Host Interface (HIF) Specification

Version 2.0

Host Interface (HIF) Specification, Version 2.0

The text pages of this document have been printed on recycled paper consisting of 50% recycled fiber and virgin fiber; the post-consumer waste content is 10%. These pages are recyclable.

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

Contents

About This Specification

Chapter 1

Introduction

Chapter 2

System Call Mechanism

Chapter 3

HIF Service Routines

Chapter 4

Process Environment

Appendix A

HIF Quick Reference

Appendix B

HIF Error Numbers

Index

Figures and Tables

Figures

Tables

About This Specification

The Host Interface (HIF) is the software specification that defines the standard set of kernel services that interface a user-application program to a host operating system. HIF currently provides the interface between the user's high-level language program and products such as the Advanced Micro Devices (AMD®) 29K™ Processor Architectural Simulator, PC Execution Boards (EB29K™ development tool, EB29030™ add-in board, and more), and Standalone Demonstration and Execution Boards (SA-29200™, SA-29240™, SD-29240™, EZ-030, and more).

End-users include the following:

- Those using AMD-supplied hardware execution vehicles or simulators
- Those developing a custom kernel operating system for a 29K Family processor design
- Those who are using the AMD-supplied high-level language development tools, but who must conform to another kernel operating system interface

How to Use This Documentation

About This Specification

The contents of each chapter and appendix of this document are described below:

- Chapter 1: “Introduction” discusses the important concepts underlying the host interface definition.
- Chapter 2: “System Call Mechanism” describes the mechanism used to make calls on the HIF services, and includes information on register usage for passing parameters and receiving results.
- Chapter 3: “HIF Service Routines” lists the services defined in HIF, and then describes each of the services and shows details of the code sequences, including examples, for invoking the services.
- Chapter 4: “Process Environment” describes the standard memory allocation and register initializations performed by the HIF-conforming kernel prior to execution of a user program.
- Appendix A: “HIF Quick Reference” lists all of the services and service parameters used in this document, in quick reference form.
- Appendix B: “HIF Error Numbers” lists the error codes that HIF-conforming services may return.

Intended Audience

This has been written for systems designers and programmers with a strong working knowledge of the 29K Family and their supporting peripheral hardware. This specification does not cover CPU design, the processor instruction sets, or any other hardware details.

Reference Documents

The following AMD documents may be of interest:

- *Am29000™ and Am29005™ User's Manual and Data Sheet*
  Advanced Micro Devices, order number 16914A.
- *Am29030™ and Am29035™ Microprocessors User's Manual and Data Sheet*
  Advanced Micro Devices, order number 15723B
- *Am29050™ Microprocessor User's Manual*
  Advanced Micro Devices, order number 14778A
- *Am29050™ Data Sheet*
  Advanced Micro Devices, order number 15039A.
- *Am29200™ RISC Microcontroller User's Manual and Data Sheet*
  Advanced Micro Devices, order number 16362B
- *Am29205™ RISC Microcontroller Data Sheet*
  Advanced Micro Devices, order number 17198A
- *Am29240™, Am29245™, and Am29243™ RISC Microcontrollers User's Manual and Data Sheet*
  Advanced Micro Devices, order number 17741A
- *Processor Initialization and Run-Time Services: OSBOOT*
  Advanced Micro Devices, order number 18275A
- *Programming the 29K™ RISC Family*
  by Daniel Mann, P T R Prentice-Hall, Inc. 1994
- *Universal Debugger Interface (UDI) Specification*
  Advanced Micro Devices, order number 18276A

Documentation Conventions

This specification assumes some familiarity with the UNIX® operating system and the C language. In this specification, the conventions presented in the sections below are assumed.

Numeric Values

All numeric values are presumed to be expressed in decimal notation unless otherwise stated. Hexadecimal values are prefaced by the characters **0x**. Any value not prefaced by **0x** is defined to be a decimal number. For example:

`100092`	Decimal number
`0x100092`	Hexadecimal number

The first number above is a decimal value by implication, because it has not been prefaced by **0x**. The second constant includes the explicit **0x** prefix, designating it as a hexadecimal value.

Character Strings

In the documentation, frequent mention is made of character strings that hold filenames, pathnames, and environment variable names. In all cases, the HIF Specification requires that strings be constructed as a sequence of ASCII characters terminated by a NULL byte (an 8-bit character composed of all zero bits). This is the form in which strings are represented in the C language. Thus, the space reserved for a string must be one byte longer than the length of the string, to accommodate the NULL byte.

Languages such as Pascal, which require counted strings (that is, a single 8-bit byte in the first character of the string that specifies the number of bytes that follow), are required to convert these to NULL-terminated form before calling the HIF kernel services. In addition, languages other than C may need to convert strings passed back from the HIF kernel services to a compatible internal form. All returned strings are in NULL-terminated form.

Chapter 1

Introduction

Advanced Micro Devices is developing a complete line of 29K processor simulators, hardware target execution vehicles, and high-level language development tools for the 29K Family of 32-bit RISC microprocessors. These products are designed to support end-users who are building embedded system applications based on a 29K Family processor. For these users, often there is no existing operating system or kernel for their hardware design.

Before AMD could create development tools for the 29K processors, a standard set of kernel services had to be defined that would interface a user-application program, written in a high-level language, to a host operating system or any one of the 29K Family of processors.

The Host Interface (HIF) is the software specification that defines this standard set of kernel services. Figure 1–1 shows the level where HIF resides. As implied by the figure, HIF does not describe any particular implementation; but rather each simulator, hardware vehicle, and high-level language implements HIF in its own way. The kernel services provide the minimum functionality needed to interface high-level language library functions to the user's operating system code.

Using HIF, program modules written in any of the languages available for the 29K processor can be combined, and the resulting program can run, without change, on any 29K processor simulator or hardware execution vehicle. Future AMD products will also use HIF, and AMD is actively encouraging third-party vendor support.

AMD is indebted to Embedded Performance, Incorporated (EPI), who originally developed the HIF concepts and then graciously made them available.

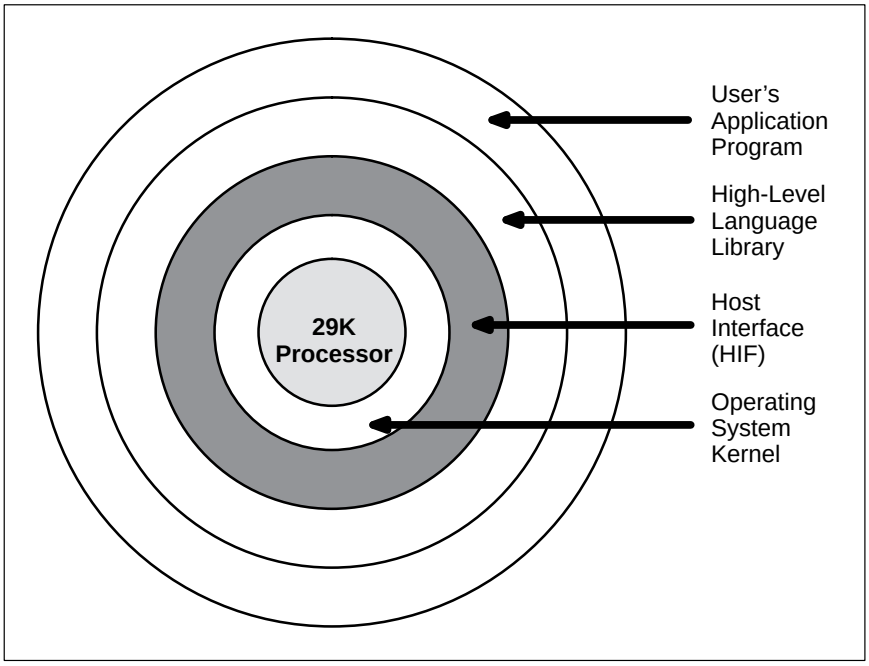

Figure 1–1. HIF Interface

HIF Applications

The HIF specification has broad applications; it provides the interface between the user's high-level language program and many hardware and software products. Some of the hardware and software products supported are as follows:

- *29K Processor Architectural Simulator.* This software product provides the means to simulate the operation of the 29K Family processor in a specified system environment. It provides detailed performance statistics by modeling the internal architecture of the 29K processors, as well as system memory configurations and timing. The HIF specification is implemented to provide the interface between the user's program and the host operating system.
- *EZ-030 Demonstration Board*. This hardware product is intended to be an evaluation vehicle for the Am29030™ processor. The entire HIF specification is implemented on this board, which contains a resident operating system to implement the necessary kernel services.
- *SA-29200 Demonstration Board.* This hardware product contains an Am29200™ processor and memory. It is intended to be an evaluation vehicle for the Am29200 processor.
- *SD-29240 Stand-Alone Demonstration Board* has limited development support and is designed to demonstrate the Am29240™ or Am29245™ microcontrollers.
- *SA-29240 Development and Evaluation Board* provides a demonstration and evaluation platform for the Am29240, Am29245 and Am29243™ microcontrollers.
- *PC Execution Boards (EB29K development tool and EB29030 add-in board).* These hardware/software products contain an Am29000 and Am29030 processor and memory, and are add-in boards to IBM PC-based systems. One part of the HIF specification is implemented on the board, and it's counter-part, which interfaces directly with MS-DOS®, is implemented on the PC.

Because HIF is a general-purpose standard, it can be used to interface any high-level language to the 29K Family of processors. User programs need not be written entirely in a high-level language; they may incorporate assembly-language functions when maximized performance is the primary concern.

HIF Users

There are three categories of end users who need to know the details of the host interface:

- Those using AMD-supplied hardware execution vehicles or simulators. This document defines the low-level mechanisms of HIF. With this information and the design concepts presented herein, end-users can extend the HIF environment to meet the needed degree of software functionality and sophistication.
- Those developing a custom kernel operating system for any of the 29K Family of microprocessors. These users need access to AMD's high-level and assembly-language development tools. This document provides the information required to build a HIF-conforming kernel that uses the high-level language development tools directly. With this information, end-users can extend and customize the operating system code without interfering with the basic capabilities of the HIF.
- Those who are using the AMD-supplied high-level language development tools, but who must conform to another kernel operating system interface. There is sufficient information in this document to enable users to modify the development tools to properly interface with the target kernel's specifications.

HIF Concepts

Programmers developing software in a high-level language do not work directly with the processor. Instead, they think in terms of a virtual machine ideally suited to the computational paradigm of the language. For instance, the C-language virtual machine has operations such as **fprintf()** and **strcpy()**, and the FORTRAN machine has operations such as **alog** and **sqrt**.

In actual practice, these virtual machines are implemented by libraries of object code that perform language-specific operations. As long as programmers use only the functions of the language's implied virtual machine, the programs will be portable across a broad range of implementations of the language.

However, computer systems generally provide another virtual machine to the world: one that is defined by the operating system software. This virtual machine requires system calls to perform the services that are implemented within the operating system code. Typical services are: process management, file system management, device management, and memory management.

The high-level language virtual machine usually consists of: 1) functions that can be implemented entirely within library routines, and 2) functions that require the services of the operating system. The functions of the first group (usually defined as the standard library for that language) are independent of the operating system virtual machine on which they are implemented. The functions of the second group must be coded in terms of the operating system virtual machine. In other words, they must make system calls.

Making system calls is often useful for end-users, even though this practice makes their programs less portable. This requirement can be accommodated by augmenting the language library with glue routines that specifically invoke the system calls, while providing the end-user with suitable high-level syntax and semantics.

Given the previous discussion, the required task is to create high-level language development tools that can be used easily and efficiently on a variety of execution vehicles. This task can be broken down into the following steps:

- Define an operating system virtual machine that provides sufficient functionality to support the fundamental requirements of each high-level language, but not so much as to require a massive development effort to create.
- Add appropriate glue routines to the standard libraries of the language so the libraries are defined in terms of the operating-system virtual machine.
- Implement the operating system's virtual machine services on the various execution vehicles. For hardware vehicles, the virtual machine is implemented by a kernel typically contained in a resident monitor software program. For simulation vehicles, the virtual machine is implemented by code internal to the simulator and by code simulated by the simulator.

For the 29K Family of hardware and software support products, HIF consists of the following operating system virtual machine definitions:

- A carefully defined, efficient system call mechanism. Accessing a HIF kernel service requires a transition from user mode to supervisor mode on the processor. This requires a specific mechanism, such as a trap handler, to be invoked.
- A set of services supporting the primitive requirements of C, FORTRAN, and Pascal. Most of the services are defined according to UNIX operating system interface specifications.
- A specification of the environment created by the kernel. This involves the definition of storage allocation and register initializations implemented by the kernel.

Implementation Types

Implementations of the HIF specification take two fundamental forms: self-hosted and embedded.

The SA-29200 , SA-29240 and SD-29240 are some of AMD's single-board computers that incorporate microcontrollers, program and data memory. Serial ports and timer-counter resources are resident in the microcontroller. In the case of the SA-29200, the Am29200 processor is used. In the case of the SA-29240 and SD-29240 boards, one of the Am2924*x* microcontrollers is used. The HIF implementation for these boards includes a resident **osboot** program that is programmed into ROM at low-memory locations and implements the kernel services described in the "HIF Service Routines" chapter of this document.

In contrast to the single-board computers, the EB29K and EB29030 tools are two of AMD's add-in boards for IBM PC-compatible computers. The EB29K and EB29030 boards incorporate an Am29000 or Am29030 processor, program and data memory, and PC dual-interface memory resources. The HIF implementation for these boards consists of two portions of code. One portion performs some of the kernel services on the board and the other portion performs some of the kernel services through the auspices of the DOS operating system. In the sense that the HIF is grafted onto the existing host operating system, it is called an embedded implementation. The architectural and instruction simulators are also embedded implementations because they share the HIF implementation between custom code and existing host-computer operating-system code.

There is no preference for either type of implementation as long as the services and features of the HIF specification are fully implemented in the target environment. With the standard interfaces that a HIF implementation presents, application programs written for one environment will run equally well in another.

Chapter 2

System Call Mechanism

System calls on 29K processor-based systems are accomplished through invocation of a specific software trap. The 29K processor traps are roughly equivalent to software interrupts on other CPUs. System call traps are invoked through execution of an appropriate assert instruction whose assertion is FALSE at the time the instruction is executed.

