **UDIRunning** + fine state anything) to (**UDIRunning** + fine state 0). The logic here is that the transition from (*not* **UDIRunning**) to **UDIRunning** can only come from a DFE action (for example, **UDIExecute**) and that (**UDIRunning** + fine state 0) would be the next expected state anyway.

If **UDIWait** is called before any calls to **UDIExecute** or **UDIStep** are made, it returns **UDINotExecuting** as a reason..

For some TIPs (for example, a simulator TIP on a DOS host), all actual target execution progress occurs when the DFE calls **UDIWait**. Consequently, a DFE may not assume that any progress has been made by the sequence: **UDIExecute**, long pause, **UDIStop** (and a DFE must call **UDIWait** to ensure progress).

In general, **UDIWait** returns an error code only when it cannot determine the true state of the target (an example might be **UDIErrorTargetNotResponding**). If **UDIWait** can determine the true state of the target, it should instead return **UDINoError** and return the state in *StopReason*. *StopReason* is divided into two parts: the lower 8 bits indicate a gross process state and the upper 24 bits indicate a fine process state. The following table shows the gross *StopReasons* that are defined and their meanings. Some *StopReasons* use the fine state field to return further information and that is defined below as well.

StopReason	Meaning
UDIBreak	Breakpoint was hit. UDIBreak is returned with the upper 24 bits indicating which breakpoint was hit.
UDIExited	Program stopped executing because it exited (or its equivalent). Upper 24 bits give the exit code.
UDIHalted	CPU has halted. Fine state can be 0 or any from the fine state table below. Target will not leave UDIHalted state without DFE intervention.
UDINotExecuting	If **UDIWait** is called before any calls to **UDIExecute** or **UDIStep** are made on a connection, it returns **UDINotExecuting** as a reason.
UDIRunning	CPU is running. Fine state can be 0 or any from the fine state table below.
UDIStdoutReady **UDIStderrReady** **UDIStdinNeeded** **UDIStdinModeX**	TIP needs to perform I/O (usually on behalf of the program). Upper 24 bits indicate how much output is needed, how much input can be accepted, or what mode is desired (whichever is applicable).
UDIStepped	Step count from the current **UDIStep** call expired.
UDIStopped	Target stopped because DFE called UDIStop.
UDITrapped	Invalid or unexpected trap was taken. Upper 24 bits of StopReason indicate which trap was

	taken.
UDIWaiting	CPU is in wait mode.
UDIWarned	CPU was warned.

The two *StopReasons*, **UDIRunning** and **UDIHalted**, are special. They can return a fine state of 0, or if the TIP wants to return additional information, they can use a fine state from the following list of predefined fine states or a TIP–defined negative fine state. Note that in all cases, gross state **UDIHalted** means that the target will remain halted until some action by the DFE (for example, **UDIExecute**) is taken to restart it.

Fine State[2]	Meaning
UDIResetAsserted	Reset is currently asserted.
UDITransientReset	Target went through a reset but is not currently reset.
UDINoClock	Indicates the target has no clock.
Any negative value	Fine state is TIP defined. DFE can call UDIGetErrorMessage, passing the fine state as an argument to get a textual description for the fine state.

The following are some combinations of gross state with these fine states and how they would be interpreted:

UDIRunning + **UDITransientReset**	Target is executing but went through a transient Reset since the last **UDIWait** call.
UDIRunning + **UDIResetAsserted**	Target is currently Reset but may resume execution at any time.
UDIHalted + **UDIResetAsserted**	Target is currenly Reset and will not resume execution until some action is taken by the DFE to restart it. (If reset deasserts, will enter UDIHalted + UDITransientReset state).
UDIHalted + **UDITransientReset**	Target is not currently reset but is halted (until further DFE action is taken) because reset had been asserted earlier and the target is configured such that resets cause it to enter a halted state.

[2] Used with **UDIHalted** or **UDIRunning**

UDIRunning + **UDINoClock**	A static target, for example, may still be "running" even though it has no clock.

Note that when **UDIWait** returns any particular **UDIHalted** *StopReason*, it does not imply whether any other UDI call to the target will succeed or fail. For example, returning (**UDIHalted** + **UDIResetAsserted**) does not imply that an attempted **UDIRead** will fail. If the TIP, for example, cannot satisfy a **UDIRead** because reset is asserted, it returns a **UDIErrorResetAsserted** error on **UDIRead**. If a TIP can, in fact, satisfy the **UDIRead** even when reset is asserted, it would just return **UDINoError** on **UDIRead**.

UDIWrite

Call

```
UDIError UDIWrite (
    UDIHostMemPtr          From,          /* In */
    UDIResource            To,            /* In */
    UDICount               Count,         /* In */
    UDISizeT               Size,          /* In */
    UDICount *             CountDone,     /* Out */
    UDIBool                HostEndian     /* In */
    );
```

Description

UDIWrite is the same as **UDIRead** (on page 3-36), except data flows from the DFE to the TIP.

Return Codes

`UDIErrorUnknownResourceSpace`
`UDIErrorInvalidResource`
`UDIErrorResourceNotWriteable`

UDIDFE calls

The UDIDFE calls on the following pages are all provided by the DFE and called by the TIP. The TIP is only allowed to call these services while in the process of handling a request from the DFE. The DFE, on receiving one of these calls, may make any UDI call to the TIP with the exception of transparent mode calls.

UDIDFEEndTIPIO

Call

```
UDIDFEEndTIPIO UDIParams((
    void
    ));
```

Description

The call to **UDIDFEEndTIPIO** provides a way for the TIP to indicate a logical boundary for a set of I/O requests. If there have been any **UDIIOTypeTIP*xxx*** *IOType* calls made by the TIP during the handling of a UDI request, this call must be made before returning from the original UDI request. The DFE may find it useful to close a special I/O window.

Note that on any call to **UDIDFEPutOutput** or **UDIDFEGetInput**, the DFE may return **UDIErrorAborted**, indicating that the user wants to terminate this "TIP Screen I/O session" and the TIP should then call **UDIDFEEndTIPIO** and should try to finish the original UDI request without further input and output requests. (Remember that these calls can only be made by the TIP while it is servicing some UDI request from the DFE.)

UDIDFEEvalExpression

Call

```
UDIDFEEvalExpression (
    char *              Expression,         /* In */
    UDIInt              KindofAnswer,       /* Out */
    UDIResource *       AnswerResource,     /* Out */
    UDIHostMemPtr       AnswerBuf,          /* Out */
    UDISizeT            BufSize,            /* In */
    UDICount *          CountDone,          /* Out */
    UDISizeT *          Size,               /* Out */
    UDIExprType *       Type,               /* Out */
    );
```

Description

By calling **UDIDFEEvalExpression**, the TIP is asking the DFE to evaluate an expression. The input parameter *Expression* will contain the zero–terminated ASCII string to be evaluated. The TIP, not knowing whether the expression will evaluate to a value or a resource address, can provide an *AnswerBuf* and *AnswerResource*. Optionally, either *AnswerBuf* or *AnswerResource* can be set to NULL if the TIP is not interested in that kind of answer. (The DFE will then return an error if the expression did in fact evaluate to the kind of answer that was set to NULL.) If *AnswerBuf* is provided, the TIP also passes the size of the *AnswerBuf* in bytes.

The DFE treats the string as an expression, and evaluates it using its own expression evaluation rules.

If the expression evaluates to a value or a set of values, *KindOfAnswer* is set to **UDIExprKindValue** and the values are returned in *AnswerBuf*. (The *AnswerResource* parameter is not used in this case.) *CountDone* indicates the number of values returned, *Size* indicates the size of each value in bytes, and *Type* contains simple type information for the expression. The endian type of the objects is always host–endian (i.e. the data is in host byte order). Note that the DFE is constrained to return a single *Size* descriptor. Thus, while arrays of scalar values could be returned, structures which mixed size fields could not. The *Type* information returned is fairly basic, being one of the following:

```
    UDITypeUnknown         /* type not applicable, or DFE doesn't
                            * support types */
    UDITypeOther           /* type is known but does not match one
in
                            * the UDI list */
    UDITypeChar            /* an 8-bit ASCII character */
    UDITypeInt             /* an integral type. With Size, handles
                            * short, int, long */
    UDITypeFloat           /* with Size, handles float, double, long
                            * double */
```

If *BufSize* was not big enough to hold the values, (i.e. *CountDone* * *Size* > *BufSize*), the DFE leaves the buffer unfilled and returns the error **UDIDFEErrorBufTooSmall** but still sets *CountDone*, *Size*, and *Type* as if the buffer had been big enough. This allows the TIP to reissue the request with a large enough buffer if it so desires.

If the expression evaluates to an "address," *KindofAnswer* is set to **UDIExprKindResource** and the resource description (class and offset) is returned in *AnswerResource*. (The *AnswerBuf* parameter is not used in this case). If the DFE knows the type of the data that the address is pointing at, it should set *Type*, *Count*, and *Size* appropriately. If the DFE does not know the type, size, or count of the data that the address is pointing at, it can return: *Type* = **UDITypeUnknown** and *Count* or *Size* = 0.

Return Codes

```
UDIDFEErrorCouldNotEvaluate
UDIDFEErrorEvaluatedToValue
         (This could happen if the supplied
          AnswerBuf was NULL.)
UDIDFEErrorEvaluatedToResource
         (This could happen if the supplied
          AnswerResource was NULL.)
UDIDFEErrorBufTooSmall
```

UDIDFEEvalResource

Call

```
UDIDFEEvalResource UDIParams((
    UDIResource           ResourceToEval,       /* In */
    UDIBool               ExactEval,            /* In */
    char *                SymbolAnswer,         /* Out */
    UDISizeT              BufSize,              /* In */
    ));
```

Description

This function is provided by the DFE and called by the TIP. The TIP is only allowed to call this function while in the process of handling a request from the DFE. By calling **UDIDFEEvalResource**, the TIP is asking the DFE to map the resource *ResourceToEval* to a symbolic expression. The DFE should map this resource to an ASCII string of the form *symbolname* or *symbolname* + *offset*, where *symbolname* is the nearest symbol whose space is the same as *resource.space* and whose offset is less than *resource.offset*.

If *ExactEval* is true, the symbol name's offset must match the *resource.offset* exactly. The DFE returns the answer in the form of a null–terminated ASCII string in the *SymbolAnswer* parameter. *BufSize* indicates the maximum size of the *SymbolAnswer* buffer in bytes. If the resource cannot be mapped to a string, a null string is returned along with one of the errors listed below.

Return Codes

`UDIDFEErrorCouldNotEvaluate`
`UDIDFEErrorNoExactEval`
`UDIDFEErrorBufTooSmall`
`UDIErrorUnknownResourceSpace`
`UDIErrorInvalidResource`

UDIDFEGetInput

Call

```
UDIDFEGetInput UDIParams((
    UDIHostMemPtr          Buf,           /* Out */
    UDIIOType              IOType,        /* In */
    UDISizeT               BufSize,       /* In */
    UDIMode                Mode           /* In */
    UDISizeT *             CountDone      /* Out */
    ));
```

Description

This function is provided by the DFE and called by the TIP. The TIP is only allowed to call this function while in the process of handling a request from the DFE. *Buf* is a set of maximum *BufSize* bytes that the TIP wants the DFE to fill with input from the user. *Mode* indicates the mode to be used to get the input, the default is line–buffered, echoed, with editing. Other modes are described under the description of the **UDIStdinMode** call. *IOType* can be one of the following:

- UDIIOTypeTIPStdin
- UDIIOTypeTargetStdin

In other words, *IOType* indicates the destination of the input (TIP or Target). A DFE may want to get input from the user differently, depending on the *IOType*, for example, a windowed debugger may get TIP input and target input from different windows. On return, the DFE sets *CountDone* to indicate how many bytes were actually input from the user.

Using **UDIIOTypeTIPStdin** allows the TIP to get a response from the user to let the TIP know how to proceed while handling a UDI request. It is often used together with **UDIDFEPutOutput** (with parameter **UDIIOTypeTIPStdout**), which presents TIP–specific information to the user.

Using **UDIIOTypeTargetStdin** allows the TIP to get input from the DFE for a running program. This represents an alternative to returning **UDIStdInNeeded** from **UDIWait** and waiting for the DFE to call **UDIPutStdin**. For some TIPs, this may be a simpler alternative.

Return Codes

`UDIErrorAborted`

UDIDFEPutOutput

Call

```
UDIDFEPutOutput (
    UDIHostMemPtr         Buf,          /* In */
    UDIIOType             IOType,       /* In */
    UDISizeT              Count,        /* In */
    UDISizeT *            CountDone     /* Out */
    );
```

Description

This function is provided by the DFE and called by the TIP. The TIP is only allowed to call this function while in the process of handling a request from the DFE. *Buf* contains a set of *Count* bytes that the TIP wants the DFE to present to the user. *IOType* can be one of the following:

- UDIIOTypeTIPStdout
- UDIIOTypeTIPStderr
- UDIIOTypeTargetStdout
- UDIIOTypeTargetStderr

Using **UDIIOTypeTIPStdout** (or **TIPStderr**) allows the TIP to present TIP–specific status information to the user. When used together with **UDIDFEGetInput** (with parameter **UDIIOTypeTIPStdin**), the TIP can get a response from the user indicating how to proceed while handling a UDI request.

Using **UDIIOTypeTargetStdout** (or **TargetStderr**) allows the TIP to feed output from a running program to the DFE . This represents an alternative to returning **UDIStdOutReady** or **UDIStderrReady** from **UDIWait** and waiting for the DFE to call **UDIGetStdout** or **UDIGetStderr**. For some TIPs, this may be a simpler alternative.

Return Codes

`UDIErrorAborted`

Chapter 4

UDI IPC Methods for DOS Hosts

This chapter specifies how UDI DFEs and TIPs communicate using the DOS IPC mechanism. The UDI version covered is UDI 1.4 but compatibility with previous versions of UDI (back to version 1.2) is also discussed. (For complete information on compatibility and interoperability of different version TIPs and DFEs, please see page D–1.)

NOTE: Refer back to Chapter 3 for semantics on most of the fields of the calls.

