Mica Working Design Document
Condition, Exit, and AST Handling

Revision 0.7

2–December–1987

Issued by:
Robert Bismuth

digital™

TABLE OF CONTENTS

FIGURES

Revision History

Date	Revision Number	Author	Summary of Changes
June 8, 1986	0.0	Robert Bismuth	Original
June 26, 1986	0.1	Robert Bismuth	Incorporate review changes Expand mechanism vector Redefine DEPTH argument Change to unwinding to a depth only Expand system service names Remove FLBC displacement field definition Remove discussion of LIB$ ESTABLISH and REVERT Simplify exit handling algorithm Include solution to the WAITFOR problem
July 23, 1986	0.2	Robert Bismuth	Remove user-settable dispatch vector Change unwind: allow unwind from anywhere except during exit handling, also allow unwind to a target frame (if one exists) Change condition values to be pointers to records Change references to P.VMS to Mica, also change PVMS$ to exec$ Add second last chance handler for P.ULTRIX Document the usage of vectored handlers Add discussion of instruction emulation for partial instruction set machines Add discussion of vector instruction recovery Add entry to signal array to indicate if continue possible Add comments regarding reinvoke_handler bit in procedure entry description

Date	Revision Number	Author	Summary of Changes
Feb. 22, 1987	0.3	Robert Bismuth	Change the name "static handler" to "entry descriptor-based handler" Document the system catchall handler Remove query on restricting the uses of primary vectored and last chance handlers Document multiple unwind request behaviour Clean up the use of "exception" versus "condition" Remove section on condition and status values Complete the removal of user replacement of condition dispatcher procedures Change PLIB$RAISE_EXCEPTION to PLIB$RAISE_CONDITION Change exec$read_exception_vec to exec$read_vectored_hand Change exec$set_exception_vec to exec$establish_vectored_hand Replace the four vectored handlers with two lists of vectored handlers located in the TCR Add process-wide exit handlers Add optional record pointer argument for exit handlers Correct discussion of vector enable faults Remove all references to current-mode hardware exceptions Rename signal array/vector to condition array/vector Add optional condition array parameter to unwind Add discussion of user-mode ASTs and the ULTRIX Last Chance Handler Change P.TBD to Mica Replace PLIB$SIGNAL with exec$handler_dispatcher Replace PSS$_ with exec$_ Change unwind to not perform piecemeal stack cleanup Change unwind through ASTs to use a handler established when the AST was delivered Add exit-unwind support Add collided-unwind support Reformat condition and mechanism arrays (include typing condition arguments) Replace PLIB$ with exec$ Add service to return the entry descriptor of the next condition handler the handler dispatcher will find Add condition stack support and exec$switch_stack system service Add exec$get_next_condition_hand system service Update thread exit description Remove kernel-mode exit handlers Remove definition of exec$exit Add definition of exec$terminate Add e$user_exit support for clean exit handling algorithm and to avoid potential AST problems Correctly document e$/exec$ services with procedure variables in the PCR

Date	Revision Number	Author	Summary of Changes
May 15, 1987	0.4	Robert Bismuth	Remove SIL INTEGER status return Remove variables named "type" Change "array" to "record" for condition and mechanism data structures Expand "cond" abbreviation to "condition" Add discussion of asynchronous software conditions Reword vector enable fault discussion Add the return of the register contents in place of the faulting instruction for the FLBC fault Add discussion of binary compatibility for hardware exception status values Clean up discussion of default state of handlers with regard to being reinvocable
Sep. 19, 1987	0.5	Robert Bismuth	Remove ULTRIX references Remove variables named "record" Remove compatibility constraint with VMS for condition values Modify ULTRIX signal discussion to a general discussion of interthread condition raising Change record layouts to Pillar record definitions Modify exit handling to use data structures in the PCR and TCR from objects Remove VMS compatability bit from condition arrays Strengthen discussion around unwind exit versus straight exit Add Boolean to unwind for exit unwind Add bit to condition vector characteristics field to indicate unwind in progress status
Oct. 26, 1987	0.6	Robert Bismuth	Add User-Mode AST Handling section
Nov. 24, 1987	0.7	Robert Bismuth	Change unwind algorithm to single pass Change condition array Pillar definition to a listhead of linked condition vectors Correct typo in overview to give correct number of volatile registers State that modifying R8 and R9 in a mechanism array does in fact change the return value Add status flags to condition vectors: exec$c_condition_exit_unwind, exec$c_condition_during_ast, exec$c_condition_async Add register section to mechanism array: all of a handler's establisher's register context Modify condition dispatch algorithm to preserve all flags except the Continue flag of a primary condition vector Remove general unwind by depth: only depth allowed is to caller of establisher Add a Boolean argument to the Unwind system service to suppress automatic stack collapse at completion of unwind Add exec$change_unwind_vector system service for the debugger