Execution of an **ASEQ**, **ASGE**, **ASGEU**, **ASGT**, **ASGTU**, **ASLE**, **ASLEU**, **ASLT**, **ASLTU**, or **ASNEQ** instruction, where the result of the assertion is FALSE, will cause the trap specified in the instruction to be taken.

Once the trap is invoked, the 29K Family processor accesses a trap vector contained in a table of 256 separate trap handler addresses.

With the Am29000 and Am29050™ microprocessors, the operating system software may implement direct trap execution for increased efficiency, (although in most implementations, the table of vectors is used). Since the need for a vector table lookup is not required, this solution offers an efficiency gain, even though it requires the reservation of a much greater amount of system memory.

When a trap is taken, the normal program execution sequence is interrupted and the trap handler is invoked. At this point, the current program's context is contained in CPU registers of the 29K processor. No saving or restoring of registers is performed by the processor when a trap occurs. HIF services are required to preserve the following registers and restore their contents before returning to the application program:

- All local registers
- Global registers *gr1*, *gr112–gr115*, and *gr125*
- Global registers *gr126* and *gr127* should be preserved according to AMD calling conventions. Their values may differ upon return from a HIF service, but the span between their values will remain the same.

The HIF services may modify the contents of certain registers without first saving their values, namely: *gr121*, *gr96*, and *gr97*; although, the application program should not count on *gr96* through *gr111* to be untouched by current and future HIF kernel services.

HIF Service Invocation

Before invoking HIF services, the service number and any input parameters to be passed must be loaded into the general registers of the 29K processor. Both local and global registers are used for various HIF services, as shown in the HIF Service Calls table on page A–1. Details for invoking specific services are contained in the “HIF Service Routines” chapter.

Service Number

Every HIF system service is identified by a unique number. Service numbers 0–127 and 256–383 are reserved for use by AMD and should not be used for user-supplied extensions.

The service number must be loaded into global register *gr121*, the trap-handler argument register. Global register *gr121* is a temporary register and its value is **not** preserved over a system call, nor will its value be preserved over any trap invoked by the running program.

Input Parameters

Any input parameters to be passed must be placed in local registers *lr2* through *lr17*. See the appropriate 29K Family processor documentation for specific details describing the parameter-passing mechanism.

Invoking a HIF Service

The HIF services are accessed by forcing trap 69 to occur, after the service number and parameters (if any) are loaded in the designated registers. Trap handler 69 executes the service in supervisor mode.

Returned Values

Most of the services return values, usually a single integer value (number of bytes read or written, number of clock ticks, size of a memory block, etc.), or a pointer (address of a file descriptor, address of a memory block, etc.). These values are returned in register *gr96*, per standard high-level language calling conventions.

If a service returns multiple values, the additional values are returned in *gr97*, *gr98*, and so forth. If the service fails to perform the requested task, the validity of the values contained in *gr96* and succeeding registers is not guaranteed.

See the documentation that accompanies your language processor for additional details on 29K Family processor high-level language calling conventions.

Status Reporting

In all cases, upon return from a HIF service, global register *gr121* contains either a TRUE value (0x80000000), or a positive nonzero integer error code indicating the reason for failure. Predefined error codes for existing HIF implementations are listed starting on page B–1.

HIF does not specify these error codes. They may be completely defined by an implementation, except for cases in which there is a corresponding, existing, UNIX error code. In these cases, the UNIX error code is expected to be used (see Appendix B).

Example Assembly Code

The following code fragment shows how the definitions given previously are implemented in Am29000 processor assembly-language to invoke the **open** HIF service to open a file:

```
const    lr2,input_file    ;set input file
consth   lr2,input_file    ;pathname address
const    lr3,O_RDONLY      ;set open mode
const    gr121,17          ;service number=17 (open)
asneq    69,gr1,gr1        ;force trap 69
                           ;(a system call)
jmpf     gr121,err_hand    ;handle service error
nop
```

In this example, local register *lr2* is loaded with the address of the filename constant; local register *lr3* contains the code: O_RDONLY, indicating that the file is to be opened for read-only access. The service number (17) is loaded into global register *gr121* and the service is executed by asserting that register *gr1* is not equal to itself. Since this is FALSE, the trap is invoked. Upon return from the service, global register *gr121* contains either a TRUE value, indicating that the service was successful; or a positive nonzero error code, indicating that the service could not complete. If an error code is returned, *gr121* will test as FALSE, providing the means to invoke an appropriate error handler routine.

User-Mode Traps

When a trap is invoked, the 29K Family processor switches from user mode to supervisor mode to execute the trap handler code. Most of the traps are properly executed in this mode, including the kernel services that implement the HIF specification. However, a few traps, such as the spill/fill handlers, are intended to execute in user mode. In these cases, the trap handler code is not part of the kernel, but is supplied by the particular high-level language product library and is linked with the user's application program.

In order to use a consistent trap-handling mechanism, and to support the individual language products' methodologies for user-mode traps, a HIF service called **setvec** is called with the address of the user-mode trap- handler code for each of the traps handled in this way.

Once the user-mode handler addresses have been supplied and the corresponding trap is invoked, the operating-system kernel receives control in supervisor mode. It then reinstates user mode and invokes the appropriate language library trap handler to complete the required operation. This bouncing from user mode to supervisor mode and back to user mode is referred to as a trampoline effect. When the trap handler's execution is complete, it returns directly to the user's application program rather than back through the kernel.

The register stack spill/fill handlers are appropriate examples of code that is intended to execute in user mode. When a user's application program calls a function that requires a large number of local registers to execute, some currently unused registers may have to be written to main memory to free enough of the on-chip registers. In this case, the registers are spilled to memory via the spill-trap handler. When the function completes execution and intends to return to its caller, the spilled registers may have to be restored by calling the fill-trap handler. Since register stack management is unique for each application environment, individual spill/fill handlers are provided with each of the high-level language products.

Supervisor-Mode Traps

The **settrap** service offers the ability for supervisor-mode traps to be installed at the discretion of the implementation designer. These traps are installed directly into the vector table whose base address is pointed to by the Vector Area Base Address special-purpose register (VAB). It is up to the implementation designer whether this facility will be implemented and made available to the user program.

In many dedicated hardware systems, programs are given permission to access the full facilities of the system hardware. In this case, the implementation designer should determine which trap vectors may be set or modified by the **settrap** service. In cases where only a limited number of vectors may be modified in this way, the designer should test the **trapno** parameter to validate the request.

In cases where certain trap vectors have privileged access, or if access to the **settrap** service is not allowed to a particular user, the implementation should take care to return the EHIFNOTAVAIL error code. This will ensure portability of applications across different implementations.

When a trap occurs, whether in user or supervisor mode, the 29K Family processor enters supervisor mode to execute the trap-handling function pointed to by the trap address stored in the vector. The trap-handling function is required to save and restore the registers described in this specification.

Two special traps are handled under the auspices of the **signal** facility. This service call lets user programs specify trap handlers for user-interrupt and floating-point exception errors. The **signal** service is described beginning on page 3–64 of this specification.

Chapter 3

HIF Service Routines

The HIF service routine calls currently defined are listed by decimal service number in Table 3–1, and in alphabetical order in Table 3–2. Descriptions of the individual services follow on the remaining pages of this chapter, and are listed in order of service number. Table 3–3 describes the parameter names used in the service descriptions.

Most HIF calls are similar or identical to equivalent UNIX operating- system calls. The titles given in the tables are not the names that actually exist in a particular library but, instead, are the generic names of the services.

Service numbers 0–127 and 256–383 are reserved by AMD and should not be used for user-supplied extensions.

Each service description on the following pages contains a concise explanation of the purpose of the service, the input and result register contents, and example assembly-language code to invoke the service. In all cases, operating-system kernel services meeting the HIF specifications are invoked by forcing the software trap 69 to occur. The service number is always contained in general register *gr121* and parameters are passed, if necessary, in local registers beginning with *lr2*.

When the service returns, general register *gr121* is required to report the success or failure of the service. If successful, *gr121* is expected to contain a TRUE boolean value (a 1 bit in the most significant bit position). If the service is not successful, a positive nonzero error code is returned in *gr121*. If the service returns results, the first result is held in *gr96*, the second in *gr97*, and so forth.

HIF implementations are required to return an error code when a requested operation is not possible. The codes from 0–10,000 are reserved for compatibility with current and future HIF error return standards. The currently assigned codes and their meanings are listed in Appendix B. If a HIF implementation returns an error code in the range of 0–10,000, it **must** carry the identical meaning to the corresponding error code in this table. Error code values larger than 10,000 are available for implementation-specific errors.

In the examples for each service call, references are made to error handlers that are not part of the example code. These are assumed to be contained in the larger part of the user's code and are not supplied as part of the HIF specification. The JMPF instructions have been provided to show that interface glue routines should incorporate this error-testing philosophy in order to be robust. In practice, error handling may be relegated to a single routine, or may be vested in individual sections of either inline code, or as callable services by the glue routines.

Since HIF implementations may exist over a wide spectrum of systems, the capabilities of the HIF may vary from one system to the next. In the simplest case, the HIF implementation in an embedded Am29000 processor system, such as a printer controller, may contain no external file system. In this event, the input/output facilities specified in the kernel service descriptions need not be implemented. In more common cases, where the HIF will exist on systems that have full operating-system capabilities, such as DOS or UNIX, it is assumed that all of the features of the HIF will be implemented. The service descriptions in this document provide a set of standard interfaces for commonly implemented operating- system interfaces. If individual features are implemented, the interfaces are expected to follow the guidelines in this specification.

It is suggested that unimplemented services consist of skeleton code that always returns an EHIFNOTAVAIL error code, to aid in portability between implementations. Undefined HIF services, if invoked, should return the EHIFUNDEF error code; although this is up to the discretion of the implementor.

Table 3–1. HIF Service Calls in Numerical Order

Number	Title	Description	Page
1	exit	Terminate a program	3–7
17	open	Open a file	3–8
18	close	Close a file	3–14
19	read	Read a buffer of data from a file	3–16
20	write	Write a buffer of data to a file	3–19
21	lseek	Seek a file byte	3–22
22	remove	Remove a file	3–25
23	rename	Rename a file	3–26
24	ioctl	Input/output control	3–28
25	iowait	Test and wait I/O complete	3–32
26	iostat	Input/output status	3–35
33	tmpnam	Return a temporary name	3–37
49	time	Return seconds since 1970	3–39
65	getenv	Get environment	3–41
66		Reserved	
67	gettz	Get time zone	3–43
257	sysalloc	Allocate memory space	3–45
258	sysfree	Free memory space	3–46
259	getpsize	Return memory page size	3–48
260	getargs	Return base address	3–49
261		Reserved	
273	clock	Return time in milliseconds	3–51
274	cycles	Return processor cycles	3–53
289	setvec	Set trap address	3–55
290	settrap	Set trap vector	3–57
291	setim	Set interrupt mask	3–59
305	query	Return version information	3–61
321	signal	Register signal handler	3–64
322	sigdfl	Perform default signal action	3–68
323	sigret	Return from signal interrupt (normal)	3–69
324	sigrep	Return from signal interrupt (repeat operation)	3–70
325	sigskp	Return from signal interrupt (skip operation)	3–71
326	sendsig	Send signal	3–72

Table 3–2. HIF Service Calls in Alphabetical Order

Name	Description	Page
clock	Return time in milliseconds	3–51
close	Close a file	3–14
cycles	Return processor cycles	3–53
exit	Terminate a program	3–7
getargs	Return base address	3–49
getenv	Get environment	3–41
getpsize	Return memory page size	3–48
gettz	Get time zone	3–43
ioctl	Input/output control	3–28
iostat	Input/output status	3–35
iowait	Test and wait I/O complete	3–32
lseek	Seek a file byte	3–22
open	Open a file	3–8
query	Return version information	3–61
read	Read a buffer of data from a file	3–16
remove	Remove a file	3–25
rename	Rename a file	3–26
sendsig	Send signal	3–72
setim	Set interrupt mask	3–59
settrap	Set trap vector	3–57
setvec	Set trap address	3–55
sigdfl	Perform default signal action	3–68
signal	Register signal handler	3–64
sigrep	Return from signal interrupt (repeat operation)	3–70
sigret	Return from signal interrupt (normal)	3–69
sigskp	Return from signal interrupt (skip operation)	3–71
sysalloc	Allocate memory space	3–45
sysfree	Free memory space	3–46
time	Returns seconds since 1970	3–39
tmpnam	Return a temporary name	3–37
write	Write a buffer of data to a file	3–19

Table 3–3. Service Call Parameters

Parameter	Description
027vers	The version number of the installed Am29027 arithmetic accelerator chip (if any).
addrptr	A pointer to an allocated memory area, a command-line-argument array, a pathname buffer, or a NULL-terminated environment variable name string.
baseaddr	The base address of the command-line-argument vector returned by the **getargs** service.
buffptr	A pointer to the buffer area where data is to be read from or written to during the execution of I/O services, or the buffer area referenced by the **wait** service.
capcode	The capabilities request code passed to the **query** service. Code values are: 0 (request HIF version), 1 (request CPU version), 2 (request Am29027 arithmetic accelerator version), 3 (request CPU clock frequency), and 4 (request memory environment).
clkfreq	The CPU clock frequency (in Hertz) returned by the **query** service.
count	The number of bytes actually read from file or written to a file.
cpuvers	The CPU family and version number returned by the **query** service.
cycles	The number of processor cycles (returned value).
di	The disable interrupts parameter to the **setim** service.
dstcode	The daylight-savings-time-in-effect flag returned by the **gettz** service.
errcode	The error code returned by the service. These are usually the same as the codes returned in the UNIX *errno* variable. See Appendix B for a list of HIF error codes.
exitcode	The exit code of the application program.
filename	A pointer to a NULL-terminated ASCII string that contains the directory path of a temporary filename.
fileno	The file descriptor that is a small integer number. File descriptors 0, 1, and 2 are guaranteed to exist and correspond to open files on program entry (0 refers to the UNIX equivalent of stdin and is opened for input; 1 refers to the UNIX stdout and is opened for output; 2 refers to the UNIX stderr and is opened for output).
funaddr	A pointer to the address of a spill or fill handler passed to the **setvec** service.
hifvers	The version of the current HIF implementation returned by the **query** service.
iostat	The input/output status returned by the **iostat** service.