In the DOS IPC mechanism, the TIP makes itself resident in memory along with a table of pointers to its UDI service routines. The DFE locates this table and calls through the table to the TIP routines. The DFE and TIP thus use a shared memory scheme and can pass real–mode pointers to each other.

The sections below describe how the DFE connects to the TIP, and the calling conventions and the format of the stack when the DFE wants to call a routine in the TIP or vice versa.

NOTE: All pointers in this specification are real–mode FAR pointers consisting of a 16–bit offset followed by a 16–bit segment. All data, unless otherwise noted, are in little–endian format. In particular, on calls that use a *HostEndian* parameter (like **UDIRead** and **UDIWrite**), *HostEndian* = 1 implies little endian.

Establishing the Connection

The TIP places itself in memory in some manner that is not defined by UDI (usually using DOS terminate and stay–resident calls). A TIP has a **TIPVecRec** structure which the DFE uses to communicate with the TIP. The format of the **TIPVecRec** structure is:

```
struct TIPVecRec {
    union rec recognizer;         /* 0xcf, 'u', 'd', 'i' */
    struct UDIVecRec FAR *Next;   /* Pointer to next TIP */
    struct UDIVecRec FAR *Prev;   /* Pointer to previous TIP */
    char FAR *exeName;            /* Name that the DFE uses when
                                   * searching for a TIP and the
                                   * remainder of the structure
                                   * are far pointers to the UDI
                                   * routines in the TIP */
/* These routines are described separately below */
    UDIConnect_12,
    UDIDisconnect,
    UDISetCurrentConnection,
    UDICapabilities_12,
    UDIGetErrorMsg,
    UDIGetTargetConfig,
    UDICreateProcess,
    UDISetCurrentProcess,
    UDIDestroyProcess,
    UDIInitializeProcess,
    UDIRead,
    UDIWrite,
    UDICopy,
    UDIExecute,
    UDIStep,
    UDIStop,
    UDIWait,
    UDISetBreakpoint_13,
    UDIQueryBreakpoint_13,
    UDIClearBreakpoint,
    UDIGetStdout,
    UDIGetStderr,
    UDIPutStdin,
    UDIStdinMode,
    UDIPutTrans,
    UDIGetTrans,
    UDITransMode
/* Fields below this did not exist in UDI 1.2 TipVecRec */
    Signature_1.3          /* characters "1.3\0", identifies the
                            * newer TipVecRec structure */
    TIPIPCId               /* so DFE can identify UDI level from
                            * structure */
    UDIConnect_13
    UDIFind
    UDICapabilities_13
    UDIConnect_14
    UDISetBreakpoint_14
    UDIQueryBreakpoint_14
```

Before becoming resident, the TIP links its **TIPVecRec** structure into the UDI chain. The UDI chain consists of a doubly linked list of **TIPVecRec** structures anchored at a single interrupt vector between 60h and 66h.

The UDI chain is located through the following process:

Each interrupt vector from 60h to 66h is inspected in turn. If the interrupt vector points to a location whose first four bytes contain the **TIPVecRec** signature, that is assumed to be the UDI chain anchor. If no interrupt vector from 60h to 66h matches the **TIPVecRec** signature, (as would be true for the first TIP) then the first vector in that 60h through 66h group that equals NULL is selected as the UDI chain anchor.

The *exeName* field of the **TIPVecRec** points to a "TIP identifier" string which the TIP uses to identify itself and which the DFE uses when it searches down the UDI chain for a particular TIP. The way the TIP chooses this TIP identifier string and the manner in which the DFE learns the TIP identifier string is outside the scope of this specification. (In the sample AMD implementation, the DFE actually spawns the TIP and the TIP assumes argv[1] is the identifier string.)

When the DFE locates the UDI chain and finds a TIP with the correct TIP identifier string, it must first call the TIP's **UDIConnect** routine. The parameters for the **UDIConnect** routine are discussed below. Notice that one of the things the DFE passes to the TIP at **UDIConnect** time is a pointer to a **DFEVecRec** structure. The **DFEVecRec** structure contains call entry points for the UDIDFE calls. The TIP uses this **DFEVecRec** structure to call back into the DFE. The structure of this **DFEVecRec** is:

```
struct     DFEVecRec {
    UDIDFEEvalExpression   /* all FAR pointers */
    UDIDFEEvalResource
    UDIDFEGetInput
    UDIDFEPutOutput
    UDIDFEEndTIPIO
    }
```

General Call Interface Information

The information here applies to both DFE–to–TIP calls and to TIP–to–DFE calls. The processor must be in real mode at the time of the call. On entry, the called routine may not assume the contents of any registers except CS, IP, SS, and SP. In all respects, the calling convention is that for Microsoft C compiled large model using the compiler option—Alfu (DS != SS). Refer to Microsoft documentation for further details on this calling convention.

Typedefs of UDI Parameters

The following type definitions are used in many of the routine descriptions of the specific calls and parameters (starting on page 4-6).

```
typedef unsigned long     UDIUInt32;     /* unsigned integers */
typedef unsigned short    UDIUInt16;
typedef unsigned char     UDIUInt8;
typedef long              UDIInt32;      /* 32-bit integer */
typedef short             UDIInt16;      /* 16-bit integer */
typedef char              UDIInt8;       /* unreliable signedness
*/
typedef UDIInt16          UDIInt;
typedef UDIUInt16         UDIUInt;
typedef UDIUInt16         UDISizeT;
typedef UDIInt            UDIError;
typedef UDIInt            UDISessionId;
typedef UDIInt            UDIPId;
typedef UDIInt            UDIStepType;
typedef UDIInt            UDIBreakType;
typedef UDIUInt           UDIBreakId;
typedef UDIUInt           UDIMode;
typedef void              UDIVoid;       /* void type */
typedef void *            UDIVoidPtr;    /* void pointer type */
typedef void FAR*         UDIHostMemPtr; /* arbitrary mem pointer
*/
typedef UDIInt16          UDIBool;
typedef UDIStruct
{
    CPUSpace       Space;
    CPUOffset      Offset;
 } UDIResource;
typedef    UDIStruct
{
    CPUOffset      Low;
    CPUOffset      High;
 } UDIRange;
typedef UDIStruct
{
    CPUSpace       Space;
    CPUOffset      Offset;
    CPUSizeT       Size;
} UDIMemoryRange;
```

Specific Calls and Parameters

This section describes the parameters for each call that the DFE can make into the TIP. The calls are arranged alphabetically on the following pages for easy access. In most cases, the semantics of each field of the messages is described in Chapter 3. Message field descriptions not found in Chapter 3 are located in the descriptions of the specific messages on the following pages.

NOTE: Two calls described in Chapter 3 are handled entirely on the DFE side and do not result in calls across the IPC interface to the TIP. These calls are **UDIEnumerateTIPs** and **UDISetCurrentConnection**.

NOTE: (For UDI 1.2) On all calls except **UDIConnect** and **UDIDisconnect**, the *connection_id* field appears as the last parameter to the call. This parameter is new to UDI 1.3 and later versions and was not defined in UDI 1.2. Because of the calling convention, it can be sent without harm to a 1.2 TIP but the TIP will ignore it. Also, it will not be valid for calls from a 1.2 DFE and must be ignored.

UDICapabilities

Call

```
UDIError (FAR *UDICapabilities_12) UDIParams((
    UDIUInt32 far *        TIPId,                /* Out */
    UDIUInt32 far *        TargetId,             /* Out */
    UDIUInt32              DFEId,                /* In */
    UDIUInt32              DFE,                  /* In */
    UDIUInt32 far *        TIP,                  /* Out */
    UDIUInt32 far *        DFEIPCId,             /* Out */
    UDIUInt32 far *        TIPIPCId,             /* Out */
    char far *             TIPString             /* Out */
));
 UDIError (FAR *UDICapabilities_13) UDIParams((
    UDIUInt32 far *        TIPId,                /* Out */
    UDIUInt32 far *        TargetId,             /* Out */
    UDIUInt32              DFEId,                /* In */
    UDIUInt32              DFE,                  /* In */
    UDIUInt32 far *        TIP,                  /* Out */
    UDIUInt32 far *        DFEIPCId,             /* Out */
    UDIUInt32 far *        TIPIPCId,             /* Out */
    char far *             TIPString             /* Out */
    UDISizeT               BufSize,              /* In - not used
in 1.2 */
    UDISizeT far *         CountDone,            /* Out - not used
in 1.2 */
    UDISessionId           connection_id
));
```

Description

The semantics of the parameters above are described on page 3-11 and the following additional information should also be noted. The first call from the DFE to the TIP must be a **UDIConnect** call. If the **UDIConnect** request succeeds, the second message must be a **UDICapabilities** request. **UDICapabilities_12** will only be called by 1.2 DFEs, **UDICapabilities_13** will only be called by 1.3 DFEs.

With the UDI 1.3 **UDICapabilities** call, the maximum length of the returned *TIPString* is limited to *RqMsg.BufSize* characters (including the terminator). If exactly *BufSize* characters are returned, this indicates that the TIP may have more characters to return in its *TIPString* and the DFE must then send another **UDICapabilities** request and the TIP must return the next *BufSize* characters.

With the UDI 1.2 **UDICapabilities** request, the maximum length of the returned *TIPString* is limited to 80 characters (including the terminator).

UDIClearBreakpoint

Call

```
UDIError (FAR *UDIClearBreakpoint) UDIParams((
    unsigned int          BreakId               /* In */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-11.

UDIConnect

Call

```
UDIError (FAR *UDIConnect_12) UDIParams((
    char           far *   tip_parameters        /* In */
    UDISessionId   far *   connection_id,        /* Out */
    struct DOSTerm far *   TermStruct            /* In */
     ));
 UDIError (FAR *UDIConnect_13) UDIParams((
    char           far *   tip_parameters        /* In */
    UDISessionId   far *   connection_id,        /* Out */
    struct DOSTerm far *   TermStruct            /* In */
    UDIUInt32              DFEIPCId              /* In -- not used
in 1.2 */
    UDIUInt32      far *   TIPIPCId              /* Out -- not used
in 1.2 */
    struct DFEVecRec far * VecRec                /* In  -- not used
in 1.2 */
    UDISessionId           DFESessionId          /* In -- new with
1.3 */
    ));

UDIError (FAR *UDIConnect_14) UDIParams((
    char far *                    tip_parameters /* In */
    UDISessionId far *            connection_id, /* Out */
    struct DOSTerm far *          TermStruct     /* In */
    UDIUInt32                     DFEIPCId       /* In -- not used
in 1.2 */
    UDIUInt32      far *          TIPIPCId       /* Out -- not used
in 1.2 */
    struct DFEVecRec far *        VecRec         /* In  -- not used
in 1.2 */
    UDISessionId           DFESessionId          /* In -- new with
1.3 */
    UDIUInt32                     DFE            /* In -- new with
1.4 */
    ));
```

where:

tip_parameters This is an ASCII string which the TIP should interpret as TIP–specific configuration parameters.

connection_id This uniquely defines this UDI connection for this TIP. It is passed on every call after **UDIConnect** by UDI 1.3 DFEs. (See notes on UDI 1.2, 1.3, and 1.4 compatibility on page D–1).

TermStruct It is a requirement of UDI that all UDIxxx calls return to the DFE, even a call where the TIP chooses to remove itself from memory. A TIP might remove itself from memory on either a failing **UDIConnect** or on a **UDIDisconnect** of the last connection.

The *TermStruct* parameter provides a way for the TIP to remove itself from memory and still return back to the DFE. The *TermStruct* parameter has the following format:

```
UDIStruct DOSTerm {
    void (far *TermFunc)(void);
    UDIUInt16 sds;
    UDIUInt16 sss;
    UDIUInt16 ssi;
    UDIUInt16 sdi;
    UDIUInt16 ssp;
    UDIUInt16 retval;
    UDIUInt16 sbp;
};
```

The semantics are such that if the TIP stores the DS, SS, SI, SP, and BP registers, which exist at the entry point to the call (**UDIConnect** or **UDIDisconnect**), into the sds, sss, ssi, ssp, retval, and sbp fields, stores the return error code into the retval field, and also sets *TermFunc* as its PSP exit function (offset 16 in the PSP) before calling the DOS exit function, then flow will return properly to the DFE caller after the DOS exit by the TIP.

If the TIP is not removing itself from memory, *TermStruct* can be ignored.

DFEIPCId The lowest 12 bits define the DFE IPC version number. For UDI1.2 DFEs, this will be 0x12N. For UDI1.3 DFEs, this will be 0x13N. In each case, N is a minor version indicator and should be ignored by the TIP. This parameter is present so that in future versions of UDI, the TIP will know the UDI version of the DFE.

DFESessionId
This is a number that uniquely identifies this connection for the DFE. This is passed by the DFE to the TIP at connect time and then used by the TIP on each of the callbacks and allows the DFE to distinguish which TIP the callback came from.

DFE The DFE parameter has the same semantics as the DFE parameter in the **UDICapabilities** call; it indicates to the TIP what version of UDI the DFE prefers, i.e., the latest version number the DFE understands. (This parameter appears in the UDIConnect 1.4 procedural interface).

NOTE: When the return value from **UDIConnect** is negative, the TIP must stay in memory and the DFE must call **UDIGetErrorMessage** to get the error text. See Appendix A for descriptions of UDI errors, including **UDIErrorTryAnotherTIP** and **UDIErrorConnectionUnavailable**.**?**

Description

The first call from the DFE to the TIP must be a **UDIConnect** call. There are three different UDIConnect entries in the call table, the **UDIConnect_12** entry is used only by 1.2 DFEs; the **UDIConnect_13** entry is used only by 1.3 and later DFEs and the **UDIConnect_14** adds the DFE parameter supported by the UDIConnect_14 procedural interface. Because a TIP must have installed its **TIPVecRec** table in memory before a DFE tries to connect to it, a DFE can tell the IPC version of the TIP by looking for the *Signature_1.3* field and the *TIPIPCId* in the **TIPVecRec**. The absence of the *Signature_1.3* field marks the TIP as 1.2.

A TIP knows the IPC version of the DFE by looking at the DFEIPCId parameter of the connect request. A 1.2 DFE can be detected because it is the only one who will call the old **Connect_12** entry point.

UDICopy

Call

```
UDIError (FAR *UDICopy) UDIParams((
    UDIResource           From,                /* In */
    UDIResource           To,                  /* In */
    UDICount              Count,               /* In */
    UDISizeT              Size,                /* In */
    UDICount far *        CountDone,           /* Out */
    UDIBool               direction,           /* In */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-17.