CHAPTER 1

CONDITION, EXIT, AND AST HANDLING

1.1 Overview

This chapter describes four facilities: condition handling, unwinding, support for exit handlers, and support for user-mode asynchronous system trap (AST) handlers. The chapter specifies goals, interfaces, and algorithms for the areas. These facilities are all related and are designed to work together.

There are currently no outstanding issues for this chapter and the only planned modifications are those resulting from the last group-wide review, with the addition of a section on user-mode AST handlers—previously undocumented for Mica.

Each facility is described separately.

1.1.1 Condition Handling

A *condition* results from an error encountered during thread execution. It may be due to a hardware or software failure. Examples of such hardware errors are arithmetic traps, access violations, and so on. Examples of such software errors are range checking, argument checking, and so on. The Mica *condition handling facility* provides the capability for programs to process such conditions in a controlled fashion.

A *condition handler* is a procedure written as a part of a program or supplied by a run-time facility to handle conditions if they occur during the execution of that program. Should a condition occur during program execution, Mica must be able to find a thread's condition handlers. The condition handling facility provides the mechanism by which handlers are found and established (either at runtime or compile time).

When a condition occurs, it is said to be raised in the thread which caused it. Raising a condition interrupts the normal control flow in a thread, saves its context, and causes a search to be made for a condition handler established by the thread. If a handler is found, it is called as a procedure, with arguments describing the nature of the condition (the *condition record*) and the environment in which it occurred (the *mechanism record*).

A condition handler may choose to handle the condition (that is, perform some actions relating to the condition) or may choose to reraise the condition (normally done for conditions which that handler is not written to handle). In the second case, the search for handlers continues and the next handler found is called. This process continues until some handler either indicates that the thread should continue (either from the location of the condition or using the unwind facility from a different location) or causes the thread to exit, or until no more established handlers can be found. In this last case, the *system catchall handler* is called.

There are three types of condition handlers:

- Vectored handlers
- Invocation descriptor-based handlers

use this as part of new overview.

- The system catchall handler

There may be many vectored handlers or invocation descriptor-based handlers, none of which are supplied by Mica. There is only one system catchall handler, which is always provided by Mica.

Vectored handlers may only be established at runtime, by using a system service. There are two types of vectored handlers: primary and last chance. Primary vectored handlers are the first searched for when a condition is raised. The list of primary handlers is called in FIFO order with respect to when they were established. If all have been called and reraised, the invocation descriptor-based handlers are then called for currently active procedures, from the most recently active to the oldest. Finally, if all these reraise the condition, the last chance vectored handlers are called in LIFO order with respect to when they were established. Should all these reraise as well, then the system catchall handler is called, which produces an error message and causes the thread to exit.

Invocation descriptor-based handlers are established at compile time. They are located from a procedure's invocation descriptor. These handlers are used to implement a particular language's condition handling semantics. For the Pillar language, they are used to implement structured condition handling.