Parameter	Description
mask	The interrupt mask value passed to and returned by the **setim** service.
memenv	The memory environment returned by the **query** service.
mode	A series of option flags whose values represent the operation to be performed. Used in the **open**, **ioctl**, and **wait** services to specify the operating mode.
msecs	Milliseconds returned by the **clock** service.
name	A pointer to a NULL-terminated ASCII string that contains an environment variable name.
nbytes	The number of data bytes requested to be read from or written to a file, or the number of bytes to allocate or deallocate from the heap.
newfile	A pointer to a NULL-terminated ASCII string that contains the directory path of a new filename.
newsig	The address of the new user signal handler passed to the **signal** service.
offset	The number of bytes from a specified position (*orig*) in a file, passed to the **lseek** service.
oldfile	A pointer to NULL-terminated ASCII string that contains the directory path of the old filename.
oldsig	The address of the previous user signal handler returned by the **signal** service.
orig	A value of 0, 1, or 2 that refers to the beginning, the current position, or the position of the end of a file.
pagesize	The memory page size, in bytes, returned by the **getpsize** service.
pathname	A pointer to a NULL-terminated ASCII string that contains the directory path of a filename.
pflag	The UNIX file access permission codes passed to the **open** service.
retval	The return value that indicates success or failure.
secs	The seconds count returned by the **time** service.
sig	A signal number passed to the **sendsig** service.
sigptr	A pointer to the HIF signal stack containing preserved registers.
trapaddr	The trap address returned by the **setvec** and **settrap** services; a trap address passed to and returned by the **settrap** service.
trapno	The trap number passed to the **setvec** and **settrap** services.
where	The current position in a specified file returned by the **lseek** service.
zonecode	The time zone minutes correction value returned by the **gettz** service.

Service 1 – exit
Terminate a Program

Description

This service terminates the current program and returns a value to the system kernel, indicating the reason for termination. By convention, a zero passed in *lr2* indicates normal termination, while any nonzero value indicates an abnormal termination condition. There are no returned values in registers *gr96* and *gr121* since this service does not return.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	1 (0x1)	Service number
	lr2	exitcode	User-supplied exit code
Returns:	*gr96*	undefined	This service call does not return
	gr121	undefined	This service call does not return

Example Call

```
const    lr2,1          ;exit code = 1
const    gr121,1        ;service = 1
asneq    69,gr1,gr1     ;call the operating system
```

In the above example, the operating system kernel is being called with service code 1 and an exit code of 1, which is interpreted according to the specifications of the individual operating system. The value of the exit code is not defined as part of the HIF specification.

In general, however, an exit code of zero (0) specifies a normal program termination condition, while a nonzero code specifies an abnormal termination resulting from detection of an error condition within the program.

Programs can terminate normally by falling through the curly brace at the end of the **main** function in a C-language program. Other languages may require an explicit call to the kernel's **exit** service.

Service 17 – open
Open a File

Description

This service opens a named file in a requested mode. Files must be explicitly opened before any **read**, **write**, **close**, or other file-positioning accesses can be accomplished. The **open** service, if successful, returns an integer token that is used to refer to the file in all subsequent service requests. In many high-level languages, the returned token is referred to as a file descriptor. Filenames are generally not portable from one implementation to another. In some cases, names can be made more portable by limiting them to six or fewer uppercase alphabetic characters, or by using the **tmpnam** HIF service (33) to create names that conform to the current implementation's file system requirements.

Environment variables can also be used to specify legal filenames for application programs wishing to conform to the requirements of a particular HIF implementation. The **getenv** service (65) provides the means to associate a filename or pathname with a mnemonic reference. This is the most portable means to specify pathnames for implementations incorporating the **getenv** service.

The HIF specification intentionally refrains from defining the constituents of a legal pathname or any intrinsic characteristics of the implemented file system. In this regard, the only requirement of a HIF-conforming kernel is that when the **open** service is successfully performed, the routine must return a small integer value that can be used in subsequent input/output service calls to refer to the opened file.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	17 (0x17)	Service number
	lr2	pathname	A pointer to a filename
	lr3	mode	See parameter descriptions below
	lr4	pflag	See parameter descriptions below
Returns:	*gr96*	fileno	Success: $\geq$ 0 (file descriptor) Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Parameter Descriptions

Pathname is a pointer to a zero-terminated string that contains the full path and name of the file being opened. Individual operating systems have different means to specify this information. With hierarchical file systems, individual directory levels are separated with special characters that cannot be part of a valid filename or directory name. In UNIX-compatible file systems, directory names are separated by forward slash characters, /, (e.g., **/usr/jack/files/myfile**); where **usr**, **jack**, and **files** are succeedingly lower directory levels, beginning at the root directory of the file system. The name **myfile** is the filename to be opened at the specified level. The individual characteristics of files and pathnames are determined by the specifications of a particular operating system implementation.

The *mode* parameter is composed of a set of flags whose mnemonics and associated values are listed in Table 3–4.

Table 3–4. Open Service Mode Parameters

Name	Value	Description
O_RDONLY	0x0000	Open for read-only access
O_WRONLY	0x0001	Open for write-only access
O_RDWR	0x0002	Open for read and write access
O_APPEND	0x0008	Always append when writing
O_NDELAY	0x0010	No delay
O_CREAT	0x0200	Create file if it does not exist
O_TRUNC	0x0400	If the file exists, truncate it to zero length
O_EXCL	0x0800	Fail if writing and the file exists
O_FORM	0x4000	Open in text format

The O_RDONLY mode provides the means to open a file and guarantee that subsequent accesses to that file will be limited to **read** operations. The operating system implementation will determine how errors are reported for unauthorized operations. The file pointer is positioned at the beginning of the file unless the O_APPEND mode is also selected.

The O_WRONLY mode provides the means to open a file and guarantee that subsequent accesses to that file will be limited to **write** operations. The operating system implementation will determine how errors are reported for unauthorized operations. The file pointer is positioned at the beginning of the file unless the O_APPEND mode is also selected.

The O_RDWR mode provides the means to open a file for subsequent **read** and **write** accesses. The file pointer is positioned at the beginning of the file unless the O_APPEND mode is also selected.

If O_APPEND mode is selected, the file pointer is positioned to the end of the file at the conclusion of a successful **open** operation, so that data written to the file is added following the existing file contents.

Ordinarily, a file must already exist in order to be opened. If the O_CREAT mode is selected, files that do not currently exist are created; otherwise, the **open** function will return an error condition in *gr121*.

If a file being opened already exists and the O_TRUNC mode is selected, the original contents of the file are discarded and the file pointer is placed at the beginning of the (empty) file. If the file does not already exist, the HIF service routine should return an error value in *gr121*, unless O_CREAT mode is also selected.

The O_EXCL mode provides a method for refusing to open the file if the O_WRONLY or O_RDWR modes are selected and the file already exists. In this case, the kernel service routine should return an error code in *gr121*.

O_FORM mode indicates that the file is to be opened as a text file rather than a binary file. The nominal standard input, output, and error files (file descriptors 0, 1, and 2) are assumed to be open in text mode prior to commencing execution of the user's program.

When opening a FIFO (interprocess communication file) with O_RDONLY or O_WRONLY set, the following conditions apply:

- If O_NDELAY is set (i.e., equal to 0x0010):
  - A read-only open will return without delay.
  - A write-only open will return an error if no process currently has the file open for reading.
- If O_NDELAY is clear (i.e., equal to 0x0000):
  - A read-only open will block until a process opens a file for writing.
  - A write-only open will block until a process opens a file for reading.

When opening a file associated with a communication line (e.g., a remote modem or terminal connection), the following conditions apply:

- If O_NDELAY is set, the open will return without waiting for the carrier detect condition to be TRUE.
- If O_NDELAY is clear, the open will block until the carrier is found to be present.

The optional *pflag* parameter specifies the access permissions associated with a file; it is only required when O_CREAT is also specified (i.e., create a new file if it does not already exist). If the file already exists, *pflag* is ignored. This parameter specifies UNIX-style file access permission codes (*r*, *w*, and *x* for read, write, and execute, respectively) for the file's owner, the work group, and other users. If *pflag* is -1, then all accesses are allowed. See the UNIX operating system documentation for additional information on this topic.

Example Call

```
path:       .ascii     "/usr/jack/files/myfile\0"
            .set       mode,0_RDWR|0_CREAT|0_FORM
            .set       permit,0x180

fd:         .word      0
            const      lr2,path            ;address of
            consth     lr2,path            ;pathname
            const      lr3,mode            ;open mode
                                           ;settings
            const      lr4,permit          ;permissions
            const      gr121,17            ;service=17 (open)
            asneq      69,gr1,gr1          ;perform OS call
            jmpf       gr121,open_err      ;jump if error on
                                           ;open
            const      gr120,fd            ;set address of
            consth     gr120,fd            ;file descriptor
            store      0,0,gr96,gr120      ;store file
                                           ;descriptor
```

In the above example, the file is being opened in read/write text mode. The UNIX permissions of the owner are set to allow reading and writing, but not execution, and all other permissions are denied. As indicated above in the parameter descriptions, the file permissions are only used if the file does not already exist. When the **open** service returns, the program jumps to the **open_err** error handler if the open was not successful; otherwise, the file descriptor returned by the service is stored for future use in **read**, **write**, **lseek**, **remove**, **rename**, or **close** service calls.

As described in the introduction to these services, the HIF can be implemented to several degrees of elaboration, depending on the underlying system hardware and whether the operating system is able to provide the full set of kernel services. In the least capable instance (i.e., a standalone board with a serial port), it is likely that only the O_RDONLY, O_WRONLY, and O_RDWR modes will be supported. In more capable systems, the additional modes should be implemented if possible.

If an error is encountered during the execution of an **open** call, no file descriptor will be allocated.

Service 18 – close
Close a File

Description

This service closes the open file associated with the file descriptor passed in *lr2*. Closing all files is automatic on program exit (see **exit**), but since there is an implementation-defined limit on the number of open files per process, an explicit **close** service call is necessary for programs dealing with many files.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	18 (0x12)	Service number
	lr2	fileno	File descriptor
Returns:	*gr96*	retval	Success: = 0 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
fd:         .word     0
            const     gr96,fd             ;set address of
            consth    gr96,fd             ;file descriptor
            load      0,0,lr2,gr96        ;get file descrip-
                                          ;tor
            const     gr121,18            ;service=18
            asneq     69,gr1,gr1          ;call the OS
            jmpf      gr121,clos_err      ;handle close
                                          ;error
            nop
```

The previous example illustrates loading a previously stored file descriptor (*fd*, in this case) and calling the kernel's **close** service to close the file associated with that descriptor. If an error occurs when attempting to close the file, the kernel will return an error code in *gr121* (the content of that register will not be TRUE) and the program will jump to an error handler; otherwise, program execution will continue. The file will be closed and the file descriptor deallocated, even when an error is encountered. Requested data is available.

Service 19 – read
Read a Buffer of Data from a File

Description

This service reads a number of bytes from a previously opened file (identified by a small integer file descriptor in *lr2* that was returned by the **open** service) into memory starting at the address given by the buffer pointer in *lr3*. *lr4* contains the number of bytes to be read. The number of bytes actually read is returned in *gr96*. Zero is returned in *gr96* if the file is already positioned at its end-of-file. If an error is detected, a small positive integer is returned in *gr121*, indicating the nature of the error.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	19 (0x13)	Service number
	lr2	fileno	File descriptor
	lr3	buffptr	A pointer to buffer area
	lr4	nbytes	Number of bytes to be read
Returns:	*gr96*	count*	*See **Return Value** table, below.
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

The value returned in register *gr96* can be interpreted differently, depending on the current operating mode of the file identified by the *fileno* parameter. The operating mode is established or changed by invoking the **ioctl** service (24). The Return Value table shows how the return value in *gr96* should be interpreted for various operating modes.

Return Value

Count	Non-ASYNC	ASYNC	NBLOCK
gr96>0	count	n/a	count
gr96 =0	EOF	success	EOF
gr96 <0	fail	fail	if = –1 and *gr121 =* EAGAIN, no data is available. Otherwise, fail.

In the Return Value table, for normal synchronous **read** service requests, the return value contains a *count* of the number of bytes read (if *gr96* > 0), end-of-file (if *gr96* = 0), or an indication that the operation failed (*gr96* < 0). For ASYNC mode, the operation is only scheduled by invoking the **read** service, so the return value in *gr96* merely indicates that the request succeeded or failed. Nonblocking **read** requests indicate that data is to be returned if available; otherwise, the service is to return control to the user process with an indication that the operation would block if allowed to continue. When *gr96* contains the value –1, and the *errcode* value in register *gr121* is EAGAIN, then no data is available to be read. If *gr96* contains any other negative value, or if register *gr121* contains any other error code, the service request was not accepted.

If the operating mode of the file descriptor referenced by the **read** service has previously been set to ASYNC using the **ioctl** service, the **iowait** service should be used to test the completion status of this operation, and to access the number of bytes that have been transferred. If a previously issued asynchronous **read**, **write**, or **lseek** operation is not complete, the current **read** request will return a failure status. Only one outstanding request is allowed.

If the operating mode has previously been set to NBLOCK (nonblocking), the *count* value returned in *gr96* will only reflect the number of bytes currently available in the buffer. NBLOCK mode only applies to terminal-like devices.

Example Call