UDICreateProcess

Call

```
UDIError (FAR *UDICreateProcess) UDIParams((
    UDIPId far *          Id,                   /* Out */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-18.

UDIDestroyProcess

Call

```
UDIError (FAR *UDIDestroyProcess) UDIParams((
    UDIPId              PId                 /* In */
    UDISessionId        connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-19.

UDIDisconnect

Call

```
UDIError (FAR *UDIDisconnect) UDIParams((
    UDISessionId          connection_id,        /* In */
    UDIBool               Terminate,            /* In */
    struct DOSTerm far *  TermStruct            /* In - not seen
in UDIP */
));
```

where:

connection_id Is the one returned by **UDIConnect**.

Description

The semantics of the parameters are described on page 3-20.

NOTE: For *TermStruct*, see the description under **UDIConnect** on page 4-8.

UDIExecute

Call

```
UDIError (FAR *UDIExecute) UDIParams((
    UDISessionId      connection_id
));
```

Description

The semantics of the parameters are described on page 3-22.

UDIFind

Call

```
UDIError UDIFind UDIParams((
    UDIMemoryRange        WhereToLook,          /* In */
    UDIInt32              Stride,               /* In */
    UDIHostMemPtr         Pattern,              /* In */
    UDIHostMemPtr         PatternMask,          /* In */
    UDICount              PatternCount,         /* In */
    UDISizeT              PatternSize,          /* In */
    UDIBool               PatternHostEndian,    /* In */
    UDICount              MaxToFind,            /* In */
    UDICount *            CountFound,           /* Out */
    CPUOffset             FoundAtOffset[],      /* Out */
    UDIHostMemPtr         FoundValues[],        /* Out */
    UDISessionId          connection_id
));
```

Description

The semantics of the parameters are described on page 3-23.

UDIGetErrorMessage

Call

```
UDIError (FAR *UDIGetErrorMsg) UDIParams((
    UDIError              ErrorCode,            /* In */
    UDISizeT              MsgSize,              /* In */
    char far *            Msg,                  /* Out */
    UDISizeT far *        CountDone             /* Out */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-24.

UDIGetStderr

Call

```
UDIError (FAR *UDIGetStderr) UDIParams((
    UDIHostMemPtr         Buf,                  /* Out */
    UDISizeT              BufSize,              /* In */
    UDISizeT far *        CountDone             /* Out */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-25.

UDIGetStdout

Call

```
UDIError (FAR *UDIGetStdout) UDIParams((
    UDIHostMemPtr          Buf,                  /* Out */
    UDISizeT               BufSize,              /* In */
    UDISizeT far *         CountDone             /* Out */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-26.

UDIGetTargetConfig

Call

```
UDIError (FAR *UDIGetTargetConfig) UDIParams((
    UDIMemoryRange far     KnownMemory[],        /* Out */
    UDIInt far *           NumberOfRanges,       /* In/Out */
    UDIUInt32 far          ChipVersions[],       /* Out */
    UDIInt  far *          NumberOfChips,        /* In/Out */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-27.

UDIGetTrans

Call

```
UDIError (FAR *UDIGetTrans) UDIParams((
    UDIHostMemPtr          Buf,                /* Out */
    UDISizeT               BufSize,            /* In */
    UDISizeT far *         CountDone           /* Out */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-28.

UDIInitializeProcess

Call

```
UDIError (FAR *UDIInitializeProcess) UDIParams((
    UDIMemoryRange far     ProcessMemory[],      /* In */
    UDIInt                 NumberOfRanges,       /* In */
    UDIResource            EntryPoint,           /* In */
    CPUSizeT far           StackSizes[],         /* In */
    UDIInt                 NumberOfStacks,       /* In */
    char far *             ArgString,            /* In */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-30.

UDIPutStdin

Call

```
UDIError (FAR *UDIPutStdin) UDIParams((
    UDIHostMemPtr          Buf,                /* In */
    UDISizeT               Count,              /* In */
    UDISizeT far *         CountDone           /* Out */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-32.

UDIPutTrans

Call

```
UDIError (FAR *UDIPutTrans) UDIParams((
    UDIHostMemPtr          Buf,                  /* In */
    UDISizeT               Count,                /* In */
    UDISizeT far *         CountDone             /* Out */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-33.

UDIQueryBreakpoint

Call

```
UDIError (FAR *UDIQueryBreakpoint 1.2) UDIParams((
    UDIBreakId             BreakId,       /* In */
    UDIResource far *      Addr,          /* Out */
    UDIInt32 far *         PassCount,     /* Out */
    UDIBreakType far *     Type,          /* Out */
    UDIInt32 far *         CountRemaining,/* Out */
    UDISessionId           connection_id
    ));

UDIError (FAR *UDIQueryBreakpoint_14) UDIParams((
    UDIBreakId             BreakId,              /* In */
    UDISizeT               BufSize,              /* In */
    UDIBreakInfo far *     BreakInfo,            /* Out */
    UDIBreakId far *       NextBreakId           /* Out */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-34.

The UDIQueryBreakpoint_13 call returns a subset of the information returned by UDIQueryBreakpoint_14. Ranges are not supported. Only the low 4 bits of Type are defined. Any information in the vendor-specific Buf field of a 1.4 breakpoint cannot be returned to UDIQueryBreakpoint_13.

In actual practice it is unlikely that these mapping problems will arise since they require a new DFE to set up breakpoints on a new TIP, then disconnect (without terminating the TIP) and have an old DFE connect and query the breakpoints on the TIP.

UDIRead

Call

```
UDIError (FAR *UDIRead) UDIParams((
    UDIResource           From,                /* In */
    UDIHostMemPtr         To,                  /* Out */
    UDICount              Count,               /* In */
    UDISizeT              Size,                /* In */
    UDICount far *        CountDone,           /* Out */
    UDIBool               HostEndian,          /* In */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-36.

UDISetBreakpoint

Call

```
UDIError (FAR *UDISetBreakpoint_13) UDIParams((
    UDIResource           Addr,                 /* In */
    UDIInt32              PassCount,            /* In */
    UDIBreakType          Type,                 /* In */
    UDIBreakId far *      BreakId ,             /* Out */
    UDISessionId          connection_id
    ));

UDIError (FAR *UDISetBreakpoint_14) UDIParams((
    UDIBreakInfo          *BreakInfo,           /* In */
    UDIBreakId far        *BreakId,             /* Out */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-37.

The **UDISetBreakpoint_13** call allows setting breakpoints with a subset of the information that is possible with **UDISetBreakpoint_14**. Ranges are not supported. Only the low 4 bits of Type are defined. The **CountRemaining** must be initialized to the absolute value of the **PassCount**. No vendor-specific Buf field is supported.

UDISetCurrentProcess

Call

```
UDIError (FAR *UDISetCurrentProcess) UDIParams((
    UDIPId              PId                 /* In */
    UDISessionId        connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-41.

UDIStdinMode

Call

```
UDIError (FAR *UDIStdinMode) UDIParams((
    UDIMode        far *   Mode                  /* Out */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-42.

UDIStep

Call

```
UDIError (FAR *UDIStep) UDIParams((
    UDIUInt32             Steps,                /* In */
    UDIStepType           StepType,             /* In */
    UDIRange              Range                 /* In */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-43.

UDIStop

Call

```
UDIVoid (FAR *UDIStop) UDIParams((
    UDISessionId            connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-44.

NOTE: **UDIStop** is unique in that it can be called by the DFE even before the TIP has returned from the previous UDI call. (It can also be called after a **UDIExecute** and before a **UDIWait**).

UDITransMode

Call

```
UDIError (FAR *UDITransMode) UDIParams((
    UDIMode  far *         Mode,                 /* Out */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-45.

UDIWait

Call

```
UDIError (FAR *UDIWait) UDIParams((
    UDIInt32              MaxTime,              /* In */
    UDIPId far *          PId,                  /* Out */
    UDIUInt32 far *       StopReason,           /* Out */
    UDISessionId          connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-46.

UDIWrite

Call

```
UDIError (FAR *UDIWrite) UDIParams((
    UDIHostMemPtr          From,                /* In */
    UDIResource            To,                  /* In */
    UDICount               Count,               /* In */
    UDISizeT               Size,                /* In */
    UDICount far *         CountDone,           /* Out */
    UDIBool                HostEndian,          /* In */
    UDISessionId           connection_id
    ));
```

Description

The semantics of the parameters are described on page 3-50.

UDIDFE calls

The calls on the following pages are made from the TIP to the DFE. All UDIDFE*xxx* calls can only be made by the TIP while it is in the middle of servicing some UDI request from the DFE (i.e., before it has returned from the UDI call). In addition, while the DFE is servicing the UDIDFE*xxx* request, the DFE can issue a new UDI request to the TIP. The call traffic in this last case would be:

--> **UDI***xxx* called by DFE

<-- **UDIDFE***yyy* called by TIP

--> **UDI***zzz* called by DFE

<-- **UDI***zzz* returns

--> **UDIDFE***yyy* returns

<-- **UDI***xxx* returns

UDIDFEEndTIPIO

Call

```
UDIDFEEndTIPIO UDIParams((
    UDISessionId          connection_id
 ));
```

Description

The semantics of the parameters are described on page 3-52.

UDIDFEEvalExpression

Call

```
UDIDFEEvalExpression UDIParams((
    char *          Expression,           /* In */
    UDIInt          KindofAnswer,         /* Out (None, Resource,
Value)     */
    UDIResource *   AnswerResource,       /* Out (used when Kind =
Resource) */
    UDIHostMemPtr   AnswerBuf,            /* Out (used when Kind =
Value)    */
    UDIHostMemPtr   AnswerBuf,            /* Out (used when Kind =
Value)    */
    UDISizeT        BufSize,              /* In  (size of AnswerBuf
in bytes)*/
    UDICount *      CountDone,            /* Out (used when Kind =
Value)    */
    UDISizeT *      Size,                 /* Out (used when Kind =
Value)    */
    UDIExprType *   Type,                 /* Out (used when Kind =
Value)    */
    UDISessionId    connection_id
));
```

Description

Each response specifies (by *KindOfAnswer*) either an *AnswerResource* or an *AnswerBuf* with *CountDone*, *Size*, and *Type*.

Any data in *AnswerBuf* consists of objects of size *Size* and is always in the same endian format as the target.

UDIDFEEvalResource

Call

```
UDIDFEEvalResource UDIParams((
    UDIResource    ResourceToEval,        /* In */
    UDIBool        ExactEval,             /* In, true if Exact
                                           * Evaluation Desired */
    char *         SymbolAnswer,          /* Out */
    UDISizeT       BufSize,               /* In */
    UDISessionId   connection_id
 ));
```

Description

The semantics of the parameters are described on page 3-55.

UDIDFEGetInput

Call

```
UDIDFEGetInput UDIParams((
    UDIHostMemPtr  Buf,                 /* Out */
    UDIIOType      IOType,              /* In */
    UDISizeT       BufSize,             /* In */
    UDIMode Mode   Mode,                /* In */
    UDISizeT *     CountDone,           /* Out */
    UDISessionId   connection_id
 ));
```

Description

The semantics of the parameters are described on page 3-56.

UDIDFEPutOutput

Call

```
UDIDFEPutOutput UDIParams((
    UDIHostMemPtr  Buf,                 /* In */
    UDIIOType      IOType,              /* In */
    UDISizeT       Count,               /* In */
    UDISizeT *     CountDone,           /* Out */
    UDISessionId   connection_id
 ));
```

Description

The semantics of the parameters are described on page 3-58.

Chapter 5

UDI IPC Methods for UNIX Hosts

This chapter specifies how UDI DFEs and TIPs communicate using the socket–based IPC mechanism. The UDI version covered is UDI 1.4 but compatibility with previous versions of UDI (back to version 1.2) is also discussed. (For complete information on compatibility and interoperability of different version TIPs and DFEs, please see page D–1.)

NOTE: Refer back to Chapter 3 for semantics on most of the fields of the messages.

In the socket–based IPC mechanism, the DFE establishes a socket connection to the TIP. The DFE and TIP then exchange requests and responses by sending messages over that socket. Also, for a few unusual asynchronous situations, the DFE can send a signal to the TIP.

The sections below describe how the DFE connects to the TIP, the format of the request and response messages that are sent between the DFE and TIP on the socket, and the signals that the DFE can send to the TIP.

NOTE: In this chapter, when a reference is made to a UDIConnect request message, it means "either a **UDIConnect_12** request message or a **UDIConnect_14** request message" unless otherwise noted.

Establishing the Connection

The TIP and DFE communicate over a socket. The socket may be based on any of the socket address families. The method by which the TIP chooses a socket to listen on and the way the DFE learns the socket address and address family of the TIP is outside the scope of this specification. (As an example, the TIP and DFE could take the socket name as a startup parameter. In some cases, the DFE might actually spawn the TIP.)

The TIP calls **socket()**, **bind()**, **listen()**, and **accept()**. After an **accept()** establishes a socket descriptor, the TIP expects all data for that connection from the DFE to arrive on that same socket descriptor.

For each connection the DFE wants to make, even if it is another connection to the same TIP, the DFE calls **socket()** and **connect()**, creating a new socket descriptor. All messages for that connection are then sent over that same socket descriptor.

Note that a TIP which needs to support more than one connection will have to listen for messages from more than one socket, and the socket that a message arrives on implies its connection ID.

General Message Format Information

Each request or response message consists of a string of bytes. The fields of each message are described below using the structure syntax. The fields of the messages are always packed on byte boundaries; there are no padding bytes between the fields.

The following field descriptors are used in the specific message formats which start on page 5-9. The endian type of the various fields is specified on page 5-4.

`UDIUInt32` An unsigned 32–bit integer sent as 4 bytes (endian type specified on page 5-4).

`UDIInt32` A signed 32–bit integer sent as 4 bytes (endian type specified on page 5-4).

`UDIByteArray` A string of bytes. The length or dimension of the array is always implicit from other information and is not sent as part of the array. (See the dimension = comments in the specific requests.) For the **`UDIRead`**, **`UDIWrite`**, **`UDIFind`** and **`UDIDFEEvalExpression`** requests, it is possible that the objects specified by the byte array are not actually bytes. In these cases, the endian type of the objects is described in the notes for each specific request.