Mica condition handling is designed to allow the processing of nested conditions and also to handle boundary problems with stack limitations. It also provides the following additional features:

- Invocation descriptor-based handlers may be called multiple times when multiple conditions are active. This behavior may be enabled per handler. (Note that this is required for PL/1 support.)
- Environment information relating to a condition contains the set of scratch registers used in a PRISM procedure call, together with the stack pointer (SP) and frame pointer (FP) at the time of the condition (that is, registers R1 through R31, inclusive).
- Condition information is complete, including information relating to message files and argument typing.
- A separate stack is available for the execution of vectored handlers. This improves the capabilities of the Mica debugger.

1.1.2 Unwinding

The Mica *unwind facility* centrally provides the capability to perform nonlocal GOTOs within a thread. It is implemented as a user-mode procedure, mapped in system space, and reached via a procedure variable in the process control region (PCR).

A call to the Unwind service specifies a target procedure and point in that procedure from which to continue thread execution. The target procedure must be an ancestor of the calling procedure. A target procedure invocation is specified either by its stack frame pointer (procedure invocations without stack frames may not be unwound to), or as the caller of the establisher of the last active condition handler. A condition record may be specified along with this target to give information relating to why the unwind operation is taking place.

Prior to returning execution to the target procedure invocation, the unwind facility searches for and calls any invocation descriptor-based condition handlers established for any procedure invocations found between the calling procedure and the target procedure invocation. These are called with the condition record and a mechanism record (constructed by the Unwind service) relating to where the unwind request was made. This allows procedure invocations that are being discarded a chance to clean up; for example, to deallocate any virtual storage they have allocated.

Once this phase has completed, the target invocation's register context is restored and the execution is continued from the specific point. Note that in the case of an unwind after a condition handler has been active, R8 and R9 are restored from the mechanism record, allowing a return status to be set.

Since processes in Mica are multithreaded, it is necessary for each thread to clean up its use of the common address space. The unwind algorithm is designed to help this take place: instead of exiting a thread by using the Exit system service, a call to Unwind is made, specifying the beginning of the call hierarchy as the target. Unwind then calls all established invocation descriptor-based handlers, causing them all to clean up their own environments. When the beginning of the call hierarchy is reached, Unwind calls the thread Exit system service, with the input condition record argument as status.

Thus, in Mica, user-mode thread exit is accomplished using the unwind facility, not by using the thread Exit system service directly.

1.1.3 Exit Handling

The Mica *exit handling facility* allows threads and processes to perform overall clean-up actions on their environment or deallocation of system resources. *Exit handlers* are procedures established by a thread during execution and called in user mode after a thread has called the thread (or process) Exit system service. There is no way in Mica a thread or process can exit without attempting to call exit handlers.

There are two types of exit handlers: thread and process exit handlers. Thread exit handlers are called when a thread exits. Process exit handlers are called when the last thread in a process has finished executing the last of its thread exit handlers.

Exit handlers are established using a system service and kept as a list in either the PCR or the thread control region (TCR). This helps ensure that the exit handler list cannot be accidentally corrupted. The lists are called in LIFO order. Each list entry has a procedure variable and a nontyped 64-bit parameter, which may be used to pass information to the handler when it is activated during thread exit. Entries may only be removed by using the Establish Exit Handler system service with the appropriate arguments.

Exit handlers are called with the 64-bit, user-specified argument kept in the list entry and a condition record. This is the condition record that was used in the call to the thread Exit system service which activated the exit handler. An exit handler completes it processing by calling the thread Exit system service. Thus, each call to the Exit system service removes a handler from the list and calls it, until the list is empty, in which case the rest of thread rundown continues.

If, during the execution of an exit handler, a forced exit request is made for that thread, then the current exit handler is terminated, and the next one on the list is called. All handlers are allowed to run until they exhaust CPU quota. They may not establish new exit handlers. Should an exit handler exceed the thread's CPU quota, it is terminated. The thread's CPU quota is then incremented by a fixed amount and the next handler found and called.

Note that a forced exit request for a thread which is not executing exit handlers causes an exit unwind operation to occur prior to calling any exit handlers.