```
fd:         .word     0
            .block    256
            const     gr119,fd
            consth    gr119,fd
            load      0,0,lr2,gr119     ;get file
                                        ;descriptor
            const     lr3,buf           ;set buffer
            consth    lr3,buf           ;address
            const     lr4,256           ;specify buffer
                                        ;size
            const     gr121,19          ;service = 19
            asneq     69,gr1,gr1        ;call the OS
            jmpf      gr121,rd_err      ;handle read errors
```

The example call requests the HIF to return 256 bytes from the file descriptor contained in the variable *fd*. If the call is successful, *gr121* will contain a TRUE value and *gr96* will contain the number of bytes actually read. If the service fails, *gr121* will contain the error code.

Service 20 – write
Write a Buffer of Data to a File

Description

This service writes a number of bytes from memory (starting at the address given by the pointer in *lr3*) into the file specified by the small positive integer file descriptor that was returned by the **open** service when the file was opened for writing. *lr4* contains the number of bytes to be written. The number of bytes actually written is returned in *gr96*. If an error is detected, *gr121* will contain a small positive integer on return from the service, indicating the nature of the error.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	20 (0x14)	Service number
	lr2	fileno	File descriptor
	lr3	buffptr	A pointer to buffer area
	lr4	nbytes	Number of bytes to be written
Returns:	*gr96*	count*	*See **Return Value** table, below.
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

The value returned in register *gr96* can be interpreted differently, depending on the current operating mode of the file identified by the *fileno* parameter. The operating mode is established or changed by invoking the **ioctl** service (24). The following table shows how the return value in *gr96* should be interpreted for various operating modes.

Return Value

Count	Non-ASYNC	ASYNC	NBLOCK
gr96=lr4	success	n/a	(NBLOCK mode is not illegal for **write** requests, but requests are performed in either synchronous or ASYNC mode. Return values are interpreted accordingly.)
0≤ gr96 < lr4	fail	=0, success	
gr96 <0	extreme	fail	

In the Return Value table, for normal synchronous **write** service requests, the return value contains a *count* of the number of bytes written. If the value returned in *gr96* is equal to the *nbytes* argument passed to the service in *lr4*, the write operation was successful. Any other return value indicates that an error occurred. If *gr96* contains a value between 0 and the value of *nbytes*, the failure is not catastrophic. Negative values returned in *gr96* indicate extreme errors.

For ASYNC mode, the operation is only scheduled by invoking the **write** service, so the return value in *gr96* merely indicates that the request succeeded or failed. A return value of 0 in *gr96* indicates that the asynchronous write operation was successfully scheduled.

Nonblocking **write** requests are performed in either synchronous or asynchronous mode, depending on whether the ASYNC operating mode was selected. NBLOCK mode is ignored; the return value in *gr96* is interpreted according to the values shown for non-ASYNC and ASYNC modes in the table.

Example Call

```
fd:       .word    0
buf:      .block   256
          const    gr96,fd          ;set address of
          consth   gr69,fd          ;file descriptor
          load     0,0,lr2,gr96     ;get file
                                    ;descriptor
          const    lr3,buf          ;set buffer
          consth   lr3,buf          ;address
          const    lr4,256          ;specify buffer
                                    ;size
          const    gr121,20         ;service = 20
          asneq    69,gr1,gr1       ;call the OS
          jmpf     gr121,wr_err     ;handle write
                                    ;errors
          const    gr120,num        ;set address of
          consth   gr120,num        ;“num” variable
          store    0,0,gr96,gr120   ;store bytes
                                    ;written
```

The example call writes 256 bytes from the buffer located at *buf* to the file associated with the descriptor stored in *fd*. If errors are detected during execution of the service, the value returned in *gr121* will be FALSE. In this case, the **wr_err** error handler will be invoked. The number of bytes actually written is stored in the variable *num*.

Service 21 – lseek
Seek a File Byte

Description

This service positions the file associated with the file descriptor in *lr2*, in an *offset* number of bytes from the position of the file referred to by the *orig* parameter. *lr3* contains the number of bytes offset and *lr4* contains the value for *orig*. The parameter *orig* is defined as:

0 = Beginning of the file

1 = Current position of the file

2 = End of the file

The **lseek** service can be used to reposition the file pointer anywhere in a file. The offset parameter may either be positive or negative. However, it is considered an error to attempt to seek in front of the beginning of the file. Any attempt to seek past the end of the file is undefined and is dependent on the restrictions of each implementation.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	21 (0x15)	Service number
	lr2	fileno	File descriptor
	lr3	offset	Number of bytes offset from orig
	lr4	orig	A code number indicating the point within the file from which the offset is measured
Returns:	*gr96*	where*	*See **Return Value** table, below.
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

The value returned in register *gr96* can be interpreted differently, depending on the current operating mode of the file identified by the *fileno* parameter. The operating mode is established or changed by invoking the **ioctl** service (24). The Return Value table shows how the return value in *gr96* should be interpreted for various operating modes.

Return Value

Count	Non-ASYNC	ASYNC	NBLOCK
gr96≥0 *gr96<0*	where fail	n/a fail	(NBLOCK mode is not illegal for **lseek** requests, but requests are performed in either synchronous or ASYNC mode. Return values are interpreted accordingly.)

In the Return Value table, for normal synchronous **lseek** service requests, the return value contains the current position in the file, if the value is greater than or equal to 0. Negative values returned in *gr96* indicate that the request was not accepted.

The file position returned by the **lseek** service in *gr96* (*where*) is always measured from the beginning of the file. A value of 0 refers to the beginning, and any other positive nonzero value refers to the current position in the file. To determine the size in bytes for a particular file, an **lseek** request with an *offset* value of 0 and an *orig* value of 2 will position the file to its end and return the byte position of the end-of-file, which is an accurate measure of the size of the file.

Asynchronous **lseek** requests are allowed if the operating mode for the file descriptor associated with the request has been set to ASYNC. In this case, the file position returned in *gr96* (*where*) will not be relevant. The **iowait** service call should be used to determine the final file position when the seek operation is complete.

If a previously issued **read** or **write** request is still in progress when an **lseek** is issued, a failure status will be returned for the **lseek** request. Only one request can be pending at a time. To properly handle this situation, the **iowait** service should be used to ensure the completion of an outstanding **read** or **write** before issuing the **lseek** service request.

Example Call

```
fd:     .word     6                ;file descriptor=6
        const     gr96,fd          ;set address of
        consth    gr69,fd          ;file descriptor
        load      0,0,lr2,gr96     ;get file descriptor
        consth    lr3,23           ;offset argument=23
        consth    lr4,0            ;origin argument=0
        const     gr121,21         ;service = 21
        asneq     69,gr1,gr1       ;call the OS
        jmpf      gr121,seek_err   ;seek error if false
        nop
```

The call example shows how a file can be positioned to a particular byte address by specifying the *orig*, which is the starting point from which the file position is adjusted, and the *offset*, which is the number of bytes from the *orig* to move the file pointer. In this case, the file identified by file descriptor 6 is being repositioned to byte 23, measured from the beginning of the file (origin = 0).

The file descriptor, *offset*, and *orig* values are loaded and **lseek** is called to perform the file positioning operation. If an error occurs when attempting to reposition the file, the value returned in *gr121* is FALSE, and contains an error code that indicates the reason for the error. Upon return, *gr96* also contains the file position measured from the beginning of the file.

Service 22 – remove
Remove a File

Description

This service deletes a file from the file system. *lr2* contains a pointer to the pathname of the file. The path must point to an existing file, and the referenced file should not be currently open. The behavior of the remove service is undefined if the file is open. Any attempt to remove a currently open file will have an implementation-dependent result.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	22 (0x16)	Service number
	lr2	pathname	A pointer to string that contains the pathname of the file
Returns:	*gr96*	retval	Success: = 0 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
path:     .ascii    "/usr/jack/files/myfile\0"

          const     lr2,path          ;set address of
          consth    lr2,path          ;file pathname
          const     gr121,22          ;service = 22
          asneq     69,gr1,gr1        ;call the OS
          jmpf      gr121,rem_err     ;jump if error
          nop
```

In the example call, a file with a UNIX-style pathname stored in the string named path is being removed. The address (*pointer*) to the string is put into *lr2* and the kernel service 22 is called to remove the file. If the file does not exist, or if it has not previously been closed, an error code will be returned in *gr121*; otherwise, the value in *gr121* will be TRUE.

Service 23 – rename
Rename a File

Description

This service moves a file to a new location within the file system. *lr2* contains a pointer to the file's old pathname and *lr3* contains a pointer to the file's new pathname. When all components of the old and new pathnames are the same, except for the filename, the file is said to have been renamed. The file identified by the old pathname must already exist, or an error code will be returned and the rename operation will not be performed.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	23 (0x17)	Service number
	lr2	oldfile	A pointer to string containing the old pathname of the file
	lr3	newfile	A pointer to string containing the new pathname of the file
Returns:	*gr96*	retval	Success: = 0
			Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
old:      .ascii    "/usr/fred/payroll/report\0"
path:     .ascii    "/usr/fred/history/june89\0"

          const     lr2,old           ;set address of
          consth    lr2,old           ;old pathname
          const     lr3,new           ;set address of
          consth    lr3,new           ;new pathname
          const     gr121,23          ;service = 23
                                      ;(rename)
          asneq     69,gr1,gr1        ;call the OS
          jmpf      gr121,ren_err     ;jump if rename
                                      ;error
          nop
```

The example call moves a file from its old path (renaming it in the process) to its new pathname location. The file will no longer be found at the old location.

Service 24 – ioctl
Input/Output Control

Description

This service establishes the operating mode of the specified file or device. It is intended to primarily be applied to terminal-like devices; however, certain modes apply to mass-storage files or to other related input/output devices.

Type	Regs	Contents	Description
Calling:	*gr121*	24 (0x18)	Service number
	lr2	fileno	File descriptor number to be tested
	lr3	mode	Operating mode
Returns:	*gr121*	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful. EHIFNOTAVAIL if service not implemented (implementation dependent)

Parameter Descriptions

In the above interface, local register *lr2* is expected to contain a legal file descriptor, *fileno*, assigned by the HIF **open** service (HIF service number 17). The *mode* parameter establishes the desired operating mode, which is selected from one or more of the following:

Table 3–5. Open Service Mode Parameters

Name	Value	Description
COOKED	0x0000	Process I/O data characters
RAW	0x0001	Do not process I/O data characters
CBREAK	0x0002	Process only I/O signals
ECHO	0x0004	Echo read data
ASYNC	0x0008	Asynchronous data read
NBLOCK	0x0010	Nonblocking data read

Multiple mode values are possible; however, COOKED, RAW, and CBREAK modes are mutually exclusive. Other mode values can be combined with these by logically ORing them to form a composite mode value. Certain mode values do not apply to every open file descriptor. For example, the ASYNC mode is used to establish a data input mode that will cause a **read**, **write**, or **lseek** operation, once initiated, to complete at a later time. With the ASYNC mode set, a **read** or **write** request will immediately return after passing the buffer address and file descriptor to the operating system, leaving the scheduling of the operation up to the HIF implementation. **lseek** operations can also be serviced in ASYNC mode. The completion status of these operations can be tested by issuing an **iowait** service request (HIF service number 25). When a **read** or **write** operation is issued for a file descriptor whose operating mode is ASYNC, the *count* returned in *gr96* will be 0 if the operation was accepted, or less than 0 if the operation was rejected. An **iowait** service should be issued to ascertain the number of bytes that have been transferred upon completion of the operation.

The default I/O processing mode is COOKED (0x0000), which implies that the HIF implementation examines input and output data characters as they are received, or before they are sent, and may perform some alteration of the data. Specific alterations are not explicitly indicated in this specification; however, it is common to perform end-of-line processing for files whose operating mode is COOKED. ASCII carriage-return and line-feed translations are common, as may be the translation of ASCII TAB characters to a number of equivalent spaces. When RAW mode is selected, no translation of input or output characters will be performed by HIF-conforming implementations.

Normally, when a **read** operation is issued for a terminal-like device by the application program, the processor will block any further execution of the subject program until the data has been transferred. The NBLOCK mode is intended to specify for terminal-like devices that subsequent **read** operations be executed without suspending (blocking) further CPU operation. This is particularly relevant to **read** operations when RAW mode is also selected. If NBLOCK mode has been specified, a subsequent **read** operation will return (in *gr96*) the number of characters currently available, or –1 if none are available. NBLOCK mode is not meaningful for **write** operations, but they are handled in the same fashion as synchronous or asynchronous operations, depending on whether ASYNC mode was specified.

RAW mode delivers the characters to/from the I/O device without conversion or interpretation of any kind.

If COOKED mode has been selected, line-buffering is implied. If NBLOCK is also selected, a subsequent **read** operation will return –1 for the *count*, unless an entire line of input is available.

The ECHO mode applies only to the standard input device (file descriptor = 0), and makes provision to automatically echo data received from that device to the standard output device (file descriptor = 1). ECHO mode is undefined for any other file descriptor.

The CBREAK mode is intended for file descriptors that refer to serial communication channels. CBREAK mode specifies that I/O signal inputs will be processed, which could alter the data stream.

The NBLOCK and ASYNC settings are not necessarily mutually exclusive. There may be occasions where this is a legal mode. NBLOCK specifies that subsequent **read**, **write**, or **lseek** operations not block until completion. If a **read** is requested, for example, and no data is currently available, the **read** service will return –1 (with an *errcode* value in *gr121* of EAGAIN), rather than blocking further execution until data becomes available. ASYNC mode simply allows an operation, once invoked, to proceed asynchronously with other operations, if the HIF implementation provides this capability.

If the above *mode* settings are not implemented, the EHIFNOTAVAIL error code should be returned to the user if the **ioctl** service is invoked.

Although the *mode* parameter occupies a 32-bit word, only the low-order 16-bits are reserved. The upper 16-bits are available for implementation-dependent mode settings, and are not part of this specification.

Example Call