```
UDIString   This is shorthand for:
                    struct {
                            UDIUInt32         Len;
                            UDIByteArray      Buf;
                    }
            A Len of zero is permitted. When the
            UDIString deals with actual ASCII
            strings, the null terminator byte is
            always included in both the length and
            the array.

UDIRange    This is shorthand for:
                    struct {
                            UDIUInt32         Low;
                            UDIUInt32         High;
                    }

UDIResource This is shorthand for:
                    struct {
                            UDIUInt32
                    Space;
                            UDIUInt32
                    Offset;
                    }
```

UDIMemoryRange

This is shorthand for:

```
                    struct {
                            UDIUInt32
                    Space;
                            UDIUInt32
                    Offset;
                            UDIUInt32         Size;
                    }

UDIBreakInfo
            This is shorthand for:
            struct {
                    UDIUInt32         Type;
                    UDIMemoryRange  Region;
                    UDIInt32 PassCount;
                    UDIInt32 CntRemaining;
                    UDIString         Buf;
            }
```

`UDIUInt32Array`
: This is a string of UDIUInt32 fields. The number of fields is not sent explicitly in the message but is implied by other parts of the request or response message. (See the dimension = comments in the specific requests.)

`UDIMemoryRangeArray`
: This is a string of UDIMemoryRange fields. The number of fields is not sent explicitly in the message but is implied by other parts of the request or response message. (See the dimension = comments in the specific requests.)

Endian Type of Fields in Messages

There are two types of fields in UNIX socket IPC messages:

`Target–Endian fields`
: These are the buffers used in **UDIRead**, **UDIWrite**, et cetera, that are explicitly specified to be target–endian by a UDI parameter i.e., *HostEndian* parameter set to FALSE.

`Host–Endian fields`
: All other fields in all IPC messages including those buffers used in **UDIRead**, **UDIWrite**, et cetera, for which the *HostEndian* parameter is set to TRUE.

The target–endian fields are always sent target–endian. It is assumed that both the DFE and TIP know the target–endian type.

The following discussion applies to the endian type of host–endian fields in socket IPC messages.

In UDI 1.3 and earlier, all host–endian fields are sent big–endian regardless of the endian type of either the DFE or TIP.

In UDI 1.4, the following rules apply which make it more efficient when both the DFE and TIP are little endian:

- In the **UDIConnectRqMsg** and until the **UDIConnectRespMsg** is received, all fields in all messages are always sent big endian.

- **UDIConnectRqMsg** has a *DFEIPCId* field and **UDIConnectRespMsg** has a *TIPIPCId* field.

- Bit 31 of each *xxxIPCId* field, if set, indicates a little–enidan host (bits 30-0 are the usual *IPCId* format).

- After the **UDIConnectRespMsg** has been sent, both sides know the endian type of the other end of the connection. In the case where both ends are little endian, all following messages on that connection use little–endian format for "host–endian" fields. If either end is big endian, all following messages on that connection use big–endian format for "host–endian" fields.

Request and Response Codes

The following table defines the codes for requests from the DFE to the TIP.

Table 5_1 **Codes for Requests from DFE to TIP**

Request	Code
UDIConnect_12_c	0
UDIDisconnect_c	1
Reserved	2
UDICapabilities_12_c	3
UDIEnumerateTIPs_c	4
UDIGetErrorMsg_c	5
UDIGetTargetConfig_c	6
UDICreateProcess_c	7
UDISetCurrentProcess_c	8
UDIDestroyProcess_c	9
UDIInitializeProcess_c	10
UDIRead_c	11
UDIWrite_c	12
UDICopy_c	13
UDIExecute_c	14
UDIStep_c	15
UDIStop_c	16
UDIWait_c	17
UDISetBreakpoint_13_c	18
UDIQueryBreakpoint_13_c	19
UDIClearBreakpoint_c	20
UDIGetStdout_c	21
UDIGetStderr_c	22
UDIPutStdin_c	23
UDIStdinMode_c	24
UDIPutTrans_c	25

UDIGetTrans_c	26
UDITransMode_c	27
Reserved	28
Reserved	29
UDIFind_c	30
UDICapabilities_13_c	31
UDIConnect_14_c	32
UDISetBreakpoint_14_c	33
UDIQueryBreakpoint_14_c	34

The following table defines the codes for requests from the TIP to the DFE.

Table 5_2. **Codes for Requests from TIP to DFE**

Request	Code
UDIDFEEvalExpression_c	1000
UDIDFEEvalResource_c	1001
UDIDFEGetInput_c	1002
UDIDFEPutOutput_c	1003
UDIDFEEndTIPIO_c	1004

For both UDI and UDIDFE requests, a response code is always a 32–bit unsigned integer equal to the request code but with the high bit set.

NOTE: The response message format for UDI 1.2 is slightly different. The *response_code* field is always omitted. The DFE always assumes that the first message that comes back on the socket is the response to the request. (This works because, in UDI 1.2, UDIDFE requests from the TIP to the DFE are not allowed.).

Signals from the DFE to the TIP

The DFE sends a “signal” to the TIP to implement the **UDIStop** functionality as described in Chapter 3.

On AF_UNIX sockets, the UNIX signal, SIGUSR1, is sent to the *tip_pid* which the TIP returns in the **UDIConnectRespMsg**.

On AF_INET sockets, an "out–of–bound" message is sent on the socket, which causes a SIGURG signal at the TIP.

Upon the receipt of either of these signals, the TIP should take the actions described under **UDIStop** in Chapter 3. No response as such is sent for the signal. If the TIP was in the middle of handling some UDI request when the signal came in, a response to that UDI request is sent back to the DFE with the error **UDIErrorAborted**. If the TIP was executing a target program, execution stops and the next **UDIWaitRqMsg** from the DFE should get a response with **UDIStopped**.

Specific Message Formats

The format of each request and response message between the DFE and the TIP are provided on the following pages. The messages are arranged alphabetically for easy access. In most cases, the semantics of each field of the messages is described in Chapter 3. Message field descriptions not found in Chapter 3 are located in the descriptions of the specific messages on the following pages.

NOTE: A few calls described in Chapter 3 do not result in request messages. These are **UDIEnumerateTIPs** and **UDISetCurrentConnection**. Also, the **UDIStop** call results in a signal rather than a message. These signals are described in the preceding section.

UDICapabilities

Message

```
struct UDICapabilities_13_RqMsg {
    UDIInt32       service_id;
    UDIUInt32      DFEId;
    UDIUInt32      DFE;
    UDIUInt32      BufSize;
 }
struct UDICapabilities_13_RespMsg {
    UDIInt32       response_id;
    UDIUInt32      TIPId;
    UDIUInt32      TargetId;
    UDIUInt32      TIP;
    UDIUInt32      Unused;
    UDIUInt32      TIPIPCId;
    UDIString      TIPString;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described in Chapter 3 and the following additional information should also be noted. The first message on the socket connection from the DFE to the TIP must be a **UDIConnect** message. If the **UDIConnect** request succeeds, the second message must be a **UDICapabilities** request.

With the UDI 1.3 **UDICapabilities** request, the maximum length of the returned *TIPString* is limited to *RqMsg.BufSize* characters (including the terminator). If exactly *BufSize* characters are returned, this indicates that the TIP may have more characters to return in its *TIPSTring* and the DFE must then send another **UDICapabilities** request and the TIP must return the next *BufSize* characters.

When either side of the connection is UDI 1.2 (as determined by the connection request IPC IDs), the following message format is used instead:

```
struct UDICapabilities_12_RqMsg {
    UDIInt32       service_id;
    UDIUInt32      DFEId;
    UDIUInt32      DFE;
 }
struct UDICapabilities_12_RespMsg {
    UDIInt32       response_id;
    UDIUInt32      TIPId;
    UDIUInt32      TargetId;
    UDIUInt32      TIP;
    UDIUInt32      Unused;
    UDIUInt32      TIPIPCId;
    UDIString      TIPString;
    UDIUInt32      err;
 }
```

With the UDI 1.2 **UDICapabilities** request, the maximum length of the returned *TIPString* is limited to 80 characters (including the terminator).

UDIClearBreakpoint

Message

```
struct UDIClearBreakpointRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      BreakId;
 }
struct UDIClearBreakpointRespMsg {
    UDIInt32       response_id;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-13.

UDIConnect

Message

```
struct UDIConnect_14_RqMsg {
    UDIInt32       service_id;
    UDIUInt32      DFEIPCId;
    UDIString      tip_parameters;
    UDIUInt32      DFE;    /* New with 1.4 */
 }
struct UDIConnect_14_RespMsg {
    UDIInt32       response_id;
    UDIUInt32      TIPIPCId;
    UDIUInt32      tip_pid;
    UDIUInt32      tip_id;
    UDIUInt32      err;
 }
```

The older format of this message is:

```
struct UDIConnect_12_RqMsg {
    UDIInt32       service_id;
    UDIUInt32      DFEIPCId;
    UDIString      tip_parameters;
}
```

with the response message being identical to **UDIConnect_14_RespMsg**

where:

DFEIPCId The lowest 12 bits define the DFE IPC version number. For UDI 1.2 DFEs, this will be 0x12N. For UDI 1.3 DFEs, this will be 0x13N. In each case, N is a minor version indicator and should be ignored by the TIP.

tip_parameters This is an ASCII string which the TIP should interpret as TIP–specific configuration parameters.

TIPIPCId The lowest 12 bits define the DFE IPC version number. For UDI 1.2 TIPs, this will be 0x12N. For UDI 1.3 TIPs, this will be 0x13N. In each case, N is a minor version indicator and should be ignored by the DFE.

tip_pid	The UNIX PID of the TIP. For AF_UNIX sockets, this is used to signal the TIP for **UDIStop** functionality (see signals below). For other address family sockets, this parameter is ignored.
tip_id	This uniquely defines the UDI connection for this TIP. It is used by the DFE in the **UDIDisconnect** message.
errno	When the *errno* is negative, the TIP socket connection must remain open and the DFE must send a **UDIGetErrorMessage** request to get the error text. See Appendix A for descriptions of **UDIErrorTryAnotherTIP** and **UDIErrorConnectionUnavailable**.
DFE	The DFE parameter has the same semantics as the DFE parameter in the **UDICapabilities** call; it indicates to the TIP what version of UDI the DFE prefers, i.e., the latest version number the DFE understands.

Description

The first message on the socket connection from the DFE to the TIP must be a UDIConnect_12 message. This is because older TIPs will not know how to respond to later vintage UDIConnect messages and the DFE cannot tell the version of the TIP until it sends some UDIConnect message. This complication is handled in the following manner:

If the TIP is a pre-1.4 TIP, the TIP acts on the UDIConnect_12 message and sends a UDIConnect_12 response. This concludes the UDIConnect activity for older TIPs.

Similarly, if the TIP is 1.4 or later, and the TIP gets a UDIConnect_12 message from a pre-1.4 DFE (as detected in the DFEIPCId), the TIP acts on the UDIConnect_12 message and sends a UDIConnect_12 response. . This concludes the UDIConnect activity for older DFEs connecting to newer TIPs.

However, if the TIP is 1.4 or later, and the TIP gets a UDIConnect_12 message from a 1.4 or later DFE, the TIP does not act on the UDIConnect_12 message but instead sends a UDIConnect_12 response that has the TIPIPCId field filled in and also contains the special error code, UDIErrorIPCConnectIncomplete. The 1.4 or later DFE who receives UDIErrorIPCConnectIncomplete then knows the TIP's IPCid and the DFE must then complete the Connect transaction by sending a second request. This second request is the one the TIP acts upon This second request can be either a UDIConnect_14 message or a UDIConnect_12 message.

UDICopy

Message

```
struct UDICopyRqMsg {
    UDIUInt32             service_id;
    UDIResource           From;
    UDIResource           To;
    UDIUInt32             Count;
    UDIUInt32             Size;
    UDIUInt32             Direction;
 }

struct UDICopyRespMsg {
    UDIInt32              response_id;
    UDIUInt32             CountDone;
    UDIUInt32             err;
 }
```

Description

The semantics of the parameters are described on page 3-17.

UDICreateProcess

Message

```
struct UDICreateProcessRqMsg {
    UDIUInt32      service_id;
 }
struct UDICreateProcessRespMsg {
    UDIInt32       response_id;
    UDIUInt32      PId;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-18.

UDIDestroyProcess

Message

```
struct UDIDestroyProcessRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      PId;
 }
struct UDIDestroyProcessRespMsg {
    UDIInt32       response_id;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-19.

UDIDisconnect

Message

```
struct UDIDisconnectRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      tip_id;        /* see UDIConnect */
    UDIUInt32      Terminate;
 }
struct UDIDisconnectRespMsg {
    UDIInt32       response_id;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-20.

UDIExecute

Message

```
struct UDIExecuteRqMsg {
    UDIUInt32      service_id;
 }

struct UDIExecuteRespMsg {
    UDIInt32       response_id;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-22.

UDIFind

Message

```
struct UDIFindRqMsg {
    UDIUInt32      service_id;
    UDIMemoryRange WhereToLook;
    UDIInt32       Stride;
    UDIUInt32      PatternCount;
    UDIUInt32      PatternSize;
    UDIUInt32      PatternHostEndian;
    UDIUInt32      MaxToFind;
    UDIByteArray   Pattern;              /* dimension =
                                          * PatternCount *
PatternSize */
    UDIUInt32      PatternMaskLen;       /* either zero or
                                          * PatternCount *
PatternSize */
    UDIByteArray   PatternMask;          /* dimension =
PatternMaskLen */
}
struct UDIFindRespMsg {
    UDIUInt32      response_id;
    UDIUInt32      CountFound;
    UDIUInt32Array FoundAtOffset; /* dimension = CountFound */
    UDIByteArray   FoundValues;   /* if PatternMaskLen = 0
                                   * dimension = 0 else
                                   * dimension =
                                   * CountFound * PatternCount *
PatternSize */
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-23.