1.1.4 User-Mode AST Handling

The Mica user-mode *AST handling facility* provides a mechanism for delivering asynchronous event notification in user mode to threads. Many Mica system services have the capability of executing in parallel with a thread's execution and/or causing subsequent asynchronous event notification to the thread. Thus, a thread may issue a system service, the service may return with a pending status, and the thread may continue executing. When the service later completes the requested action or an event associated with that service occurs, if the thread established an AST handler in the service call, a user-mode AST is queued to the thread. The AST is delivered as soon as the thread is next eligible to run, unless an AST has already been delivered and is being processed by the thread, or if the thread has disabled the delivery of user-mode ASTs.

An *AST handler* is a procedure that is intended to receive such notification. These procedures are part of the program and are associated with a particular event or system service completion notification required by the thread during its execution. An AST cannot occur unless the thread has established an AST handler for it.

To establish an AST handler, a thread uses a procedure variable for the handler in the system service call that can cause the desired AST. Along with this procedure variable, the thread may specify a 64-bit quadword untyped parameter. When the procedure is subsequently called to process the AST, this parameter, together with an AST-specific, 64-bit quadword untyped parameter, is used as an input argument. These arguments are used to identify the AST and to pass information to the thread concerning the AST.

Once a user-mode AST has been delivered, no other user-mode ASTs can be delivered until it has finished being processed. Subsequent ASTs are blocked by hardware until the thread explicitly leaves AST state, thereby removing the block. The AST is delivered to system-supplied code in user mode. This procedure sets up a stack frame and then, in turn, calls the specified AST procedure with the AST parameters. When the AST procedure returns, the system procedure uses a system service to remove the AST In Progress flag for the thread and dismisses the AST state, allowing the delivery of further user-mode ASTs (if any are pending). The stack is then cleaned and an REI instruction used to continue the thread's previous execution.

Note that the "false" stack frame is important: it is used to provide continuity when attempting an unwind through an AST event. The system procedure has an invocation descriptor-based condition handler established specifically to deal with this possibility.

At any time, a thread may disable or enable the delivery of user-mode ASTs. This is accomplished using the SWASTEN instruction and does not involve any system services.

1.1.5 Dependencies

This chapter depends on the:

1. PRISM SRM—specifically, hardware exceptions
2. PRISM Calling Standard
3. Mica process architecture design
4. Mica kernel design

Note that the Mica debugger design depends on this chapter.

1.2 Condition Handling

1.2.1 Introduction

Mica condition handling provides a mechanism by which all error conditions encountered during a thread's execution may be reliably handled by the thread in a controlled manner.

A thread is notified of an error in the use of either hardware or software by raising a condition. When a condition is raised, the thread's execution is interrupted and the thread starts executing a system-supplied dispatch procedure, referred to as the *dispatcher*, which locates a condition handler. Note that the dispatcher executes as if it had been called immediately after the condition was raised.

A condition handler is a procedure that is supplied by the thread and is written to deal with conditions.

The dispatch procedure calls condition handlers by locating them using the algorithm specified in this chapter. Once located, handlers are called as procedures with arguments reflecting the nature of the condition and environment in the thread where the condition was raised. When a handler completes, it either returns with status to the dispatcher, starts unwinding procedure invocations to resume thread execution in a different procedure invocation than that during which the condition was raised, or forces the thread to exit.

If control is returned to the dispatcher, it may take one of two possible actions: look for another condition handler to call (reraise case) or continue the thread's execution from the original exception site (if possible).

Note that termination from within a condition handler always causes any exit handlers established by the thread to be called. Exit handlers are covered in Section 1.3.

1.2.2 Goals

This design assumes the overall PRISM architecture goals. The following are the specific design goals of condition handling for Mica:

1. Support the condition handling requirements specified in the PRISM Calling Standard.
2. Condition handling services are provided on a per-thread basis. Threads may not affect each other's condition handling.
3. Hardware exception and software condition handling are implemented via a reliable common software interface, using one central dispatch procedure for locating handlers.
4. The majority of processing occurs in the mode in which a condition occurs. (This includes searching for handlers, as described below.)
5. Provide a unified condition handling mechanism that may be used, if so desired, by code running either in kernel or user mode.
6. Provide a degree of programming compatibility with the VMS handling standards. That is, where possible, the majority of existing VMS applications should not need any logic or any drastic source changes to use Mica condition handling.