```
fd:         word      0                ;variable to
                                       ;contain the file
                                       ;descriptor
            const     gr120,fd         ;Get fd address
            consth    gr120,fd         ;
            load      0,0,lr2,gr120    ;load file
                                       ;descriptor
            const     lr3,0x0010       ;NBLOCK mode
            const     gr121,24         ;service = 24
            asneq     69,gr1,gr1       ;call the OS
            jmpf      gr121,io_err     ;jump if failure
```

In the example call, a previously assigned file descriptor is passed to the service in order to specify that subsequent read requests not block if data is not available. If an error occurs when servicing this request, *gr121* will be set to FALSE and the program will jump to an error handling routine (**io_err**) when the service returns.

Service 25 – iowait
Test and Wait I/O Complete

Description

This service is used in conjunction with the **ioctl** (ASYNC mode) and **read, write**, or **lseek** services to test the completion of an asynchronous input/output operation and, optionally, to wait until the operation is complete. The **iowait** service is called with the file descriptor returned by the **open** service when the file was originally opened. The *mode* parameter specifies whether the **iowait** will block until the operation is complete, or immediately return the completion status in the result register (*gr96*). If the operation was complete, *gr96* will contain the number of bytes transferred for **read** or **write** service requests (*count*), or the ending file position (measured from the beginning of the file) for **lseek** service requests (*where*).

If no previous asynchronous (**ioctl** ASYNC mode) **read**, **write**, or **lseek** service is pending for the specified file descriptor, or if an unrecognized mode value is provided, the **iowait** service will return an error status in *gr121*.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	25 (0x19)	Service number
	lr2	fileno	File descriptor, as returned by **open** (17)
	lr3	mode	1 = nonblocking completion test 2 = wait until read operation complete
Returns:	*gr96*	count*	*See **Return Value** table
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

The value returned in register *gr96* can be interpreted differently, depending on the value specified in the *mode* parameter (in register *lr3*) of the service request. The Return Value table shows how the return value in *gr96* should be interpreted for nonblocking and blocking completion tests.

Return Value

Count	Blocking Tests		Nonblocking Tests	
	read/write	**lseek**	**read/write**	**lseek**
gr96>0	count	where	count	where
gr96=0	EOF	where	EOF	where
gr96<0	fail	fail	IF= –1 and *gr121*=EAGAIN, there is no data available; otherwise, fail.	

In the Return Value table, for blocking completion tests, the return value specifies the status of the completed operation. If the operation was a **read** or **write** service request, the *count* value specifies the number of bytes actually transferred (gr96 > 0), that an end-of-file condition was reached (gr96 = 0), or that a failure occurred (gr96 < 0). For **lseek** requests, the return value specifies the current position of the file, unless the value is negative, in which case a failure occurred.

The return value for nonblocking completion tests of **read** and **write** service requests is interpreted the same as for blocking completion tests, except for the case where the value in *gr96* is equal to –1. In this case, and if the *errcode* in register *gr121* is EAGAIN, then no data is currently available. Any other negative return value or error code signals a failure condition.

The **iowait** service reports errors that may have occurred in the outstanding asynchronous operation— subsequent to its original issue—as well as errors in the **iowait** call itself.

Example Call

```
fd:       .word     0                ;file descriptor
          const     lr3,1            ;nonblocking
                                     ;completion test
          const     gr121,25         ;service = 25
                                     ;(iowait)
loop:     const     gr120,fd         ;load file descrip-
          consth    gr120,fd         ;tor address
          load      0,0,lr2,gr120    ;get file
                                     ;descriptor
          asneq     69,gr1,gr1       ;call the OS
          jmpf      gr121,wait_err   ;handle wait error
          const     lr3,1            ;nonblocking
                                     ;completion test
          jmpt      gr96,loop        ;wait until
                                     ;operation complete
          const     gr121,25         ;service = 25
                                     ;(iowait)
```

In the example call, the file descriptor (*fileno*) is loaded into *lr2*, nonblocking mode is selected, and the **iowait** service is invoked. If the service returns an error status in *gr121*, the program will jump to the **wait_err** label. If the operation is accepted, *gr96* will contain the completion status upon return from the service. This example jumps to reinvoke the service if the operation is not yet complete. This is equivalent to issuing a **iowait** service with a *mode* value of 2, specifying that the operation should block until the operation is complete. A more complex program might perform some useful work before retrying the operation.

Service 26 – iostat
Input/Output Status

Description

This service returns the status corresponding to a file descriptor assigned by the **open** service. If the specified file descriptor is not legal, an error code will be returned in *gr121*; otherwise, *gr121* will contain a TRUE result and *gr96* will contain the requested status. Two status values are defined:

0x0001	RDREADY	Input device ready and data available
0x0002	ISATTY	File descriptor refers to a terminal-like device (TTY)

Application programs frequently need to determine if data is currently available to be read for a terminal-like device. If the RDREADY status is returned, at least one byte of data is available to be read from the device.

The ISATTY status indicates that the device associated with the file descriptor refers to a terminal-like peripheral, rather than a mass-storage file or other peripheral device. The **iostat** service can be used to determine if a standard output device (file descriptors 1 or 2) refers to a terminal, or if output is being redirected to a mass-storage file.

The RDREADY and ISATTY status values are *not* mutually exclusive; either or both results may be present. Although the status is returned in a 32-bit word, only the lower 16 bits are reserved for HIF-conforming reply values. The upper 16 bits are available for implementation-specific status results.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	26 (0x19)	Service number
	lr2	fileno	File descriptor number
Returns:	*gr96*	iostat	Input status 0x0001 = RDREADY 0x0002 = ISATTY
	gr121	0x80000000 errcode	Logical TRUE, service successful error number, service not successful (implementation dependent)

Example Call

```
        const   lr2             ;set file
                                ;descriptor = 0
        const   gr121,26        ;service = 26
        asneq   69,gr1,gr1      ;call the OS
        jmpf    gr121,fail      ;handle failure
        sll     gr120,gr96,30   ;test ISATTy status
                                ;bit
        jmpf    gr120,not_tty   ;jump if not a tty
        nop
```

In the example call, the program calls the **iostat** service to determine if the device associated with file descriptor 0 is a **tty-like** device. If the service returns an error indication in *gr121*, the program jumps to the **fail** label; otherwise, the *iostat* value returned in *gr96* is shifted to put bit position 1 of the result into the sign-bit of *gr120*, which is tested to determine if the file descriptor refers to a **tty-like** device. If not, the program jumps to the **not_tty** label.

Service 33 – tmpnam
Return a Temporary Name

Description

This service generates a string that can be used as a temporary file pathname. A different name is generated each time it is called. The name is guaranteed not to duplicate any existing filename. The argument passed in *lr2* should be a valid pointer to a buffer that is large enough to contain the constructed filename. User programs are required to allocate a minimum of 128 bytes for this purpose.

If the argument in *lr2* contains a NULL pointer, the HIF service routine should treat this as an error condition and return a nonzero error number in global register *gr121*.

The HIF specification sets no standards for the format or content of legal pathnames; these are determined by individual operating-system requirements. Each implementation must undertake to construct a valid filename that is also unique.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	33 (0x21)	Service number
	lr2	addrptr	A pointer to buffer into which the filename is to be stored
Returns:	*gr96*	filename	Success: pointer to the temporary filename string Failure: =0 (NULL pointer)
	gr121	0x80000000	Logical TRUE, service successful
		errcode	Error number, service not successful (implementation dependent)

Example Call

```
fbuf:     .block    21              ;buffer size = 21 bytes
          const     lr2,fbuf        ;set buffer pointer
          consth    lr2,fbuf
          const     gr121,33        ;service = 33
          asneq     69,grl,grl      ;call the OS
          jmpf      gr121,tmp_err   ;jump if error
          nop
```

In the example call, the **tmpnam** service is called with a pointer to *fbuf*, which has been allocated to hold a name that is up to 21 bytes in length. If the service is able to construct a valid name, the filename will be stored in *fbuf* when the service returns. If the content of *gr121* on return is not TRUE, the program fragment jumps to **tmp_err** to handle the error condition.

Service 49 – time
Return Seconds Since 1970

Description

This service returns, in register *gr96*, the number of seconds elapsed since midnight, January 1, 1970, as an integer 32-bit value. It is assumed that the kernel service will have access to a counter whose contents can be preloaded that measures time, with at least a 1-second resolution, for this purpose.

The time value returned by this service is Greenwich Mean Time (GMT). The conversion to local time should be accomplished by a separate function that uses the value returned by the **time** service and the time-zone information from the **gettz** (Get time zone) service call to compute the correct local time.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	49 (0x31)	Service number
Returns:	*gr96*	secs	Success: ≠ 0 (time in seconds) Failure: = 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
secs:     .word          0
          const          gr121,49         ;service = 49
          asneq          69,gr1,gr1       ;call the OS
          jmpf           gr121,tim_err    ;jump if error
          const          gr120,secs       ;set the address
          consth         gr120,secs       ;for storing
                                          ;'secs'
          store          0,0,gr96,gr120   ;store the seconds
```

In the example call, the kernel service **time** is being called. If the value returned in *gr121* is TRUE, the number of seconds returned in *gr96* is stored in the *secs* variable; otherwise, the program jumps to **tim_err** to determine the cause of the error.

Service 65 – getenv
Get Environment

Description

This service searches the system environment for a string associated with a specified symbol. *lr2* contains a pointer to the symbol name. If the symbol name is found, a pointer to the string associated with it is returned in *gr96*; otherwise, a NULL pointer is returned.

In UNIX-hosted systems, the **setenv** command allows a user to associate a symbol with an arbitrary string. For example, the command **setenv TERM vt100** defines the string **vt100** to be associated with the symbol named **TERM.** Application programs can use this association to determine the type of terminal connected to the system, and therefore, use the correct set of codes when outputting information to the user's screen. To access the string, **getenv** should be called with *lr2* pointing to a string containing the **TERM** symbol name. The address returned in *gr96* will point to the corresponding vt100 string if **TERM** is found. In UNIX-hosted systems, entering a different **setenv** command lets the user select a different terminal name without requiring recompilation of the application program.

Operating-system implementations that do not include provisions for the environment variable, if always, should return a NULL value in *gr96* when this service is requested.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	65 (0x41)	Service number
	lr2	name	A pointer to the symbol name
Returns:	*gr96*	addrptr	Success: pointer to the symbol name string Failure: =0 (NULL pointer)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
mysym:    .ascii    "MYSYMBOL\0"
strptr    .word     0
          const     lr2,mysym          ;set address of
                                       ;symbol
          consth    lr2,mysym          ;to be located in
                                       ;environment
          const     gr121,65           ;service = 65
          asneq     69,gr1,gr1         ;call the os
          jmpf      gr121,env_err      ;jump if error
          const     gr120,strptr       ;set address of
          consth    gr120,strptr       ;string pointer
          store     0,0,gr96,gr120     ;store string
                                       ;pointer
```

The example call program calls the operating system **getenv** service to access a string associated with the environment variable *MYSYMBOL*. If the symbol is found, a pointer to the string associated with the symbol is returned in *gr96*. If the call is not successful (i.e., *gr121* holds a FALSE Boolean value upon return), the program jumps to **env_err** to handle the error condition.

Service 67 – gettz
Get Time Zone

Description

This service terminates the current program and returns a value to the system kernel, indicating the reason for termination. By convention, a zero passed in *lr2* indicates normal termination, while any nonzero value indicates an abnormal termination condition. There are no returned values in registers *gr96* and *gr121* since this service does not return.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	67 (0x43)	Service number
Returns:	*gr96*	zonecode	Success: ≥ 0 (minutes west of GMT) Failure: < 0 (or information unavailable)
	gr97	dstcode	Success =1 (Daylight Savings Time in effect) Success = 0 (Daylight Savings Time not in effect)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

If the result returned in **gr96** (*zonecode*) contains a value greater than 1,440 (60 minutes x 24 hours), then 1,440 should be subtracted from the result, which relates to minutes east of Greenwich.

Example Call

```
timzone:  .word   0
dstflag:  .word   0
          const   gr121,67        ;service = 67
          asneq   69,gr1,gr1      ;call the OS
          jmpf    gr121,tz_err    ;jump if error
          const   lr2,timzone     ;the address to
          consth  lr2,timzone     ;store timezone
          store   0,0,gr96,lr2    ;store the timezone
                                  ;correction
          const   lr2,dstflag     ;the address to store
                                  ;daylight savings
          consth  lr2,dstflag
          store   0,0,gr97,lr2    ;store the daylight
                                  ;savings flag
```

In the example call, the **gettz** service is called to access the current time zone correction value. Upon return, *gr121* is tested to determine if the service was successful. If not, the program jumps to an error handling routine called **tz_err**. If the service was successful, the values returned in *gr96* and *gr97* are stored in local variables called **timzone** and **dstflag**, respectively.

Service 257 – sysalloc
Allocate Memory Space

Description

This service allocates a specified number of contiguous bytes from the operating-system-maintained heap and returns a pointer to the base of the allocated block. *lr2* contains the number of bytes requested. If the storage is successfully allocated, *gr96* contains a pointer to the block; otherwise, *gr121* contains an error code indicating the reason for the call failure.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	257 (0x101)	Service number
	lr2	nbytes	Number of bytes requested
Returns:	*gr96*	addrptr	Success: pointer to allocated bytes Failure: = 0 (NULL pointer)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
blkptr:  .word   0
         const   lr2,1200          ;request 1200 bytes
         const   gr121,257         ;service = 257
         asneq   69,gr1,gr1        ;call the OS
         jmpf    gr121,alloc_err   ;jump if error
         const   gr120,blkptr      ;set address to store
         consth  gr120,blkptr      ;pointer
         store   0,0,gr96,gr120    ;store the pointer
```

The example call requests a block of 1200 contiguous bytes from the system heap. If the call is successful, the program stores the pointer returned in *gr96* into a local variable called *blkptr*. If *gr121* contains a boolean FALSE value when the service returns, the program jumps to **alloc_err** to handle the error condition.