Notes:

If the *PatternHostEndian* flag is 1, the 29K data in *UDIFindRqMsg.Pattern*, in *UDIFindRqMsg.PatternMask* (if any), and in *UDIFindRespMesg.FoundValues* (if any) must be in big–endian format (keeping in mind that the size of each object in each field is specified by the *UDIFindRqMsg.Size* parameter). Note that in this case, at the procedural interface this data is required to be in *HostEndian* format.

If the *PatternHostEndian* flag is 0, the 29K data in *UDIFindRqMsg.Pattern*, in *UDIFindRqMsg.PatternMask* (if any), and in *UDIFindRespMesg.FoundValues* (if any) must be in *TargetEndian* format. It is assumed that both the DFE and the TIP know the endian type of the target. Note that in this case, the data in these buffers is unmodified by the IPC for the procedural interface.

UDIGetErrorMessage

Message

```
struct UDIGetErrorMessageRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      ErrorCode;
    UDIUInt32      MsgSize;
 }
struct UDIGetErrorMessageRespMsg {
    UDIInt32       response_id;
    UDIString      Message;
    UDIUInt32      CountDone;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-24.

UDIGetStderr

Message

```
struct UDIGetStderrRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      BufSize;
 }
struct UDIGetStderrRespMsg {
    UDIInt32       response_id;
    UDIUInt32      CountDone;
    UDIByteArray   Buf;           /* dimension = CountDone */
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-25.

UDIGetStdout

Message

```
struct UDIGetStdoutRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      BufSize;
 }

struct UDIGetStdoutRespMsg {
    UDIInt32       response_id;
    UDIUInt32      CountDone;
    UDIByteArray   Buf;           /* dimension = CountDone */
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-26.

UDIGetTargetConfig

Message

```
struct UDIGetTargetConfigRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      MaxNumberOfRanges;
    UDIUInt32      MaxNumberOfChips;
 }
struct UDIGetTargetConfigRespMsg {
    UDIInt32              response_id;
    UDIMemoryRangeArray   KnownMemory;   /* size =
                                          * MaxNumberOfRanges */
    UDIUInt32             NumberOfRanges;
    UDIUInt32             NumberOfChips;
    UDIInt32Array         ChipVersions;  /* size =
                                          * NumberOfChips */
    UDIUInt32                    err;
 }
```

where:

MaxNumberOfRanges
: Specifies the maximum number of ranges that the TIP can return.

MaxNumberOfChips
: Specifies the maximum number of chips that the TIP can return.

Description

For the **UDIGetTargetConfigRespMsg**, note that the *KnownMemory* and *ChipVersions* arrays are handled differently. The dimension of the *KnownMemory* array is *MaxNumberOfRanges* (the DFE limit) whereas the dimension of the *ChipVersions* array is *NumberOfChips* (the actual TIP number).

UDIGetTrans

Message

```
struct UDIGetTransRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      BufSize;
 }
struct UDIGetTransRespMsg {
    UDIInt32       response_id;
    UDIUInt32      CountDone;
    UDIByteArray   Buf;           /* dimension = CountDone */
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-28.

UDIInitializeProcess

Message

```
struct UDIInitializeProcessRqMsg {
    UDIUInt32              service_id;
    UDIUInt32              NumberOfRanges;
    UDIMemoryRangeArray    ProcessMemory;        /* dimension =
                                                   * NumberOfRanges
*/
    UDIResource            EntryPoint;
    UDIUInt32              NumberOfStacks;
    UDIUInt32Array StackSizes;           /* dimension =
                                          * NumberOfStacks */
    UDIUInt32              PId;
 }

struct UDIInitializeProcessRespMsg {
    UDIInt32               response_id;
    UDIUInt32              err;
 }
```

Description

The semantics of the parameters are described on page 3-30.

UDIPutStdin

Message

```
struct UDIPutStdinRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      Count;
    UDIByteArray   Buf;           /* dimension = Count */
}
struct UDIPutStdinRespMsg {
    UDIInt32       response_id;
    UDIUInt32      CountDone;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-32.

UDIPutTrans

Message

```
struct UDIPutTransRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      Count;
    UDIByteArray   Buf;           /* dimension = Count */
}
struct UDIPutTransRespMsg {
    UDIInt32       response_id;
    UDIUInt32      CountDone;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-33.

UDIQueryBreakpoint

Message

```
struct UDIQueryBreakpoint_14_RqMsg {
    UDIUInt32      service_id;
    UDIUInt32      BreakId;
    UDIUInt32      BufSize;
 }
struct UDIQueryBreakpoint_13_RespMsg {
    UDIInt32       response_id;
    UDIBreakInfo   BreakInfo;
    UDIUInt32      NextBreakId;
    UDIUInt32      err;
 }
```

The older format of this message is as follows:

```
struct UDIQueryBreakpoint_13_RqMsg {
    UDIUInt32      service_id;
    UDIUInt32      BreakId;
 }
struct UDIQueryBreakpoint_13_RespMsg {
    UDIInt32       response_id;
    UDIResource    Addr;
    UDIInt32       PassCount;
    UDIUInt32      Type;
    UDIInt32       CountRemaining;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-34.

The **UDIQueryBreakpoint_13** call returns a subset of the information returned by **UDIQueryBreakpoint_14**. Ranges are not supported. Only the low 4 bits of Type are defined. Any information in the vendor-specific Buf field of a 1.4 breakpoint cannot be returned to **UDIQueryBreakpoint_13**.

In actual practice it is unlikely that these mapping problems will arise since they require a new DFE to set up breakpoints on a new TIP, then disconnect (without terminating the TIP) and have an old DFE connect and query the breakpoints on the TIP.

UDIRead

Message

```
struct UDIReadRqMsg {
    UDIUInt32             service_id;
    UDIResource           From;
    UDIUInt32             Count;
    UDIUInt32             Size;
    UDIUInt32             HostEndian;
 }

struct UDIReadRespMsg {
    UDIInt32              response_id;
    UDIUInt32             CountDone;
    UDIByteArray          Buf;    /* dimension = CountDone*Size */
    UDIUInt32             err;
 }
```

Description

The semantics of the parameters are described on page 3-36.

Notes:

If the *HostEndian* flag is 1, the 29k data in the *UDIReadRespMsg.Buf* must be in big–endian format (keeping in mind that the size of each object in the buffer is specified by the *UDIReadRespMsg.Size* parameter). Note that in this case, at the procedural interface this data is required to be in *HostEnidan* format.

If the *HostEndian* flag is 0, the 29k data in the *UDIReadRespMsg.Buf* must be in *TargetEndian* format. It is assumed that both the DFE and the TIP know the endian type of the target. Note that in this case, the data in this buffer is unmodified by the IPC for the procedural interface.

UDISetBreakpoint

Message

```
struct UDISetBreakpoint_14_RqMsg {
    UDIUInt32      service_id;
    UDIBreakInfo   BreakInfo;
}
struct UDISetBreakpoint_14_RespMsg {
    UDIInt32       response_id;
    UDIUInt32      BreakId;
    UDIUInt32      err;
 }
```

The older format of this message is as follows:

```
struct UDISetBreakpoint_13_RqMsg {
    UDIUInt32      service_id;
    UDIResource    Addr;
    UDIInt32       PassCount;
    UDIUInt32      Type;
 }
struct UDISetBreakpoint_13_RespMsg {
    UDIInt32       response_id;
    UDIUInt32      BreakId;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-37.

The **UDISetBreakpoint_13** call allows setting breakpoints with a subset of the information that is possible with **UDISetBreakpoint_14**. Ranges are not supported. Only the low 4 bits of Type are defined. The CountRemaining must be initialized to the absolute value of the PassCount. No vendor-specific Buf field is supported.

UDISetCurrentProcess

Message

```
struct UDISetCurrentProcessRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      PId;
 }
struct UDISetCurrentProcessRespMsg {
    UDIInt32       response_id;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-40.

UDIStdinMode

Message

```
struct UDIStdinModeRqMsg {
    UDIUInt32      service_id;
 }
struct UDIStdinModeRespMsg {
    UDIInt32       response_id;
    UDIUInt32      Mode;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-42.

UDIStep

Message

```
struct UDIStepRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      Steps;
    UDIUInt32      StepType;
    UDIRange       Range;
 }

struct UDIStepRespMsg {
    UDIInt32       response_id;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-43.

UDIWait

Message

```
struct UDIWaitRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      MaxTime;
 }

struct UDIWaitRespMsg {
    UDIInt32       response_id;
    UDIUInt32      PId;
    UDIUInt32      StopReason;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-46.

UDIWrite

Message

```
struct UDIWriteRqMsg {
    UDIUInt32             service_id;
    UDIResource           To;
    UDIUInt32             Count;
    UDIUInt32             Size;
    UDIUInt32             HostEndian;
    UDIByteArray          Buf;    /* dimension = Count*Size */
}

struct UDIWriteRespMsg {
    UDIInt32              response_id;
    UDIUInt32             CountDone;
    UDIUInt32             err;
 }
```

Description

The semantics of the parameters are described on page 3-50.

Notes:

If the *HostEndian* flag is 1, the 29K data in the *UDIWriteRqMsg.Buf* must be in big–endian format (keeping in mind that the size of each object in the buffer is specified by the *UDIWriteRqMsg.Size* parameter). Note that in this case, at the procedural interface this data is required to be in *HostEndian* format.

If the *HostEndian* flag is 0, the 29K data in the *UDIWriteRqMsg.Buf* must be in the same endian format as the target. It is assumed that both the DFE and the TIP know the endian type of the target. Note that in this case, the data in this buffer is unmodified by the IPC for the procedural interface.

UDIDFE Messages

The messages on the following pages are requests from the TIP to the DFE. All UDIDFE*xxx* requests can only be made by the TIP while it is in the middle of servicing some UDI Request from the DFE (i.e., before it has returned the UDI response). In addition, while the DFE is servicing the UDIDFE*xxx* request, it is possible for the DFE to make a new UDI request to the TIP. The message traffic in this last case would be:

-->UDI*xxx* request

<-- UDIDFE*yyy* request

--> UDI*zzz* request

<-- UDI*zzz* response

--> UDIDFE*yyy* response

<-- UDI*xxx* response

UDIDFEEndTIPIO

Message

```
struct UDIDFEEndTIPIORqMsg {
    UDIUInt32      service_id;
 }
struct UDIDFEEndTIPIORespMsg {
    UDIInt32       response_id;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-52.

UDIDFEEvalExpression

Message

```
struct UDIDFEEvalExpressionRqMsg {
    UDIUInt32      service_id;
    UDIString      Expression;
    UDIUInt32      BufSize;
 }
struct UDIDFEEvalExpressionRespMsg {
    UDIInt32       response_id;
    UDIInt  KindofAnswer;
    UDIResource    AnswerResource;
    UDIUInt32      CountDone;
    UDIUInt32      Size;
    UDIUInt32      Type;
    UDIByteArray   AnswerBuf;     /* dimension = 0 when
                                   * KindOfAnswer = Resource */
    UDIUInt32      Type;
    UDIByteArray   AnswerBuf;     /* dimension = 0 when
                                   * KindOfAnswer = Resource
                                   * dimension = CountDone*Size
when
                                  * KindOfAnswer = Value */
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-53.

Notes:

Although each response specifies (by *KindOfAnswer*) either an *AnswerResource* or an *AnswerBuf* with *CountDone*, *Size*, and *Type*, all components are sent in the IPC message. Thus, the IPC software need not inspect the value of *KindOfAnswer*.

Any data in *UDIDFEEvalExpressionRespMsg.AnswerBuf* consists of objects of size *UDIDFEEvalExpressionRespMsg.Size* and is always in big–endian format. Note that at the procedural interface this data is required to be in host–endian format (i.e. the data is in host byte order).

UDIDFEEvalResource

Message

```
struct UDIDFEEvalResourceRqMsg {
    UDIUInt32      service_id;
    UDIResource    ResourceToEval;
    UDIUInt32      ExactEval;
    UDIUInt32      BufSize;
 }
struct UDIDFEEvalExpressionRespMsg {
    UDIInt32       response_id;
    UDIString      SymbolAnswer;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-55.

UDIDFEGetInput

Message

```
struct UDIDFEGetInputRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      IOType;
    UDIUInt32      BufSize;
    UDIUInt32      Mode;
 }
struct UDIDFEGetInputRespMsg {
    UDIInt32       response_id;
    UDIUInt32      CountDone;
    UDIByteArray   Buf;           /* dimension = CountDone */
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-56.

UDIDFEPutOutput

Message

```
struct UDIDFEPutOutputRqMsg {
    UDIUInt32      service_id;
    UDIUInt32      IOType;
    UDIUInt32      Count;
    UDIByteArray   Buf;           /* dimension = Count */
}
struct UDIDFEPutOutputRespMsg {
    UDIInt32       response_id;
    UDIUInt32      CountDone;
    UDIUInt32      err;
 }
```

Description

The semantics of the parameters are described on page 3-58.

Chapter 6

UDI Developer's Toolkit

The UDI Developer's Toolkit provides source files, executables, and documentation that enables software tools developers for the 29K Family to develop and debug UDI–compliant DFEs or TIPs without having to be concerned with the details of the IPC methods. We assume that those using the toolkit are familiar with the general issues of developing debugger tools for the 29K Family and, in particular, are familiar with the UDI procedural interface, which is defined in Chapters 2 and 3 of this specification. Knowledge of the IPC methods in Chapters 4 and 5 is not required to use the UDI Developer's Toolkit.

This chapter describes the various pieces of the UDI Developer's Toolkit, with special emphasis on the sample IPC source code and its structure. The limitations of this sample IPC source code are described, as well as how to use the sample IPC source code in your own DFE or TIP. Finally, there is a discussion on how to use the utilities that are included with the toolkit, which can aid in the development of your DFE or TIP.