   (Note that there are some bounds on this goal due to the PRISM architecture. This is fully discussed in Section 1.2.4, below.)
7. Define a superset of VMS condition handling capabilities, removing some past restrictions and allowing the implementation of common language and system run-time libraries.
8. Provide a standard under which Pillar structured condition handling may be implemented.
9. Provide the means for developers outside the Mica development group to define, specify, and manage software conditions.

1.2.3 Classes of Conditions

There are two classes of conditions in a Mica system: *hardware conditions* and *software conditions*.

Hardware conditions occur when a thread attempts some action defined as incorrect, impossible, or not yet possible by the hardware. Such action results in a hardware exception interrupting execution, which in turn causes a condition to be raised in the thread that was executing.

Software conditions may be raised at any point during thread execution. This allows applications or language run-time libraries to notify threads that some action defined as incorrect, impossible, or not yet possible was attempted by the thread. For example, subscript checking or system service failures may be raised as software conditions.

\In Mica, software conditions may occur synchronously and asynchronously to thread execution. Mica has the ability for a thread in a process to raise a condition in a thread executing in a different or the same process using the Signal system services.\

1.2.3.1 Hardware Exceptions

As defined in the PRISM SRM, hardware exceptions fall into seven categories:

- Arithmetic traps
- Data alignment exceptions
- Faults occurring as a consequence of an instruction
- Memory management faults
- Serious system failures
- Stack alignment
- Vector exceptions

1.2.3.1.1 Invisible Exceptions

Some hardware exceptions are handled by the kernel and executive without notifying the condition handling services. These are:

- Translation not valid (page fault)
- Vector enable fault
- Vector restart fault
- Data alignment fault
- Reserved opcode where the instruction is emulated
- Vector memory access trap
- Serious system failures
- Kernel mode stack alignment abort

In all cases but the last two, the system takes care of performing any required operations, so that the original faulting instruction stream may continue as if no exception had occurred; that is, with no condition raised.

When executing on a PRISM processor which implements one of the possible instruction subsets specified in the SRM, the reserved opcode fault is used to enter instruction emulation code for missing instructions. In this case, this fault becomes invisible.

1.2.3.1.1.1 Vector Enable Fault

When a vector enable fault occurs, it is always delivered in kernel mode. If the previous access mode was kernel mode, then a system bugcheck occurs. If the previous mode was user mode, then the fault is used to either restore vector context or enter vector instruction emulation code.

Whenever a thread is placed in running context, the Vector Enable bit in its processor status (PS) is cleared. In a system configuration that has at least one processor with a vector unit, the fault is used to initially set a thread's processor affinity, to allocate a vector save area in the control space of the process, to initially clear the vector registers and subsequently, to load vector context after a thread is restored to running state and issues a vector instruction.

After this initial fault, whenever a thread causes a vector enable fault, a check is made to determine if this thread was the last thread to use the current processor's vector unit and that this thread has used no other vector unit since using this unit. If so, then the vector context in the vector unit is the context of this thread. The Vector Enable bit is set in the thread's PS and an REI to user mode is performed. The thread is then able to use the vector unit until preempted.

If this thread was not the last thread to use this vector unit, then the Vector Enable bit is set in the thread's PS and the thread's vector context is loaded into the vector unit. An REI to user mode is then performed.

If a thread using a vector unit is preempted by another thread, then the contents of the processor's vector registers are saved in the save area of the thread, and the thread's saved PS is modified: the Vector Enable bit is cleared to force a load or check of a vector unit the next time the thread runs and tries to execute a vector instruction.

\Note that in this way, when a thread is run on a processor, its vector context is never immediately restored. Rather, the Vector Enable bit in its PS is cleared to force the vector context loading on the first vector instruction issued. This saves unnecessary system overhead, since loading a vector unit is expensive and should only be done if needed during a thread's processor time-slice. However, when a thread is preempted, its vector context is always immediately saved.\

An attempt is always made to reschedule threads that use vector units on the same processor they ran on the last time their vector context was loaded.