Service 258 – sysfree
Free Memory Space

Description

This service returns memory to the system starting at the address specified in *lr2*. *lr3* contains the number of bytes to be released. The pointer passed to the **sysfree** service in *lr2* and the byte count passed in *lr3* must match the address returned by a previous **sysalloc** service request for the identical number of bytes. No dynamic memory allocation structure is implied by this service. High-level language library functions such as **malloc()** and **free()** for the C language are required to manage random dynamic memory block allocation and deallocation, using the **sysalloc** and **sysfree** kernel functions as their basis.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	258 (0x102)	Service number
	lr2	addrptr	Starting address of area returned
	lr3	nbytes	Number of bytes to release
Returns:	*gr96*	retval	Success: = 0 Failure: <0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
blkptr:  .word    0
         const    gr120,blkptr    ;set address of previous
         consth   gr120,blkptr    ;block pointer
         load     0,0,lr2,gr120   ;fetch pointer to block
         const    lr3,1200        ;set number of bytes to
                                  ;release
         const    gr121,258       ;service = 258
         asneq    69,gr1,gr1      ;call the OS
         jmpf     gr121,free_err  ;jump if error
         nop
```

The example calls **sysfree** to deallocate 1200 bytes of contiguous memory, beginning at the address stored in the *blkptr* variable. If the call is successful, the program continues; otherwise, if the return value in *gr121* is FALSE, the program jumps to **free_err** to handle the error condition.

Service 259 – getpsize
Return Memory Page Size

Description

This service returns, in register *gr96*, the page size (in bytes) used by the memory system of the HIF implementation.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	259 (0x103)	Service number
Returns:	*gr96*	pagesize	Success: memory page size, one of the following: 1024,2048,4096 and 8192 Failure: <0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
pagsiz:  .word    0

         const    gr121,259       ;service = 259
         asneq    69,gr1,gr1      ;call the OS
         jmpf     gr121,pag_err   ;jump if error
         const    gr120,pagsiz    ;set address to
         consth   gr120,pagsiz    ;store the page size
         store    0,0,gr96,gr120  ;store it!
```

The example calls the operating system kernel to return the page size used by the virtual memory system. If the call was successful, *gr121* will contain a boolean TRUE result and the program will store the value in *gr96* into the *pagsiz* variable; otherwise, a boolean FALSE is returned in *gr121*. In this case, the program will jump to **pag_err** to handle the error condition.

Service 260 – getargs Return Base Address

Description

This service returns the base address of the command-line-argument vector, *argv*, in register *gr96*, as constructed by the operating-system kernel when an application program is invoked.

Arguments are stored by the operating system as a series of NULL-terminated character strings. A pointer containing the address of each string is stored in an array whose base address (referred to as *argv*) is returned by the **getargs** HIF service. The last entry in the array contains a NULL pointer (an address consisting of all zero bits). The number of arguments can be computed by counting the number of pointers in the array, using the fact that the NULL pointer terminates the list.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	260 (0x104)	Service number
Returns:	*gr96*	baseaddr	Success: base address of argv Failure: 0 (NULL pointer)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
argptr:   .word    0
          const    gr121,260        ;service = 260
          asneq    69,gr1,gr1       ;call the OS
          jmpf     gr121,bas_err    ;jump if error
          const    gr120,argptr     ;set address where base
          consth   gr120,argptr     ;pointer is to be stored
          store    0,0,gr96,gr120   ;store the pointer
```

The example calls operating-system service 260 to access the command-line-argument vector address. If the service executes without error, the program continues by storing the argument vector address in the variable *basptr*. If *gr121* contains a boolean FALSE value upon return, the program jumps to **bas_err** to handle the error condition.

Service 273 – clock
Return Time in Milliseconds

Description

This service returns the elapsed processor time in milliseconds. Operating system initialization procedures set this value to zero on startup. Successive calls to this service return times that can be arithmetically subtracted to accurately measure time intervals.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	273 (0x111)	Service number
Returns:	*gr96*	msecs	Success: ≠0 (time in milliseconds) Failure: =0 (NULL pointer)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
pagsiz:   .word    0
          const    gr121,273        ;service = 273
          asneq    69,gr1,gr1       ;call the OS
          jmpf     gr121,clk_err    ;jump if error
          const    gr120,time       ;set address where
          consth   gr120,time       ;time is to be stored
          store    0,0,gr96,gr120   ;store the time in ms.
```

The example calls the operating system kernel to get the current value of the system clock in milliseconds. On return, if *gr121* contains a boolean FALSE value, the program jumps to **clk_err** to handle the error; otherwise, the time in milliseconds is stored in the variable *time*.

The return value from the clock service does not include system I/O data-transfer time incurred by HIF services with service numbers less than 256. The return value is related to the value returned by the cycles service, in that it is derived from the processor cycles counter, but scaled by the processor frequency and resolved to milliseconds.

Service 274 – cycles
Return Processor Cycles

Description

This service returns an ascending positive number in registers *gr96* and *gr97* that is the number of processor cycles that have elapsed since the last processor initialization was applied to the CPU. It provides a mechanism for user programs to access the contents of the internal Am29000 processor timer counter register. The cycle count can be multiplied by the speed of the processor clock to convert it to a time value. *gr97* will contain the most significant bits of the cycle count, while *gr96* will contain the least significant bits. HIF implementations of this service are required to provide a cycle count with a minimum of 42 bits of precision.

The implementor of this HIF service must, as best possible, eliminate system I/O data transfer time incurred by HIF services with service numbers less than 256. This will benefit the user when using this service to perform benchmarks across different hardware platforms. The user of this service should be aware that the return value may still contain cycles used in support of operating system tasks.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	274 (0x112)	Service number
Returns:	*gr96*	cycles	Success: Bits 0–31 of processor cycles Failure: = 0 (in both *gr96* and *gr97*)
	gr97	cycles	Success: Bits 32 and higher of processor cycles Failure: = 0 (in both *gr96* and *gr97*)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
cycles:   .word     0               ;MSb of cycles
          .word     0               ;LSb of cycles

          const     gr121,274       ;service = 274
          asneq     69,gr1,gr1      ;call the OS
          jmpf      gr121,cyc_err   ;jump if error
          const     gr120,cycles    ;set the address where
          consth    gr120,cycles    ;the count is to be
                                    ;stored
          store     0,0,gr97,gr120  ;store the MSb,
          add       gr120,gr120,4   ;increment the address,
          store     0,0,gr96,gr120  ;then store the LSb of
                                    ;cycles.
```

The example-call program fragment calls the operating-system service 274 to access the number of CPU cycles that have elapsed since processor initialization. The cycle count (in *gr96* and *gr97*) is stored in the two words addressed by the variable *cycles* if the service call is successful. If *gr121* contains a boolean FALSE value on exit, the program jumps to **cyc_err** to handle the error condition.

Service 289 – setvec
Set Trap Address

Description

This service sets the address for user-level trap handler services that implement the local register stack spill and fill traps. In addition, if the current HIF implementation supports program calls to set other trap vectors, this service provides that capability. It returns an indication of success or failure in register *gr121*. The method used to invoke these traps in user mode is described on page 2–6 in the User-Mode Traps section.

The only vectors supported by this specification are 64 (spill) and 65 (fill). These vectors are invoked by operating system software using the trampoline principles described in the section User-Mode Traps, and are not supported by the Am29000 processor hardware.

Extensions to this service, in implementations that support setting traps other than spill and fill, will return the previously installed trap address in register *gr96*, if the service is successful. For User Mode Traps, register *gr96* reports only the success or failure of the service. In HIF implementations where the extended **setvec** service is available, programs can use the returned (previous) vector address to implement vector chaining.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	289 (0x121)	Service number
	lr2	trapno	trap number
	lr3	funaddr	address of trap handler
Returns:	*gr96*	trapaddr	For user mode traps: Success: =0 Failure: <0 For extended trap vectors: Success: previous trap address Failure: =0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
trpadr:   .word     0
          const     lr2,64          ;trap number = 64
          const     lr3,t64_hnd     ;set address of
          consth    lr3,t64_hnd     ;trap-64 handler
          const     gr121,289       ;service = 289
          asneq     69,gr1,gr1      ;call the OS
          jmpf      gr121,vec_err   ;jump if error
          const     gr120,trpadr    ;set address where to
          consth    gr120,trpadr    ;store the trap address
          store     0,0,gr96,gr120  ;and store it!
```

The example calls the **setvec** service to pass the address to be used for the trap 64 trap handler routine. If the service returns with *gr121* containing a boolean TRUE result, the program continues by storing the trap address returned in *gr96*; otherwise, the program jumps to **vec_err** to handle the error condition.

Service 290 – settrap
Set Trap Vector

Description

This service provides the means to install trap-handler addresses directly into the vector table whose base address is pointed to by the Vector Area Base Address special-purpose register (VAB). The vector numbers that may legally be modified by this service are implementation dependent.

Implementations that do not intend to provide the ability to set trap addresses with this service should return the EHIFNOTAVAIL error code when this service is invoked. If certain vectors are restricted from being set by this service, the implementation should check the **trapno** parameter and return the EHIFNOTAVAIL error code for references to restricted trap vectors.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	290 (0x122)	Service number
	lr2	trapno	Vector number
	lr3	trapaddr	Address of trap handler
Returns:	*gr96*	trapaddr	Address of previous trap handler
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number: EHIFNOTAVAIL if service not available (implementation dependent)

Example Call

```
oldtrap:  .word     0                ;placeholder for old
                                     ;trap address
          const     lr2,54           ;floating divide trap
                                     ;vector (V_FDIV)
          const     lr3,new_div      ;set new_div as the
          consth    lr3,new_div      ;trap handler address
          const     gr121,290        ;service = 289
          asneq     69,gr1,gr1       ;call the OS
          jmpf      gr121,trap_err   ;jump if error
          const     gr120,oldtrap    ;set address for saving
          consth    gr120,oldtrap    ;the old trap handler
                                     ;address
          store     0,0,gr96,gr120   ;save the old handler
                                     ;address
```

In the example call, a new handler for the floating-point division operation is being installed. If the implementation returns an error, the program jumps to the **trap_err** label. If the service was successful and a new trap handler was installed, the previous handler address (if any) is stored into the **oldtrap** variable.

There is often a need for programs operating on dedicated hardware to enter supervisor mode. This can be accomplished by reserving a trap vector for that purpose and installing a trap-handler routine to return control to the user in supervisor mode. The operation is effected by issuing an assert instruction that invokes the specified trap. User mode can be restored by clearing (setting to 0) the Supervisor Mode bit (4) of the Current Processor Status register (CPS).

Service 291 – setim
Set Interrupt Mask

Description

This service provides the means to set the interrupt mask (IM) field and the disable interrupts (DI) field of the current processor status register (CPS). This field enables the external interrupt pins $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$, according to the following encoding:

00 $\overline{\text{INTR0}}$ enabled
01 $\overline{\text{INTR1}}$–$\overline{\text{INTR0}}$ enabled
10 $\overline{\text{INTR2}}$–$\overline{\text{INTR0}}$ enabled
11 $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ enabled

These two bits provide for a priority-oriented enabling capability; however, the $\overline{\text{INTR0}}$ interrupt cannot be disabled through the IM field alone. The disable interrupts (*di*) parameter must be set to 1 to produce this effect. A *di* value of 0 enables the selected interrupts, and a value of 2 leaves the *di*-bit of the CPS unchanged. If this service is not implemented, an error code of EHIFNOTAVAIL should be returned by the software. The error code for an illegal value in registers *lr2* or *lr3* is implementation dependent.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	291 (0x123)	Service number
	lr2	mask	New mask field value
	lr3	di	0 = Enable interrupts 1 = Disable interrupts 2 = Leave interrupt enable unchanged
Returns:	*gr96*	mask	Old mask field value
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number: EHIFNOTAVAIL if service not available (implementation dependent)

Example Call

```
oldmask:  .word     0                ;placeholder for old
                                     ;mask field value
          const     lr2,0x10         ;mask = 10 (*INTR(2:0)
                                     ;enable)
          const     lr3,0x0          ;enable interrupts
                                     ;(di = 0)
          const     gr121,291        ;service = 291
          asneq     69,gr1,gr1       ;call the OS
          jmpf      gr121,mask_err   ;jump if error
          const     gr120,oldmask    ;set address for saving
          consth    gr120,oldmask    ;the old IM field value
          store     0,0,gr96,gr120   ;save the oldIM field
                                     ;value
```

In the example call, the IM field of the current processor status register is to be set to 10, enabling external interrupt pins $\overline{\text{INTR0}}$, $\overline{\text{INTR1}}$, and $\overline{\text{INTR2}}$. If this service is not available, or if the value in *lr2* is illegal, the service will return an error code, in which case the program jumps to the **mask_err** label. If the service execution is successful, the previous contents of the IM field are stored in the **oldmask** variable.

Service 305 – query Return Version Information

Description

This service returns version information, or capabilities of the HIF implementation, as requested. On entry, the requested capability is passed as an argument in *lr2*. The service returns the requested information or indicates that it is unavailable in *gr96*.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	305 (0x131)	Service number
	lr2	capcode	Capabilities code 0 = Request HIF version 1 = Request CPU version and family code 2 = Request Am29027™ processor arithmetic accelerator version 3 = Request CPU clock frequency 4 = Request memory environment
For *lr2*=0 (HIF version)			
Returns:	*gr96*	hifvers	Success: ≥0 (encoded version information). The version number is returned as two 4-bit fields in the low-order 8 bits of the return value. The two fields are separated by an implied decimal point (e.g., 0x20 means HIF V2.0). Failure: <0 (or unavailable)
For *lr2*=1 (CPU version and family code)			
Returns:	*gr96*	cpuvers	Success: ≥0 (encoded version/family). The high-order 8 bits of the configuration register (CFG), known as the processor release level (PRL), are moved to the low-order 8 bits of *gr96*, as two 4-bit fields. Failure: <0 (or unavailable)

Type	Regs	Contents	Description
For *lr2*=2 (Am29027 version)			
Returns:	*gr96*	027vers	Success: ≥0 (encoded version information). The high-order 8 bits of the accelerator's precision register form the arithmetic accelerator release level (ARL) and are moved to the low-order 8 bits of *gr96*, as two 4-bit fields. Failure: <0 (or unavailable)
For *lr2*=3 (CPU clock frequency)			
Returns:	*gr96*	clkfreq	Success: >0 (frequency in Hertz) Failure: =0 (or unavailable)
For *lr2*=4 (Memory environment)			
Returns:	*gr96*	memenv	Success: >0 (memory environment) BYTEW 0x1 byte-write available DWSE 0x2 DW-bit set IREAD 0x4 Instruction memory readable Failure: ≤0 (or unavailable)
For all requests			
Returns:	*gr121*	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

In addition to the Return Usage table requests, negative **capcode** values in register *lr2* are available for implementation-dependent encoding of **query** requests. All positive values in register *lr2* are reserved for future expansion of the HIF **query** service.

Example Call

```
vers:     .word     0
          const     lr2,0           ;request HIF version
          const     gr121,305       ;service = 305
          asneq     69,gr1,gr1      ;call the OS
          jmpf      gr121,qry_err   ;handle query error
          const     lr2,vers        ;address to store
          consth    lr2,vers        ;version info
          store     0,0,gr96,lr2    ;store the HIF version
                                    ;number
```

In the example call, a request code of 0 is loaded into *lr2* and the service is called. Upon return, if the value in *gr121* is FALSE, indicating failure, the program jumps to an error routine. If *gr121* is TRUE, then the program stores the returned HIF version information into the variable called **vers**.