The UDI Developer's Toolkit consists of:

- The UDI Specification
- Sample IPC source code for:
  - UNIX Socket IPC method for big–endian DFEs and TIPs
  - DOS IPC method for real–mode DFEs and TIPs
  - DOS IPC method for protected–mode DFEs and TIPs
- The executables, source, makefile, and man pages for two UDI development tools: a TIP tester and a UDI trace utility.
- The complete source, makefiles, and executables for AMD's MiniMON29K product. The MiniMON29K product includes a UDI–compliant DFE and TIP so the source for these may be useful to other DFE and TIP developers.

- The executables for **isstip**. This is an instruction set simulator–based TIP normally distributed with AMD's High Cr 29Kt product. Since it is self–contained and does not require actual target hardware, it may be a convenient test TIP to connect to for DFE developers. Note that the source for **isstip** is not provided.

The UDI Procedural Interface and the Sample IPC Code

The most important part of the toolkit to UDI developers is the sample IPC source code. This sample IPC source code maps the UDI procedural interface (as defined in Chapters 2 and 3) to one of the two defined IPC methods: the DOS IPC method and the UNIX socket IPC method.

This chapter assumes that the DFE or TIP developer will write code using the UDI procedural interface and will then link with the sample IPC source code supplied in the toolkit. The advantages of this approach are:

- The procedural interface provides a host–independent interface to UDI.
- For testing and debugging purposes, developers can link a DFE and TIP together into one executable if both use the UDI procedural interface.
- Developers can save time by using the existing IPC implementation code provided in this toolkit since they will not have to be concerned with IPC details.

We refer to the IPC code in this toolkit as "sample" IPC code because the use of this code is in no way required to meet UDI interoperability. Developers could always write their own IPC code and even define a different procedural interface and it would be interoperable as long as it meets the IPC specifications described in Chapters 4 and 5 and the semantics described in Chapters 2 and 3. However, the sample IPC source code should meet the needs of most developers. The limitations of the sample IPC source code are described in the next section.

Limitations of the Sample IPC Code

The sample IPC source code has been built and tested in the following host development environments:

- UNIX socket IPC code
  - SunOS 4.1.x bundled C compiler

  - HP/UX bundled C compiler

- DOS IPC code, real–mode DFEs and TIPs
  - Microsoft C 7.0

- DOS IPC code, protected–mode DFEs and TIPs
  - MetaWare High C/r 386 3.1 compiler plus Phar Lap 4.0 linker, assembler, and DOS extender
  - Watcom 386 9.x compiler and linker plus Phar Lap 4.0 assembler and DOS extender

NOTE: The DOS IPC code for protected–mode DFEs and TIPs is dependent on the Phar Lap DOS extender.

If you plan to use a different host development environment and you find that modifications to the IPC source code are necessary to get it to build in that environment, please send e–mail to udi@amd.com and any such changes can be folded into the next version of the sources.

The sample IPC source code also has some limitations based on making some simplifying assumptions about the DFE or TIP:

- The UNIX socket IPC source code assumes that the host is big–endian. If you want to use this code on a little–endian host, you will have to swap the endian type of the appropriate fields when building the IPC messages. See the discussion in Chapter 5 on the endianness of fields in socket IPC messages.

- Although the IPC methods allow multiple TIPs per DFE and multiple DFEs per TIP, both the UNIX socket IPC source code and the DOS IPC source code assume that a DFE connects to, at most, one TIP and that a TIP gets connected to, at most, one DFE. On DOS IPC, the sender of a message always includes the proper *connection_id* in the message but the receiver of a message does not check the *connection_id*. On UNIX IPC, messages are always sent on the correct socket but DFEs and TIPs only listen on a single socket when listening for replies. Thus, DFEs and TIPs built with the sample IPC code should interoperate properly with TIPs and DFEs that do support multiple connections.

- When built with the DOS IPC source code, both the real–mode DFEs and TIPs and protected–mode DFEs and TIPs will interoperate in all four combinations. This is true for both plain MS–DOS and in a DOS window initiated from within Microsoft Windowst. However, a limitation of DPMI 0.90 (which is the protected mode interface implemented by Microsoft Windows) allows, at most, one DOS extender in a DOS window initiated from within Microsoft Windows. The sample code has a special work–around for this limitation if both the protected–mode DFE and protected–mode TIP are built with the Phar Lap DOS extender. We do not know of a solution when the protected–mode DFE and protected–mode TIP are built with two different DOS extenders.

- The DOS IPC code cannot be used to build DFEs and TIPs that are true Microsoft Windows applications.

Directory Structure of the Toolkit

src/udi

This directory contains the sample IPC source code for both the DOS IPC method and the UNIX socket IPC method. These include **.c** files, **.h** files, and a few **.asm** files (for the DOS IPC method). Developers building their own DFE or TIP would link with these **udi** files. See below for more detail on these files.

src/uditools

These are the source files used to build the TIP tester (**uditest**) and the UDI trace utility (**uditrace**). Developers building their own DFE or TIP would not link with these files, but may wish to use them for testing and debugging.

src/minimon

The directory structure of the MiniMON29K source tree is described in the MiniMON29K documentation. The discussion below describes only those parts of the tree which are relevant to UDI development.

src/minimon/host

MiniMON29K consists of target code and host code. Here we are concerned only with the host code. The host code builds two executables, **mondfe** and **montip**. The **mondfe** executable provides a monitor–level user interface and generates UDI requests. The **montip** executable basically takes UDI requests and maps them to MiniMON29K target messages and sends them to the target. The montip–to–target communication is not defined by UDI.

src/minimon/host/dfe

This directory contains the source files for building **mondfe**. Note that all UDI calls are made in the single module, **mini2udi.c**.

src/minimon/host/tip

These are the source files for building **montip**. In **montip**, all UDI calls are implemented in the single module, **udi2mtip.c**.

src/minimon/host/include

This directory contains the general **.h** files used by **mondfe** and **montip**. These **.h** files are independent of UDI.

bin_host

These are the executables for **mondfe**, **montip**, **isstip**, **uditest**, and **uditrace**. There is a **bin** directory for both **sun4** and **pc**.

lib

This directory contains the support files used by the above executables.

man

These are the man pages for **mondfe**, **montip**, **isstip**, **uditest**, and **uditrace**. The **udi(5)** man page, which describes the format of the TIP configuration file, is also included.

doc

This directory contains the UDI Specification (in Postscript format).

The Sample IPC Sources in src/udi

This section describes the individual files in the **udi** directory. The **src/udi** directory contains the include files that TIPs and DFEs use to define the UDI procedural interface, and it also includes the IPC implementation code for the two hosts, UNIX and DOS. Note that because the IPC mechanism under UDI is host–specific, many of the files in this directory are host–specific. In addition, some IPC files are used only with DFEs and others are used only with TIPs. The makefiles in **src/minimon/host** show how **mondfe** and **montip** make use of the various files in the **udi** directory. The makefiles in **src/uditools** also show how the **udi** files are used.

Files Common to all Hosts

udiproc.h

Defines the procedural interface for all of the UDI functions and defines all of the UDI data types. The **udiproc.h** file is the only include file that a DFE or TIP needs to include. The **udiproc.h** file is actually independent of the host and target architecture. The host–specific and target architecture–specific features are broken out into the subsidiary include files listed below. In the current implementation, two host types exist (UNIX and DOS) and one target architecture (29K).

udiphcfg.h

Included by **udiproc.h**. Includes either **udiphdos.h** or **udiphsun.h**.

udiptcfg.h

Included by **udiproc.h**. For now, always includes **udipt29k.h**.

udipt29k.h

Defines UDI data types specific to the 29K Family architecture.

udiids.h

Is optional and can be included by your DFE or TIP to help build the ID codes that are sent in the **UDICapabilities** call. The **udiids.h** file is described in more detail below.

udimapdf.c, udimapdf.h

This file is specific to DFEs. It is used for UDI services where the procedural interface has changed between UDI versions (for example, UDISetBreakpoint between 1.3 and 1.4). This file contains routines that map the newer versions of calls to the older versions of calls (or returns an error if mapping is not possible). This is needed when a newer DFE connects to an older TIP but still wants to be able to use the newer procedural interface.

udimaptp.c

This file is specific to TIPs and its use is optional. It is used for UDI services where the procedural interface has changed between UDI versions (for example, UDISetBreakpoint between 1.3 and 1.4). This file contains entry points for the older versions of such calls and maps them to the newer versions of the calls (or returns an error if mapping is not possible). The TIP procedural interface writer must provide implementations of the newer versions of the calls but can optionally either link with this file to get the older services or can write his own implementations of the older services.

Files Specific to DOS Hosts

udiphdos.h

Included by **udiproc.h**. Defines UDI data type sizes for DOS hosts.

udip2dos.c, dosdfe.asm

IPC code used by DFEs on all DOS hosts, both real and protected.

dos2udip.c, dostip.asm

IPC code used by TIPs on all DOS hosts, both real and protected.

udidos.h, udidos.ah

Include files used by IPC code on all DOS hosts, both real and protected.

d386cmnc.c, d386cmna.asm, realcopy.c

Additional IPC code used only by protected–mode (DOS386) DFEs and TIPs but common to both DFEs and TIPs.

d386cmnc.h, d386cmna.ah

Include files used by IPC code for protected–mode (DOS386) DFEs and TIPs but common to both DFEs and TIPs.

d386dfe.c, d386dfe.h

Additional IPC code used only by protected–mode (DOS386) DFEs.

The following summarizes the source files used by various DOS TIPs and DFEs:

Real–mode DFEs	udip2dos.c, dosdfe.asm, udimapdf.c
Real–mode TIPs	dos2udip.c, dostip.asm, udimaptp.c (optional)
Protected–mode DFEs	udip2dos.c, dosdfe.asm, d386cmnc.c, d386cmna.asm, realcopy.c, d386dfe.c, udimapdf.c
Protected–mode TIPs	dos2udip.c, dostip.asm, d386cmnc.c, d386cmna.asm, realcopy.c udimaptp.c (optional)

Files Specific to UNIX Hosts

udip2soc.c

IPC code for DFEs for UNIX hosts. Opens a socket to the TIP, builds messages which are then sent over that socket, and waits for response messages on the socket. If, instead of a response, a UDIDFE request is received, dispatches the routine, sends the response back to the TIP, and waits for the response to the original UDI call.

soc2udip.c

IPC code for TIPs for UNIX hosts. Listens on a socket for messages from the DFE and sends response messages back to the DFE. If the TIP makes a UDIDFE call to the DFE in the process of handling a UDI call, the TIP sends it to the DFE and waits for a response.

udr.c

IPC code used for both DFEs and TIPs on UNIX hosts. Contains routines for marshalling and unmarshalling various data types in the socket messages.

udisoc.h

Include file used by IPC layer on UNIX hosts.

udiphsun.h

Included by **udiproc.h**. Defines UDI data type sizes for Sun hosts.

The following summarizes the source files used by various UNIX TIPs and DFEs:

UNIX DFEs	udip2soc.c, udr.c, udimapdf.c
UNIX TIPs	soc2udip.c, udr.c, udimaptp.c (optional)

Product and Company Codes Used by UDICapabilities

AMD DFEs and TIPs use the **/src/mimimon/host/udi/udiids.h** file to generate the company, product, and version codes used in **UDICapabilities**. Listed below are the recommendations for the various fields used in the **UDICapabilities** calls:

Company Code in TIPId, DFEId, and TargetId

AMD will act as the central source of UDI company codes. Contact AMD via e–mail at udi@amd.com for your code (or codes if you need more than one).

Product Codes in TIPId, DFEId, and TargetId

Assigned by you.

Version Codes in TIPId, DFEId, and TargetId

Assigned by you.

DFE and TIP Parameters

Only the version numbers are significant in these parameters. These are UDI version numbers that the DFE or TIP can handle. The UDI specification describes how the TIP should inspect the input parameter, *DFE*, and supply the output parameter, *TIP*. If your DFE or TIP is building with the current toolkit, your version parameter should remain at 0x140 (UDI 1.4.0). This is the *UDILatestVersion* definition in **udiids.h**.

DFEIPCId and TIPIPCId Parameters

These consist of the normal company, product, and version fields. These are filled in by the IPC layers, and, in this toolkit, are defined in the IPC source files: **udip2dos.c**, **dos2udip.c**, **udip2soc.c**, and **soc2udip.c**. If you use the IPC sources from this toolkit unchanged, the IPCIds should be left unchanged as well. If you modify the IPC sources before you use them, please change the company code to your company's code, and change the IPCId version number and IPCId product codes for your own tracking purposes.

Notes for DFE Developers

As of UDI 1.3, a DFE is both a UDI client (making UDI calls) and a UDI server (implementing UDIDFE calls).

As a UDI client, a DFE may use whatever subset of the UDI services that it requires. Be aware that TIPs are not required to implement all of the UDI services, so a DFE should be written to be reasonably workable with a minimum TIP. See page 3-1 for a discussion of which UDI services a TIP is required to implement.

You will also find a list of which UDIDFE services a DFE is required to implement on page 3-1.

To make the TIP implementations somewhat simpler, there are some requirements made on the sequencing of UDI calls by the DFE. These are discussed elsewhere in this specification, but we list them here as a reminder:

- After calling **UDIGetErrorMessage**, the DFE must continue calling it until *Countdone* is less than *MsgSize*.
- After calling **UDIGetStdout** or **UDIGetStderr**, the DFE must continue calling it until *Countdone* is less than *BufSize*, unless the DFE calls **UDIStop**.

- The DFE must call **UDIWait** to ensure that execution actually takes place. This is true not only after **UDIExecute** and **UDIStep** but also after **UDIGetStdout** or **UDIGetStderr**. On some TIPs, execution may actually start before the call to **UDIWait**, but it is not required to.

- The DFE must call **UDIWait** after **UDIStop** to ensure that execution has really stopped.