For configurations in which no processors have vector units, the vector enable fault is used to provide an entry into vector instruction emulation code. On the first vector enable fault, a vector save area is allocated. However, in this case the save area is used to emulate the vector registers. Once the save area exists, the system enters the instruction emulation code in user mode with the address of the faulting instruction and the save area.

\The emulation code requires user-mode read/write access to the vector register save area. Once the faulting instruction has been emulated, the emulation code continues the original execution. Note that the Vector Enable bit in the thread's PS remains cleared so that all future vector instructions will use the same path to the emulation code.

This currently is a problem, since the save area is in the process control space.\

1.2.3.1.1.2 Vector Restart Fault

TBD

1.2.3.1.1.3 Reserved Opcode Fault

When a reserved opcode fault occurs, it is always delivered in kernel mode. The system control block (SCB) vector for this fault points to code in system space that examines the offending instruction and checks to see if it should be emulated for the current processor. If so, the emulation is carried out in kernel mode. If not, a hardware exception occurs and a condition is then raised in the executing thread.

1.2.3.1.2 Visible Exceptions

All remaining hardware exceptions cause a condition to be raised in the executing thread. This causes a search for a condition handler, and possible subsequent calls to condition handlers.

1.2.3.2 Software Conditions

Software conditions result from an explicit use of condition handling by a thread. In this system, they internally look and are handled exactly like a hardware exception. Software conditions are designed to allow program-detected conditions to use the same general handling mechanism as hardware-detected conditions.

1.2.3.2.1 FLBC Instruction Faults

The Fault On Bit Clear (FLBC) instruction is available as a general mechanism for software to declare a software condition without the usual overhead of setting up a procedure call. This technique is available to all language run-time environments, allowing an inexpensive entry to the condition handling subsystem, while at the same time possibly passing an argument. Thus, a hardware fault exception may be used to raise what are essentially software conditions.

If desired, this method may also be used to provide an unconditionally raised condition by generating the FLBC instruction with R0 as its register argument.

For example, threads are notified of any system service failure by the least significant bit of the return status code being clear. Thus, if a compiler detects that a thread is not examining a service's returned status, it may generate an FLBC instruction immediately following the call to a system service, with register R8 as an argument. If the system service fails, then on return, a fault occurs, a condition is raised, and condition handling is invoked.

1.2.3.2.2 FLBC Condition Argument

Use of the displacement argument is specified as a part of the PRISM Calling Standard.

To ease the use of this fault instruction for interpreting software conditions after they have been raised, the hardware stack parameters are modified by the interrupt code prior to translation into arguments for subsequent calls to handlers. The displacement is unpacked and delivered in place of the zero longword returned by the hardware, and the register contents are delivered in place of the faulting instruction.

This is the only hardware exception that has arguments modified prior to their delivery to condition handlers.

\Note that this is currently under review: a proposal is being considered whereby the original FLBC condition record would be replaced by a condition record corresponding to the information given by the displacement and register contents. The condition handler dispatcher would initiate the replacement.

If this proposal is accepted, this chapter will be modified to reflect it.\

1.2.4 PRISM Architectural Restrictions on Compatibility

Since PRISM and VAX processor architectures differ, there are some constraints placed on VMS (and ULTRIX) compatibility. \Note that full compatibility with either of these operating systems is not a goal of this design.\

1.2.4.3 Continuing Execution after Raising Conditions

The PRISM architecture neither guarantees that instructions are completed in the same order in which they were fetched from memory or that instruction execution is strictly sequential. However, continuation after most hardware exceptions is possible.

Arithmetic traps cannot be restarted, since they are instruction aborts and not all information is stored for a restart. The only way in which software may guarantee the ability to continue from this type of exception is by placing DRAIN instructions around an exception site. While this solves the continuation problem, it is neither practical nor desirable because of the negative effects on processor performance.