Service 321 – signal
Register Signal Handler

Description

This service provides the means to register (or un-register) a specified user signal handler. Local register *lr2* contains the address of the user signal-handler routine on entry. This routine is expected to handle the signals shown in Table 3–6.

Table 3–6. Signals Handled

Mnemonic	Value	Description
SIGINT	2	User interrupt (e.g., from keyboard)
SIGFPE	8	Floating-point exception

The HIF service returns the address of the previously installed handler in *gr96*. If no previous handler was installed, *gr96* will contain a NULL pointer (*gr96* = 0). Signal handlers may perform any appropriate processing, but only the services with service numbers above 256 are guaranteed to be available. Calls to services with numbers below 256 may result in unpredictable behavior when returning to the interrupted program—unless the service executes a **longjump()**, which avoids execution of the interrupt return service.

To un-register a signal handler, local register *lr2* must contain a value of 0 (NULL) on entry. When a handler is un-registered in this manner, signal handling will revert to the default behavior established by the operating system.

When one of the (SIGINT or SIGFPE) signals occurs, the HIF implementation must preserve the signal number that occurred; the register stack pointer (*gr1*); the register allocate bounds (*gr126*); the program counters, PC0–PC2; the channel registers (*CHA*, *CHD*, and *CHC*); the ALU register; the old processor status (OPS); and the contents of *gr121*. These registers are saved in the user memory stack. The HIF implementation must be careful not to disturb values in registers that have not been saved on the user's stack. Global register *gr125* should contain the address of the last saved value in the HIF Signal Stack (e.g., *gr121*) at the conclusion of this phase. Figure 3–1 illustrates the required user stack format for saved registers.

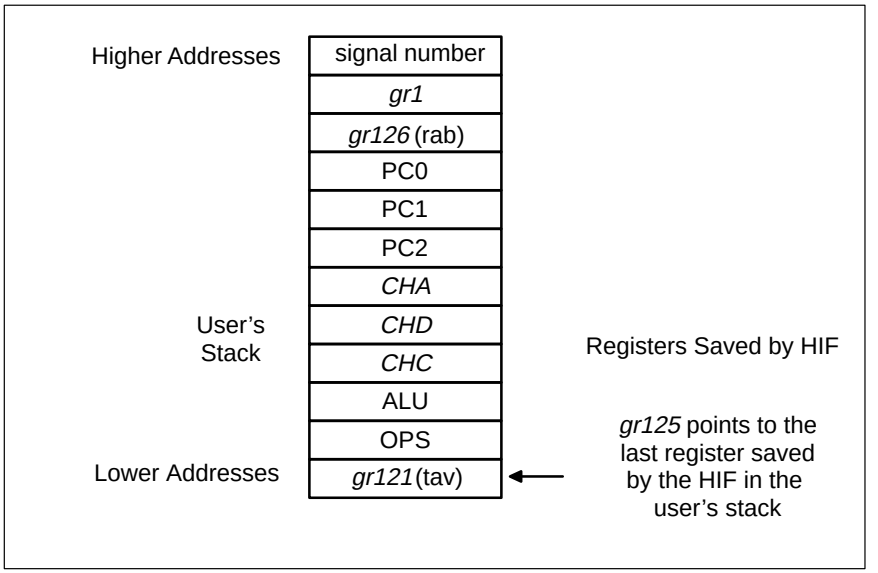

Figure 3–1. HIF Register Preservation

At this point the execution of the HIF invokes the handler specified by the **newsig** parameter to the **signal** service. The handler is invoked with the processor mode set to the mode of the interrupted program (either user or supervisor mode). Depending on the nature of the interrupt (SIGINT or SIGFPE) and the complexity of the handler, additional registers may need to be saved. In this case, the handler must preserve the values in the indirect pointers IPA, IPB, and IPC; the contents of the Q register; the stack frame pointer, *lr1*; and the local register stack free bounds in *rfb* (*gr127*). In addition, because high-level languages use global registers *gr96–gr124* as temporaries, the user signal handler may have to save these as well.

User signal handlers can be grouped into three levels of complexity, depending on the implementation:

- The least complex are handlers that have no intention of returning control to the user. In this case, only a few additional registers may need to be saved (if any).
- Floating-point error handlers are often more complex, where some of the user's context needs to be saved. In this case, probably only the indirect pointers (IPA–IPC), the Q register, and *gr125* need be preserved. After the error has been handled, the handler will issue one of the signal return services listed in Table 3–7 to return control to the user's program.

- The most complex handlers will be those needing to return to the user program at the C level of context. If the handler intends to pass control to a user-provided signal routine (e.g., to handle SIGINT), then it may be necessary to preserve all the registers indicated in Figure 3–1. In addition, handlers intending to return control at the C level of context will need to make a provision for completing any interrupted SPILL or FILL operations or complete a long-jump that may be in progress. Fortunately, AMD supplies the necessary code in library routines supplied with most tool products.

Before execution of the signal handler, the HIF is responsible for clearing the Channel Control (CHC) register (setting it to 0), to prevent restarting a load or store multiple operation that may have been interrupted. The proper contents of this register will be restored by the HIF when the handler issues one of the service requests listed in Table 3–7.

Table 3–7. Signal Return Services

Service	Name	Description
322	sigdfl	Perform default signal handling
323	sigret	Return to location indicated by PC1
324	sigrep	Return to location indicated by PC2
325	sigskp	Return to location indicated by PC0

Once a signal handler is invoked by one of the signals listed in Table 3–6, and when it has finished, it will usually return to the HIF by invoking one of the signal return services shown in Table 3–7, with register *gr125* pointing to the last saved register in the HIF-saved registers (i.e., *gr121*), as shown in Figure 3–1. More complex implementations may make other arrangements for returning to the user program's context. Sample code for saving and restoring the necessary registers is included in AMD development tool products.

The handler is responsible for determining the appropriate action for each type of interrupt (SIGINT or SIGFPE) and must return control to the HIF using one of the services listed in Table 3–7, after first restoring the indirect pointers (IPA–IPC), the Q register, and with *gr125* pointing to the last saved register in the user's stack (assuming the suggested approach for preserving registers is followed).

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	321 (0x141)	Service number
	lr2	newsig	Address if signal handler, or NULL pointer
Returns:	*gr96*	oldsig	Old handler address
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
oldhdlr: .word    0
         const    lr2,user_sigs   ;address of user signal
         consth   lr2,user_sigs   ;handler
         const    gr121,321       ;service = 321
         asneq    69,gr1,gr1      ;call the OS to install
                                  ;the handler
         jmpf     gr121,sig_err   ;jump to handle error
         const    gr120,oldhdlr   ;set address to store
         consth   gr120,oldhdlr   ;old handler address
         store    0,0,gr96,gr120  ;store the old handler
                                  ;address
```

In the example call, a user signal handler whose entry-point name is **user_sigs** is installed. When the service returns, if *gr121* contains a FALSE value, the program jumps to an error routine; otherwise, the address of the previously installed handler returned in *gr96* is stored in the local variable **oldhdlr**.

Service 322 – sigdfl
Perform Default Signal Action

Description

This service is called only from within a user signal handler installed using the **signal** (321) service. The function of this service is to instruct the HIF to perform the predetermined default action for the specified signal. The operating system is responsible for establishing the appropriate default action.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	322 (0x142)	Service number
	gr125	sigptr	Pointer to HIF Signal Stack containing preserved registers (See **signal** (321) for further information)
Returns:	Does not return		

Example Call

```
        const   gr121,322       ;service = 322
        asneq   69,gr1,gr1      ;call the OS
```

Since this service does not return, no attempt is made to test returned values or to store results.

Service 323 – sigret
Return From Signal Interrupt

Description

This service is called only from within a user signal handler installed using the **signal** (321) service. The function of this service is to return from the latest signal interrupt to the location specified by the value in program counter PC1 at the time the signal occurred. Once invoked, this service does not return to the user signal handler.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	323 (0x143)	Service number
	gr125	sigptr	Pointer to HIF Signal Stack containing preserved registers (See **signal** (321) for further information)
Returns:	Does not return		

Example Call

```
        const   gr121,323       ;service = 322
        asneq   69,gr1,gr1      ;call the OS
```

Since this service does not return, no attempt is made to test returned values or to store results.

Service 324 – sigrep
Return From Signal Interrupt

Description

This service is called only from within a user signal handler installed using the **signal** (321) service. The function of this service is to return from the latest signal interrupt to the location specified by the value in program counter PC2 at the time the signal occurred. Once invoked, this service does not return to the user signal handler.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	324 (0x144)	Service number
	gr125	sigptr	Pointer to HIF Signal Stack containing preserved registers (See **signal** (321) for further information)
Returns:	Does not return		

Example Call

```
        const   gr121,324       ;service = 324
        asneq   69,gr1,gr1      ;call the OS
```

Since this service does not return, no attempt is made to test returned values or to store results.

Service 325 – sigskp
Return From Signal Interrupt

Description

This service is called only from within a user signal handler installed using the **signal** (321) service. The function of this service is to return from the latest signal interrupt to the location specified by the value in program counter PC0 at the time the signal occurred. Once invoked, this service does not return to the user signal handler.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	325 (0x145)	Service number
	gr125	sigptr	Pointer to HIF Signal Stack containing preserved registers (See **signal** (321) for further information)
Returns:	Does not return		

Example Call

```
        const   gr121,325       ;service = 325
        asneq   69,gr1,gr1      ;call the OS
```

Since this service does not return, no attempt is made to test returned values or to store results.

Service 326 – sendsig
Send Signal

Description

This service provides the means to send a signal to the current process to support signal testing. A single parameter, *sig*, specifies the signal number to be sent.

Register Usage

Type	Regs	Contents	Description
Calling:	*gr121*	326 (0x141)	Service number
	lr2	sig	Signal number to be sent to current process
Returns:	*gr121*	0x80000000	Logical TRUE, service successful
		errcode	Error number, service not successful EHIFNOTAVAIL if service not implemented (implementation dependent)

Example Call