The toolkit does not contain a test program that tests a DFE for UDI compliance. You can, however, use the **isstip** (an Instruction Set Simulator TIP which ships with AMD's High C 29K product) to test your DFE's behavior. **isstip** is easy to use because it is self–contained and requires no real target hardware. For further testing, the TIP from the MiniMON29K product, **montip** can be ordered from AMD.

If your DFE does not behave as expected when connected to any TIP, the **uditrace** tool can be used to show all the UDI and UDIDFE calls going back and forth between the DFE and the TIP. The **uditrace** tool can also be useful in pointing out inefficiencies, for example, a DFE that uses a separate UDI call to read each register. See the **uditrace(1)** man page for instructions on how to use **uditrace**.

Notes for TIP Developers

As of UDI 1.3, a TIP is both a UDI server (implementing UDI calls) and a UDI client (making UDIDFE calls).

See page 3-1for a discussion of which of the UDI services a TIP is required to implement.

The **uditest** tool can be used as a DFE to test your TIP. See the **uditest(1)** man page for a description of how to run **uditest**. The **uditest** tool is mostly silent unless it encounters an error. If **uditest** does report an error and the error message does not fully explain the error, you should probably use the **uditrace** tool (placing **uditrace** between **uditest** and your TIP) to show exactly what UDI call caused the problem. The **uditrace** tool can be used to diagnose the traffic between any DFE and your TIP. See the **uditrace(1)** man page for instructions on using **uditrace**.

The source code for **uditest** is also provided and can be used to more fully understand the action that caused the error. If you have suggestions for further tests that should be added to **uditest.c**, please send those suggestions via e–mail to udi@amd.com.

Notes for DOS Development

You can use the UDI Toolkit to develop both real–mode and protected–mode DFEs and TIPs for PC hosts.

Real–Mode DFEs and TIPs

The targets, **dosdfe/smalldfe.exe** and **dostip/smalltip.exe**, under **src/uditools/makefile.pc** can be used as templates for real–mode DFE and TIP development on PC hosts. Further examples are available in the **minimon** tree in the **src/minimon/host/makepc.bat** file.

All the sources provided in this toolkit have been compiled on DOS using the Microsoft C 7.0 compiler.

The memory model of the DFE is not critical since the IPC code will force far calls with far pointer parameters to the TIP. The DFE stack is used for the duration of a UDI call so you may want to increase the DFE stack size.

The memory model of the TIP must be large model. Also, because the TIP procedures are basically called using the DFE's stack pointer, the model flags **-Alfu** (DS loaded on procedure entry; DS != SS) must be used.

The TIP's stack should be as small as possible (it is not used once the TIP becomes resident). Note also the use of the /CP:1 parameter in the link command line for **montip**. This greatly reduces the footprint of the TIP.

When the DFE disconnects from the TIP with the **UDITerminateSession** parameter, you should find that the TIP has been removed from memory by the IPC layer. If it has not been removed, either the DFE did not call disconnect or the TIP returned an error on **UDIDisconnect**.

Protected–Mode DFEs and TIPs

The toolkit can also be used to build 386 protected–mode DFEs and TIPs. Real–mode and protected–mode DFEs and TIPs all use the same real–mode DOS IPC method and so all combinations can intercommunicate. In addition, both the real–mode and the protected– mode DFEs and TIPs can work in a DOS window initiated from within Microsoft Windows 3.0 or later.

The targets, **dos386/smalldfe.exe** and **dos386/smalltip.exe**, under **src/uditools/makefile.pc** can be used as templates for protected–mode DFE and TIP development on PC hosts. The support assumes the use of the MetaWare High C 386 (or Watcom compiler) and Phar Lap DOS extender. The **isstip.exe** executable (which requires access to large amounts of memory at runtime to simulate 29K memory) has, in fact, been built using this method.

To build a 386 protected–mode TIP:

- Compile your own TIP code with the MetaWare High C 386 compiler.
- Similarly compile **dos2udip.c**, **d386cmn.c**, and **realcopy.c** with the MetaWare High C 386 ompiler, including:

```
DMSDOS -DDOS386
```

on the command line.

- Assemble **dostip.asm** and **d386cmna.asm** with the Phar Lap 386 assembler with the command line:

```
386asm -DDOS386 xxx.asm
```

Link all the above **.objs** using the Phar Lap 386 linker command file shown in **src/uditools/makefile.pc**. You should assume that all the linker directives shown in that example are important. It is also required that the sections defined in the **.asm** files, **dostip.asm**, and **d386cmna.asm**, appear in the first 64 K of the final link. This can be done by placing these **.objs** first in the link order.

To build a 386 protected–mode DFE:

- Compile your own DFE code with the MetaWare High C compiler.
- Similarly compile **udip2dos.c**, **d386dfe.c**, **d386cmn.c**, and **realcopy.c** with the MetaWare High C compiler, including:

```
DMSDOS -DDOS386
```

on the command line.

- Assemble **dosdfe.asm** and **d386cmna.asm** with the Phar Lap 386 assembler with the command line:

```
386asm -DDOS386 xxx.asm
```

Link all the above **.objs** using the Phar Lap 386 linker command file shown in **src/uditools/makefile.pc**. You should assume that all the linker directives shown in that example are important. It is also required that the sections defined in the **.asm** files, **dosdfe.asm** and **d386cmna.asm**, appear in the first 64K of the final link. This can be done by placing these **.objs** first in the link order.

Notes for UNIX Development

The makefiles for **smalldfe** and **smalltip** under **src/uditools** can be used as templates for DFE and TIP development on UNIX hosts. Further examples are available as the target's **mondfe** and **montip** in **src/minimon/host/makefile**. Also note that the target "minimon" in **src/minimon/host/makefile** is an example of the DFE and TIP code linked together to form a single executable (for example, for testing purposes).

All the sources in this toolkit have been compiled using the bundled cc compiler from SunOS 4.1. or from HP/UX version 9.01.

When the DFE disconnects from the TIP with the parameter, **UDITerminateSession**, you should find that the TIP process has been killed and the socket file has been deleted. If this does not happen, either the DFE did not call disconnect or the TIP returned an error on **UDIDisconnect**. Also, if the TIP remains alive and the socket file has not been deleted, then an attempt to connect to that TIP configuration again will usually hang. In such a case, kill the TIP and manually delete the socket file to recover.

Appendix A

UDI Error Numbers

In general, each service accepts a number of parameters and returns a UDI error code. UDI error codes other than the value **UDINoError** (which has the value 0) indicate an error has occurred. If the error code is negative, it is a TIP–specific error that is not documented as a UDI error in general. Such error codes can be passed to **UDIGetErrorMessage()** to retrieve a textual representation of the error. Positive UDI error codes are defined here.

Number	Error Name	Description
0	**UDINoError**	No errors have occurred.
1	**UDIErrorNoSuchConfiguration**	Indicates that the requested configuration is not present in the configuration file. Returned only from **UDIConnect**().
2	**UDIErrorCantHappen**	Indicates the existence of some condition internal to the IPC Layer that, theoretically, can't happen. This error should never occur.
3	**UDIErrorCantConnect**	The IPC Layer is incapable of supporting the connection. This may occur, for example, if the IPC Layer imposes a limit on the number of concurrent connections that can be supported and the limit has been reached. This error is returned only when a IPC–imposed limitation is reached.
4	**UDIErrorNoSuchConnection**	Indicates that the Session parameter is invalid. It can be returned only from functions that accept a Session parameter.
5	**UDIErrorNoConnection**	Is returned from functions that require a pre–established connection (virtually all UDI functions), if no connection is currently established. This error, then, can occur only before a **UDIConnect** call, or after a **UDIDisconnect** that has closed the current connection.

6	**UDIErrorCantOpenConfigFile**	The IPC Layer associates a configuration name (the first parameter to **UDIConnect**) with a TIP and its parameters by means of a configuration file. The IPC Layer was unable to open the file. This error can occur only during a **UDIConnect** call.
7	**UDIErrorCantStartTIP**	The host operating system has failed to start the TIP at the IPC Layer's request. The reason the OS failed is not indicated. Generally, the OS fails because of resource limitations or a failure to find the TIP's executable file executable. This error can occur only during a **UDIConnect** call.
8	**UDIErrorConnectionUnavailable**	This error is returned from a TIP that should connect, but cannot because the requested target is already in use. It is returned only by **UDIConnect**.
9	**UDIErrorTryAnotherTIP**	Returned by a TIP in response to a **UDIConnect** request if the TIP cannot service the UDIConnect, but does not know whether another running TIP is able to service it. This error should never be seen by the DFE.
10	**UDIErrorExecutableNotTIP**	The executable file identified in the configuration file entry for the requested configuration is not a TIP. This error can occur only during a **UDIConnect** call.
11	**UDIErrorInvalidTIPOption**	The configuration passed to a **UDIConnect** call has options identified in the configuration file that the associated TIP finds invalid. This can be simply an unrecognized option or a combination of options that is inconsistent, for example requesting a simulation of the Am29000 and the Am29050 simultaneously.
12	**UDIErrorCantDisconnect**	Indicates that the TIP is incapable of disconnecting at this time; for example, the TIP is temporarily unable to notify other processes of the disconnection. The error can occur only during UDIDisconnect calls.
13	**UDIErrorUnknownError**	Is returned from **UDIGetErrorMsg** if the

		passed–in error code is unknown.
14	**UDIErrorCantCreateProcess**	Indicates a problem honoring a **UDICreateProcess** request.
15	**UDIErrorNoSuchProcess**	If a call is made to a UDI function expecting a PId, but the passed PId is not valid, **UDIErrorNoSuchProcess** is returned.
16	**UDIErrorUnknownResourceSpace**	Any UDI function expecting a **UDIResource** can return this error if the resource's Space member is not known to the callee.
17	**UDIErrorInvalidResource**	TIPs may enforce restrictions on valid offsets within resource spaces. This error is returned by such a TIP if an offset is invalid within the associated space. Any UDI function expecting a **UDIResource** can return this error.
18	**UDIErrorUnsupportedStepType**	A TIP that does not support a StepType requested in a **UDIStep** request returns this error.
19	**UDIErrorCantSetBreakpoint**	A TIP that cannot honor a **UDISetBreakpoint** request returns this error. The error can be caused by resource limitations at the TIP, the inability to set certain types of breakpoints, or an invalid parameter.
20	**UDIErrorTooManyBreakpoints**	**UDISetBreakpoint** returns this error if the breakpoint cannot be set because a limit on the number of breakpoints the TIP supports has been reached.
21	**UDIErrorInvalidBreakId**	Functions expecting a BreakId return this error if the passed BreakId is invalid.
22	**UDIErrorNoMoreBreakIds**	**UDIQueryBreakpoint** returns this error code instead of **UDIErrorInvalidBreakId** if the BreakId queried is not only invalid, but also greater than any valid BreakId.
23	**UDIErrorUnsupportedService**	Since not all UDI functions must be implemented in each TIP and not all UDIDFE functions must be implemented in each DFE, a request to perform an unsupported service draws this error if the callee is incapable of

		the requested service.
24	**UDIErrorTryAgain**	This error code is returned when a temporary condition prevents honoring a request. If the same condition persists for more than five attempts, the TIP assumes the condition is not temporary and returns a different error indication.
25	**UDIErrorIPCLimitation**	Some IPCs impose limitations of their own. For example, early socket IPC implementations did not support read or write requests greater than a specific size. When the IPC imposes a limitation that a request exceeds, the IPC returns this error.
26	**UDIErrorIncomplete**	If a data movement operation requested by the DFE cannot be completely satisfied, the TIP returns **UDIErrorIncomplete**. Each call where this error is possible supports a CountDone parameter that provides the DFE with information about how much of the request was completed.
27	**UDIErrorAborted**	If a data movement operation requested by the DFE is aborted by the DFE (by calling UDIStop), the TIP returns **UDIErrorAborted**. Each call where this error is possible supports a CountDone parameter that provides the DFE with information about how much of the request was completed.
28	**UDIErrorTransDone**	When **UDIGetTrans** is called, if the TIP is in a position to allow the DFE to resume normal UDI operations, the TIP returns **UDIErrorTransDone**. The DFE can either leave transparent mode by calling any other UDI service or it can remain in transparent mode by calling either **UDIGetTrans** or **UDIPutTrans**. The TIP continues to return **UDIErrorTransDone** in response to **UDIGetTrans** calls subsequent to any one that returned **UDIErrorTransDone** if no other UDI service has been called.

29	**UDIErrorCantAccept**	**UDIPutStdin** and **UDIPutTrans** returns **UDICantAccept** if the TIP cannot accept more data from the DFE. The data sent with the **UDIPutStdin** and **UDIPutTrans** requests that received the error can be complete, partially complete, or ignored. The CountDone parameter indicates exactly how much data was taken.
30	**UDIErrorTransInputNeeded**	Returned by **UDIGetTrans** when the TIP requires transparent mode input from the DFE.
31	**UDIErrorTransModeX**	Returned by **UDIGetTrans** when the TIP requests a change of terminal operating mode.
32	**UDIErrorInvalidSize**	Returned by **UDIRead**, **UDIWrite**, **UDICopy**, and **UDIFind** if the TIP is unable to perform the requested operation because the object size does not correlate with the resource specified.
33	**UDIErrorBadConfigFileEntry**	Returned by **UDIConnect** when the configuration file line for the requested configuration is incorrect.
34	**UDIErrorIPCInternal**	Returned by any call if the IPC Layer detects an internal error during an operation other than **UDIConnect**.
35	**UDIErrorUnsupportedService Variation**	Indicates that a service is supported but the particular variation of that service being requested is not supported.
36	**UDIErrorResourceNotWriteable**	Some resource spaces are read–only. TIPs may return this error from any request that tries to write into a read–only resource (e.g., **UDIWrite** or **UDICopy**).
37	**UDIErrorCouldNotEvaluate**	Returned by **UDIDFEEvalExpression** if the supplied expression could not be evaluated.
38	**UDIErrorNoExactEval**	Returned by **UDIDFEEvalResource** if ExactEval was true but the resource did not match any symbol exactly.
39	**UDIErrorBufTooSmall**	Returned by **UDIDFEEvalResource** if the

		resource could be evaluated but the buffer supplied by the TIP was not large enough to hold the string.
40	**UDIErrorEvaluatedToValue**	Returned by **UDIDFEEvalExpression** if the supplied expression evaluated to a Value but AnswerBuf was a NULL pointer.
41	**UDIErrorEvaluatedToResource**	Returned by **UDIDFEEvalExpression** if the supplied expression evaluated to a Resource (address) but AnswerResource was a NULL pointer.
42	**UDIErrorResetAsserted**	The TIP could not satisfy the request because Reset was asserted on the target.
43	**UDIErrorNoPower**	The TIP could not satisfy the request because the target had no power.
44	**UDIErrorNoClock**	The TIP could not satisfy the request because the target had no clock.
45	**UDIErrorTargetNotResponding**	The TIP could not satisfy the request because the target was not responding.
46	**UDIErrorTargetAlreadyRunning**	The TIP could not satisfy the request because it was already running.

Appendix B

UDI Configuration Files

A UDI–conformant debugger front end (DFE) specifies the target interface process (TIP) to connect to, and the options to pass to that TIP, by referencing a configuration in the UDI configuration file. This appendix explains the format of the UDI configuration files for MS–DOS and UNIX hosts.

UDI Configuration Files for MS–DOS Hosts

The DFE searches in the following order to locate the UDI configuration file:

The complete filename specified by the environment variable, **UDICONF**.

The **udiconfs.txt** file in the current directory.

The **udiconfs.txt** file in the directory where the DFE is located.

The **udiconfs.txt** file in each of the directories specified by the **PATH** environment variable.

Each line of the **udiconfs.txt** file consists of the following fields:

```
tip_config_name tip_executable [tip_options]
```

where:

tip_config_name

Is an arbitrary name which the DFE will use to refer to this configuration. Each line in the **udiconfs.txt** file must have a unique *tip_config_name* field.

tip_executable

Is the name of the TIP executable file. The DFE will use the *tip_executable* filename to create the TIP if the TIP is not already running. If a full pathname is not specified, the **PATH** environment variable is used to locate the executable file.

tip_options

Are the options for the TIP being used. The rest of the line after the *tip_executable* name is passed to the TIP at connect time.

The following is an example of an entry in the UDI configuration file for MS–DOS hosts.

Example

```
ser38400 montip -t serial -baud 38400
```

In the above example, **ser38400** is the name chosen to refer to the configuration. Use this configuration name whenever invoking this configuration from a DFE. The user could have chosen any name for this configuration.

The TIP executable is **montip**. The user's **PATH** variable will be searched to find the **montip** executable.

The options "-t serial -baud 38400" are passed to **montip** when it is invoked.

UDI Configuration Files for UNIX Hosts

The DFE searches in the following order to locate the UDI configuration file:

The complete filename specified by the environment variable, **UDICONF**.

The **udi_soc** file in the current directory.

The **udi_soc** file in the directory where the DFE is located.

The **udi_soc** file in each of the directories specified by the **PATH** environment variable.

A line of **udi_soc** can have two different formats depending on whether the address family is AF_UNIX or AF_INET. The two formats are as follows:

```
tip_config_name AF_UNIX socket_name tip_executable [tip_options]

tip_config_name AF_INET host_name port_number [tip_options]
```

where:

tip_config_name

Is an arbitrary name which the DFE will use to refer to this configuration. Each line in the UDI configuration file must have a different *tip_config_name* field.

AF_UNIX This address family should be used when the DFE and TIP are running on the same host. This is the typical case.

AF_INET This address family should be used when the DFE and TIP are running on different hosts.

socket_name

Used only with AF_UNIX configurations. Specifies the socket filename that will be used to communicate between the DFE and TIP. The special socket name * indicates that a unique socket filename should be generated automatically by the IPC layer. This is useful if the user wants to have multiple DFEs connecting to the same configuration name. If the *socket_name* is specified explicitly, be aware that if any two AF_UNIX TIP configurations are being used simultaneously, they must have unique *socket_names*. For DFEs that want to disconnect from a TIP and then reconnect to that same TIP at some later time, an explicit *socket_name* is required.

tip_executable

Used only with AF_UNIX configurations. The DFE will use the *tip_executable* filename to spawn the TIP if the TIP is not already running and listening at the indicated *socket_name*. If a full pathname is not specified, the **PATH** environment variable is used to locate the executable file. Note that when the *socket_name* is *****, a new TIP executable is always created.

host_name Used only with AF_INET configurations. This specifies the name of the host where the TIP is running.

port_number

Used only with AF_INET configurations. This specifies the *port_number* at which the TIP on the remote host is listening. Note that in an AF_INET configuration, the TIP cannot be created by the DFE and must already be running at the time of the connection. The TIP on the remote host should be started with a command line of:

```
tip_executable_name AF_INET port_number
```

tip_options Valid with both AF_UNIX and AF_INET configurations. This optional string of parameters is passed through to the TIP at connect time and is usually interpreted by the TIP as a set of startup parameters.

The following are examples of entries in the UDI Configuration file on UNIX hosts.

Example

```
ser38400 AF_UNIX * montip -t serial -baud 38400
```

In the above example, **ser38400** is the name chosen to refer to the configuration. Use this configuration name whenever invoking this configuration from a DFE. The user could have chosen any name for this configuration.

AF_UNIX is the address family for the socket used to communicate between the DFE and the TIP. The ***** indicates that a unique socket name will be chosen for this connection. See the UDI man page for more information on address family and socket name options.

The TIP executable is **montip** (the TIP shipped with AMD's MiniMON29K product). The user's **PATH** variable will be searched to find the **montip** executable.

The options "-t serial -baud 38400" are passed to **montip** when it is invoked.

Example

```
iss50_remote AF_INET fasthost 7000 -29050 -r osboot
```

The above entry assumes that some TIP is already running on the host named **fasthost** and listening at port 7000. For example, **isstip** could have been started on **fasthost** with the command line:

```
isstip AF_INET 7000
```

The parameters "-29050 -r osboot" will be passed to the remote TIP at connect time.

Appendix C

29K Family UDI Resource Spaces

This appendix describes the resource spaces that are predefined when UDI is applied to targets using the AMD 29K Family of microprocessors. These definitions are used in the *Space* field of any **UDIResource** (for example, the *To* parameter in **UDIWrite**, the *From* parameter in **UDIRead**, etc.). Each space is defined along with the meaning of the offsets within the space. In the sample IPC sources from AMD, these spaces are defined in the **udipt29k.h** file.

Remember also that negative values of **UDIResource.Space** can be used for vendor–specific definitions for resources that are not covered by these predefined resource spaces.

```
#define UDI29KDRAMSpace                 0
```

Data RAM space.
Offsets are from bytes 0 to 4G.

```
#define UDI29KIOSpace                   1
```

I/O address space (not implemented on all 29K Family members).
Offsets are from bytes 0 to 4G.

```
#define UDI29KCPSpace0                  2
```

CoprocessorSpace 0 (not implemented on all 29K Family members).
Offsets are from bytes 0 to 4G.

```
#define UDI29KCPSpace1                  3
```

CoprocessorSpace 1 (not implemented on all 29K Family members).
Offsets are from bytes 0 to 4G.

```
#define UDI29KIROMSpace                 4
```

On 29K family members that have an RE bit, this is the instruction ROM space. Offsets are from bytes 0 to 4G.

```
#define UDI29KIRAMSpace                 5
```

On 29K Family members that have an RE bit, this is the instruction RAM space. Offsets are from bytes 0 to 4G.

```
#defineUDI29KLocalRegs                  8
```

Local registers (each 32 bits in size).
Offset 0 = LR0
Offset 127 = LR127

```
#defineUDI29KGlobalRegs                 9
```

Global registers (each 32 bits in size).
Offset 0 = GR0
Offset 127 = GR127
(Offsets 2 through 63 are not implemented on all 29K Family members.)

```
#define UDI29KRealRegs                  10
```

Real registers (each 32 bits in size).
Offset 0 = GR0
Offset 127 = GR127
Offset 128 = absolute register 128
Offset 255 = absolute register 255
128 through 255 map to local registers depending on the value of GR1.
(Offsets 2 through 63 are not implemented on all 29K Family members.)

```
#define UDI29KSpecialRegs               11
```

Special registers.
Offset 0 = SR0
Offset 255 = SR255
Note: A particular special register may not be implemented on every 29K Family member.

```
#define UDI29KTLBRegs                   12
```

Offset 0 = TLB0
Offset 255 = TLB255
(Not implemented on all 29K Family members.)

```
#define UDI29KACCRegs                   13
```

Accumulator registers.
Offset 0 = ACC0
Offset 3 = ACC3
(Implemented only on the Am29050 microprocessor.)

```
#define UDI29KICacheSpace               14
```

Instruction cache space.
Not available on all 29K Family members. These reference the cache entries (as opposed to the tag words). Offsets are byte addressable (but will almost always be read and written as words). For each family member, the sets are ordered and the first word of set s follows the last word of set s-1. Thus, assuming sets have size N:

Offset 0 = first byte, first word, first set

Offset 4 = first byte, second word, first set

Offset N-1 = last byte, last word, first set

Offset N = first byte, first word, second set

Offset 2*N-1 = last byte, last word, second set, etc.

```
#define UDI29KAm29027Regs         15      /* When available */
#define UDI29KPC                  16
```

Offset 0 = Real PC1
Offset 1 = Resource space of Real PC1
Offset 2 = Real PC0

NOTE: On a **UDIWrite** to **UDI29KPC** space, if Offset 0 (Real PC1) is specified, but Offset 2 (Real PC0) is not specified, then the TIP will set
Real PC0 = Real PC1 + 4.

```
#define UDI29KDCacheSpace         17
```

Data cache space.
Not available on all 29K Family members. These reference the cache entries (as opposed to the tag words). Offsets are as defined for instruction cache.

```
#define UDI29KICacheTagsSpace             18      /* When available
*/
```

Instruction cache tags.
Used together with **UDI29KICacheSpace**. Size is always 32 bits. Assuming sets have M blocks with each block having 1 tag, the offsets are:

Offset 0 = first tag, first set

Offset 1 = second tag, first set

Offset M-1 = last tag, first set

Offset M = first tag, second set

Offset 2*M-1 = last tag, second set, etc.

```
#define UDI29KDCacheTagsSpace            19      /* When available
*/
```

Data cache tags.
Used together with **UDI29KDCacheSpace**. Size is always 32 bits. Offsets are as defined under **UDI29KICacheTagsSpace**.

Appendix D

Compatibility of UDI 1.4, 1.3, and 1.2 DFEs and TIPs

At the IPC level, UDI 1.4 TIPs can recognize a UDI 1.2 or UDI 1.3 DFE because the version part of the incoming parameter, *Connect.DFEIPCId*, will indicate an earlier version of UDI. For the special case of UDI 1.2 DOS IPC, UDI 1.2 DFEs actually called a different procedure in the call table and can be recognized that way.

UDI 1.4 DFEs can recognize a UDI 1.2 or UDI 1.3 TIP because the version part of the returned parameter, *Connect.TIPIPCId*, will indicate an earlier version of UDI. For the special case of UDI 1.2 DOS IPC, UDI 1.2 DFEs recognize a 1.2 TIP because of the absence of the **Signature_13** field in the **TIPVecRec**.

UDI 1.4 DFEs and TIPs can interoperate with UDI 1.2 and UDI 1.3 DFEs and TIPs if the following precautions are taken:

- DFEs talking to 1.3 TIPs:
  - Can only use offset 0 (Real PC1) in **UDI29KPC** space.
  - For the UNIX Socket IPC, all host–endian fields are sent big endian (even when both the DFE and TIP are little endian).
  - UDISetBreakpoint_14, UDIQueryBreakpoint_14 and UDIConnect_14 calls cannot be used. If used at the procedural interface of the AMD sample implementation, the IPC will attempt to map to the corresponding _13 call, returning an error if unable to map.
- DFEs talking to 1.2 TIPs have all the above 1.3 restrictions, plus:
  - Cannot use **UDIFind** (newly introduced at UDI 1.3).
  - Will get back, at most, 80 characters in *TIPString* of **UDICapabilities**.
  - For the DOS IPC, the DFE cannot use any of the functions below the *Signature_13* field. The DFE must use the **UDIConnect_12** and the **UDICapabilities_12** table entries rather than their 1.3 and later entries.

- For the DOS IPC, 1.2 TIPs did not support multiple connections. They will ignore the *connection_id* field except on the **UDIDisconnect** call.
- For the UNIX Socket IPC, in a response message from the 1.2 TIP, the *response_code* field will not be present. Since the 1.2 TIP will not make any **UDIDFE***xxx* requests, the DFE knows the first message that comes back on the socket is the response.
- For the UNIX Socket IPC, the **UDIFind** and **UDICapabilities_13** requests cannot be used.
- For the UNIX Socket IPC, 1.2 TIPs do not support multiple connections.

- TIPs talking to 1.3 DFEs:
  - For the UNIX Socket IPC, all host–endian fields are sent big endian (even when both the DFE and TIP are little endian).
  - Cannot pass a fine state in the **UDIWait**.*StopReason* when the gross state is **UDIRunning** or **UDIHalted**. Cannot use negative fine states.
  - Only UDI error numbers through 41 can be returned.
- TIPs talking to 1.2 DFEs have all the above 1.3 restrictions, plus:
  - Cannot use any of the **UDIDFE***xxx* calls.
  - Can return, at most, 80 characters in *TIPString* of **UDICapabilities**.
  - Only UDI error numbers through 34 can be returned.
  - For the DOS IPC, only one 1.2 DFE connection should be allowed. The *connection_id* parameter from a 1.2 DFE is not valid (except on **UDIDisconnect**) and must be ignored on other calls.
  - For the DOS IPC, the 1.2 DFE will call **UDIConnect_12** and the **UDICapabilities_12** table entries rather than their 1.3 and later entries.
  - For the UNIX Socket IPC, the *response_code* field must be omitted in all response messages back to the DFE. Since you cannot make any UDI DFE requests, the DFE knows the first message that comes back on the socket is the response.

- For the UNIX Socket IPC, the DFE will make **UDICapabilities_12** requests rather than **UDICapabilities_13** requests.