```
        const   lr2,SIGFPE      ;floating-point excep-
                                ;tion
        const   gr121,326       ;service = 326
        asneq   69,gr1,gr1      ;call the OS
        jmpf    gr121,send_err  ;handle signaling error
        nop
```

In the above example, a floating-point exception error signal is being sent to the current process. It is assumed that a signal handler for the SIGFPE (floating-point exception) error has been previously installed (see **signal** service) and is being tested.

Chapter 4

Process Environment

There are standard memory and register initializations that must be performed by a HIF-conforming kernel before entry to a user program. In C-language programs, this is usually performed by the module **crt0**. This module receives control when an application program is invoked, and executes prior to invocation of the user's main function. Other high-level languages have similar modules.

Startup Initialization

Initialization procedures must establish appropriate values for the general registers mentioned below. In addition, file descriptors for the standard input and output devices must be opened.

Register Stack Pointer (*gr1*)

The register stack pointer (*rsp*) register contains the main memory address in which the local register *lr0* will be saved, and from which it will be restored. The content of *rsp* is compared to the content of *rab* to determine when it is necessary to spill part of the local register stack to memory. On startup, the values in *rab*, *rsp*, and *rfb* should be initialized to prevent a spill trap from occurring on entry to the **crt0** code, as shown by the following relations:

$256 + rab \leq rsp < rfb$

$rfb = rab + 512$

This provides the **crt0** code with at least 64 registers on entry, which should be a sufficient number to accomplish its purpose. Before entering **crt0**, the startup initialization code must load the Am29027 processor's mode register value into global registers *gr96* and *gr97*. Register *gr96* contains the most significant half of the mode register value, and *gr97* contains the least significant half.

Memory Stack Pointer (*gr125*)

The memory stack pointer (*msp*) register points to the top of the memory stack, or the lowest addressed entry on the memory stack. This register must be preserved (or, more conventionally, restored).

Register Allocate Bound (*gr126*)

The register allocate bound (*rab*) register contains the register stack address of the lowest addressed word contained within the register file. *rab* is referenced in the prologue of most user program functions to determine whether a register spill operation is necessary to accommodate the local register requirements of the called function.

Register Free Bound (*gr127*)

The register free bound (*rfb*) register contains the register stack address of the lowest addressed word not contained within the register file (and greater than *rab*). *rfb* is referenced in the epilogue of most user program functions to determine whether a register fill operation is necessary to restore previously spilled registers needed by the function's caller.

Open File Descriptors

File descriptor 0 (corresponding to the standard input device) must be opened for text mode input. File descriptors 1 and 2 (corresponding to standard output and standard error devices) must be opened for text mode output prior to entry to the user's program. File descriptors 0, 1, and 2 are expected to be in COOKED mode (see **ioctl**), and file descriptor 0 should also select ECHO mode, so that input from the standard input device (**stdin**) is echoed to the standard output device (**stdout**).

Stack Allocation Sizes

The recommended minimum allocation sizes for the Memory and Register stacks are 6 Kb and 2 Kb, respectively. It is the responsibility of the HIF implementation to prepare the corresponding support registers for these minimum sizes.

Program Termination

The only valid way for an application to terminate execution is by calling the **exit** service. Most high-level languages provide this capability, even if the programmer does not explicitly invoke a corresponding library function.

Trap Handlers

The trap vector entries shown in Table 4–1 must be installed and corresponding handlers must be provided. All HIF-conforming operating systems must provide unaligned access trap handlers.

Table 4–1. Trap Handler Vectors

Trap	Description
32	MULTIPLY
33	DIVIDE
34	MULTIPLU
35	DIVIDU
36	CONVERT
42	FEQ
43	DEQ
44	FGT
45	DGT
46	FGE
47	DGE
48	FADD
49	DADD
50	FSUB
51	DSUB
52	FMUL
53	DMUL
54	FDIV
55	DDIV
64	Spill (Set up by the user's task through a **setvec** call)
65	Fill (Set up by the user's task through a **setvec** call)
69	HIF System Call

Note: The **Spill** (64) and **Fill** (65) traps are returned to the user's code to perform the trap handling functions in user mode.

HIF-Conforming Application COFF Information

A HIF-conforming application binary file is relocatable; however, it is not necessary to implement a relocation capability in any COFF loader. Many HIF-environment support-tool developers may chose to relink portable HIF-conforming applications prior to their execution on the target hardware. Although portable HIF applications are relocatable, the relocation information should be restricted to entries that use the symbol table entry relating to the start of each section. As a result, there need only be one symbol table entry for each section. These restrictions reduce the link/load time and costs.

Appendix A

HIF Quick Reference

Table A-1 lists the HIF service calls, calling parameters, and the returned values. If a column entry is blank, the register is not used or is undefined. Table A-2 describes the parameters used in Table A-1.

Table A–1. HIF Service Calls

Service Title	Calling Parameters				Returned Values		
	gr121	lr2	lr3	lr4	gr96	gr97	gr121
exit	1	exitcode			Service does not return		
open	17	pathname	mode	pflag	fileno		errcode
close	18	fileno			retval		errcode
read	19	fileno	buffptr	nbytes	count		errcode
write	20	fileno	buffptr	nbytes	count		errcode
lseek	21	fileno	offset	orig	where		errcode
remove	22	pathname			retval		errcode
rename	23	oldfile	newfile		retval		errcode
ioctl	24	fileno	mode				errcode
iowait	25	fileno	mode		count		errcode
iostat	26	fileno			iostat		errcode
tmpnam	33	addrptr			filename		errcode
time	49				secs		errcode
getenv	65	name			addrptr		errcode
gettz	67				zonecode	dstcode	errcode
sysalloc	257	nbytes			addrptr		errcode
sysfree	258	addrptr	nbytes		retval		errcode
getpsize	259				pagesize		errcode
getargs	260				baseaddr		errcode

Service Title	Calling Parameters				Returned Values		
	gr121	lr2	lr3	lr4	gr96	gr97	gr121
clock	273				msecs		errcode
cycles	274				LSBs cycles	MSBs cycles	errcode
setvec	289	trapno	funaddr		trapaddr		errcode
settrap	290	trapno	trapaddr		trapaddr		errcode
setim	291	mask	di		mask		errcode
query	305	capcode			hifvers		errcode
		capcode			cpuvers		errcode
		capcode			027vers		errcode
		capcode			clkfreq		errcode
		capcode			memenv		errcode
signal	321	newsig			oldsig		errcode
sigdfl	322				Service does not return		
sigret	323				Service does not return		
sigrep	324				Service does not return		
sigskp	325				Service does not return		
sendsig	326	sig					errcode

Table A–2. Service Call Parameters

Parameter	Description
027vers	The version number of the installed Am29027 arithmetic accelerator chip (if any).
addrptr	A pointer to an allocated memory area, a command-line-argument array, a pathname buffer, or a NULL-terminated environment variable name string.
baseaddr	The base address of the command-line-argument vector returned by the **getargs** service.
buffptr	A pointer to the buffer area where data is to be read from or written to during the execution of I/O services, or the buffer area referenced by the **wait** service.
capcode	The capabilities request code passed to the **query** service. Code values are: 0 (request HIF version), 1 (request CPU version), 2 (request Am29027 arithmetic accelerator version), 3 (request CPU clock frequency), and 4 (request memory environment).
clkfreq	The CPU clock frequency (in Hertz) returned by the **query** service.
count	The number of bytes actually read from file or written to a file.
cpuvers	The CPU family and version number returned by the **query** service.
cycles	The number of processor cycles (returned value).
di	The disable interrupts parameter to the **setim** service.
dstcode	The daylight-savings-time-in-effect flag returned by the **gettz** service.
errcode	The error code returned by the service. These are usually the same as the codes returned in the UNIX *errno* variable. See Appendix B for a list of HIF error codes.
exitcode	The exit code of the application program.
filename	A pointer to a NULL-terminated ASCII string that contains the directory path of a temporary filename.
fileno	The file descriptor that is a small integer number. File descriptors 0, 1, and 2 are guaranteed to exist and correspond to open files on program entry (0 refers to the UNIX equivalent of stdin and is opened for input; 1 refers to the UNIX stdout and is opened for output; 2 refers to the UNIX stderr and is opened for output).
funaddr	A pointer to the address of a spill or fill handler passed to the **setvec** service.
hifvers	The version of the current HIF implementation returned by the **query** service.
iostat	The input/output status returned by the **iostat** service.

Parameter	Description
mask	The interrupt mask value passed to and returned by the **setim** service.
memenv	The memory environment returned by the **query** service.
mode	A series of option flags whose values represent the operation to be performed. Used in the **open**, **ioctl**, and **wait** services to specify the operating mode.
msecs	Milliseconds returned by the **clock** service.
name	A pointer to a NULL-terminated ASCII string that contains an environment variable name.
nbytes	The number of data bytes requested to be read from or written to a file, or the number of bytes to allocate or deallocate from the heap.
newfile	A pointer to a NULL-terminated ASCII string that contains the directory path of a new filename.
newsig	The address of the new user signal handler passed to the **signal** service.
offset	The number of bytes from a specified position (*orig*) in a file, passed to the **lseek** service.
oldfile	A pointer to NULL-terminated ASCII string that contains the directory path of the old filename.
oldsig	The address of the previous user signal handler returned by the **signal** service.
orig	A value of 0, 1, or 2 that refers to the beginning, the current position, or the position of the end of a file.
pagesize	The memory page size, in bytes, returned by the **getpsize** service.
pathname	A pointer to a NULL-terminated ASCII string that contains the directory path of a filename.
pflag	The UNIX file access permission codes passed to the **open** service.
retval	The return value that indicates success or failure.
secs	The seconds count returned by the **time** service.
sig	A signal number passed to the **sendsig** service.
sigptr	A pointer to the HIF signal stack containing preserved registers.
trapaddr	The trap address returned by the **setvec** and **settrap** services; a trap address passed to and returned by the **settrap** service.
trapno	The trap number passed to the **setvec** and **settrap** services.
where	The current position in a specified file returned by the **lseek** service.
zonecode	The time zone minutes correction value returned by the **gettz** service.

Appendix B

HIF Error Numbers

HIF implementations are required to return error codes when a requested operation is not possible. The codes from 0–10,000 are reserved for compatibility with current and future error return standards. The currently assigned codes and their meanings are shown in Table B–1. If a HIF implementation returns an error code in the range of 0–10,000, it must carry the identical meaning to the corresponding error code in this table. Error code values larger than 10,000 are available for implementation- specific errors.

Table B–1. HIF Error Numbers Assigned

Number	Error Name	Description
0		Not used.
1	EPERM	Not owner This error indicates an attempt to modify a file in some way forbidden except to its owner.
2	ENOENT	No such file or directory This error occurs when a filename is specified and the file should exist but does not, or when one of the directories in a pathname does not exist.
3	ESRCH	No such process The process or process group whose number was given does not exist, or any such process is already dead.
4	EINTR	Interrupted system call This error indicates that an asynchronous signal (such as interrupt or quit) that the user has elected to catch occurred during a system call.

Number	Error Name	Description
5	EIO	I/O error Some physical I/O error occurred during a read or write. This error may, in some cases, occur on a call following the one to which it actually applies.
6	ENXIO	No such device or address I/O on a special file refers to a subdevice that does not exist or is beyond the limits of the device.
7	E2BIG	Arg list is too long An argument list longer than 5120 bytes is presented to execve.
8	ENOEXEC	Exec format error A request is made to execute a file that, although it has the appropriate permissions, does not start with a valid magic number.
9	EBADF	Bad file number Either a file descriptor refers to no open file, or a read (write) request is made to a file that is open only for writing (reading).
10	ECHILD	No children Wait and the process has no living or unwaited-for children.
11	EAGAIN	No more processes In a fork, the system's process table is full, or the user is not allowed to create any more processes.
12	ENOMEM	Not enough memory During an execve or break, a program asks for more memory than the system is able to supply or else a process size limit would be exceeded.
13	EACCESS	Permission denied An attempt was made to access a file in a way forbidden by the protection system.
14	EFAULT	Bad address The system encountered a hardware fault in attempting to access the arguments of a system call.
15	ENOTBLK	Block device required A plain file was mentioned where a block device was required, such as in mount.

Number	Error Name	Description
16	EBUSY	Device busy An attempt was made to mount a device that was already mounted, or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, or active text segment).
17	EEXIST	File exists An existing file was mentioned in an inappropriate context (e.g., link).
18	EXDEV	Cross-device link A hard link to a file on another device was attempted.
19	ENODEV	No such device An attempt was made to apply an inappropriate system call to a device, (for example, to read a write-only device), or the device is not configured by the system.
20	ENOTDIR	Not a directory A nondirectory was specified where a directory is required, for example, in a pathname or as an argument to *chdir*.
21	EISDIR	Is a directory An attempt was made to write on a directory.
22	EINVAL	Invalid argument This error occurs when some invalid argument for the call is specified. For example, dismounting a nonmounted device, mentioning an unknown signal in **signal**, or specifying some other argument that is inappropriate for the call.
23	ENFILE	File table overflow The system's table of open files is full, and temporarily no more open requests can be accepted.
24	EMFILE	Too many open files The configuration limit on the number of simultaneously open files has been exceeded.
25	ENOTTY	Not a typewriter The file mentioned in **stty** or **gtty** is not a terminal or one of the other devices to which these calls apply.

Number	Error Name	Description
26	ETXTBSY	Text file busy The referenced text file is busy and the current request cannot be honored.
27	EFBIG	File too large The size of a file exceeded the maximum limit.
28	ENOSPC	No space left on device A write to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because no more disk blocks are available on the file system.
29	ESPIPE	Illegal seek A seek was issued to a socket or pipe. This error may also be issued for other nonseekable devices.
30	EROFS	Read-only file system An attempt to modify a file or directory was made on a device mounted read-only.
31	EMLINK	Too many links An attempt was made to establish a new link to the requested file and the limit of simultaneous links has been exceeded.
32	EPIPE	Broken pipe A write on a pipe or socket was attempted for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is caught or ignored.
33	EDOM	Argument too large The argument of a function in the math package is out of the domain of the function.
34	ERANGE	Result too large The value of a function in the math package is unrepresentable within machine precision.
35	EWOULDBLOCK	Operation would block An operation that would cause a process to block was attempted on an object in nonblocking mode.

Number	Error Name	Description
36	EINPROGRESS	Operation now in progress An operation that takes a long time to complete was attempted on a nonblocking object.
37	EALREADY	Operation already in progress An operation was attempted on a nonblocking object that already had an operation in progress.
38	ENOTSOCK	Socket-operation on nonsocket A socket-oriented operation was attempted on a nonsocket device.
39	EDESTADDRREQ	Destination address required A required address was omitted from an operation on a socket.
40	EMSGSIZE	Message too long A message sent on a socket was larger than the internal message buffer or some other network limit.
41	EPROTOTYPE	Protocol wrong type for socket A protocol was specified that does not support the semantics of the socket type requested.
42	ENOPROTOOPT	Option not supported by protocol A bad option or level was specified when accessing socket options.
43	EPROTONOSUPPORT	Protocol not supported The protocol has not been configured into the system, or no implementation for it exists.
44	ESOCKTNOSUPPORT	Socket type not supported The support for the socket type has not been configured into the system, or no implementation for it exists.
45	EOPNOTSUPP	Operation not supported on socket An example of this would be trying to accept a connection on a datagram socket.
46	EPFNOSUPPORT	Protocol family not supported The protocol family has not been configured into the system or no implementation for it exists.

Number	Error Name	Description
47	EAFNOSUPPORT	Address family not supported by protocol family An address was used that is incompatible with the requested protocol.
48	EADDRINUSE	Address already in use Only one usage of each address is normally permitted.
49	EADDRNOTAVAIL	Cannot assign requested address This normally results from an attempt to create a socket with an address not on this machine.
50	ENETDOWN	Network is down A socket operation encountered a dead network.
51	ENETUNREACH	Network is unreachable A socket operation was attempted to an unreachable network.
52	ENETRESET	Network dropped connection on reset The host the user was connected to crashed and rebooted.
53	ECONNABORTED	Software caused connection abort A connection abort was caused internal to the user's host machine.
54	ECONNRESET	Connection reset by peer A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.
55	ENOBUFS	No buffer space available An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.
56	EISCONN	Socket is already connected A connect request was made on an already connected socket; or a *sendto* or *sendmsg* request on a connected socket specified a destination when already connected.

Number	Error Name	Description
57	ENOTCONN	Socket is not connected A request to send or receive data was disallowed because the socket was not connected and (when sending on a datagram socket) no address was supplied.
58	ESHUTDOWN	Cannot send after socket shutdown A request to send data was disallowed because the socket had already been shut down with a previous shutdown call.
59	ETOOMANYREFS	Too many references; cannot splice.
60	ETIMEDOUT	Connection timed out A connect or send request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)
61	ECONNREFUSED	Connection refused No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.
62	ELOOP	Too many levels of symbolic links A pathname look-up involved more than the maximum limit of symbolic links.
63	ENAMETOOLONG	Filename too long A component of a pathname exceeded the maximum name length, or an entire pathname exceeded the maximum path length.
64	EHOSTDOWN	Host is down A socket operation failed because the destination host was down.
65	EHOSTUNREACH	Host is unreachable A socket operation was attempted to an unreachable host.
66	ENOTEMPTY	Directory not empty A nonempty directory was supplied to a *remove* directory or **rename** call.
67	EPROCLIM	Too many processes The limit of the total number of processes has been reached. No new processes can be created.

Number	Error Name	Description
68	EUSERS	Too many users The limit of the total number of users has been reached. No new users may access the system.
69	EDQUOT	Disk quota exceeded A write to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted; or the allocation of an *inode* for a newly created file failed because the user's quota of *inodes* was exhausted.
70	EVDBAD	RVD related disk error
1001	EHIFNOTAVAIL	HIF service not available. The requested HIF service is not implemented or is not available to the user program making the request.
1002	EHIFUNDEF	HIF service is undefined The HIF service referenced by the program is undefined. No valid HIF service with that service number exists.

Index

Numbers

A

B

C

D

L

M

N

O

P

Q

R

S

T

U

V

W