User stack alignment abort also does not save enough information to allow a restart or continuation. Unlike the arithmetic case, there is no way to avoid this.

Software conditions are, by definition, synchronous with the instruction stream and possess, if needed, a well-defined continuation point. Thus, a handler may have the option of requesting continuation from a software condition. However, since compiler-generated code at any instant relies on previous error-free execution, continuing from a software condition may in general result in unpredictable

behavior, unless the handler has explicitly fixed the cause of the condition in such a way as to be transparent to subsequent code.

1.2.4.4 Types of Hardware Exceptions

Because of architectural differences, PRISM hardware has a different set of possible hardware exceptions than does VAX hardware. While there is some obvious overlap, no assumptions should be generally made about the nature of similar exceptions and handling.

Arguments passed to condition handlers for PRISM hardware exceptions are as defined in the PRISM SRM. These are in no way made to look any more like VAX exception hardware arguments than they already may appear.

1.2.5 New and Changed Functionality from VMS

1.2.5.1 Invocation Descriptor-based Condition Handlers

The PRISM Calling Standard allows invocation descriptor-based condition handlers. These are established at compile time and are pointed to from the invocation descriptor of the establishing procedure. Any type of procedure may have an associated invocation descriptor-based handler. This includes procedures that do not possess a stack frame when invoked.

There is no direct operating system support for dynamic call frame-based handlers. The basic operating system condition handling is only capable of finding invocation descriptor-based handlers. Dynamic handler functionality should be provided by using a basic descriptor-based handler which can locate dynamically established handlers.

Thus, there are no native library services corresponding to the LIB$ESTABLISH and LIB$REVERT procedures of VMS.

\It is the responsibility of a language compiler to detect whether a procedure establishes a dynamic condition handler. If so, the compiler must allocate a field in the procedure's descriptor that points to the invocation descriptor for a basic descriptor-based handler provided by the language's RTL to find dynamic handlers. In this situation, compilers must make sure that the procedure has a stack frame when active.\

1.2.5.2 Reinvocable Condition Handlers

Currently, the VMS Calling Standard does not allow an active condition handler to be reinvoked should another condition occur. Moreover, the presence of a handler invocation on the call stack modifies the search order for the second condition. This has caused some problems implementing certain languages, particularly PL/1.

In Mica, handlers are allowed to be reinvoked. This attribute may only be set at compile time for invocation descriptor-based handlers.

If a condition occurs while handling an existing condition and a reinvocable handler's invocation is encountered, then the search algorithm searches for all handlers established between that invocation and the invocation of its establishing procedure. As each is found, it is called if it possesses the reinvocable attribute. Once the establishing invocation is passed, any subsequently established handlers are always called.

Note that this attribute may not be changed dynamically at runtime, and if unspecified at compile time, the default is language dependent.

1.2.5.3 Unwind Services

The Mica Unwind system service differs in several ways from the VMS Unwind system service. Once an unwind request is made with valid arguments, control is never returned to the requesting procedure. Instead, the unwind takes place immediately.

An unwind target may be specified in two ways: a request may be made to either unwind to the caller of the currently active condition handler's establisher or to unwind to a target procedure identified by its associated stack frame. Procedure invocations without stack frames may not be unwound to by this second method.

Unwind does not require a condition handler to be already active. This allows the service to be used by languages to implement nonlocal GOTOs. In all unwinds, any established handlers between the requesting procedure invocation and the target are called.

The Mica Unwind service also allows multiply active unwind requests. These may occur in two ways: overlapping and nested.

Nested unwinds are unwinds that do not collide. For example, if an unwind is taking place to target procedure A and a condition is raised during the execution of a procedure called by an active handler H, if that procedure unwinds to some invocation between itself and H, then the unwind proceeds with no effect on the original unwind operation.

Overlapping unwind requests are more complex: should an unwind request be made during an unwind operation (for example, after a condition has been raised during an unwind), and the target of the new unwind is either an invocation which would have been unwound by the original unwind or is an ancestor invocation of the original unwind target, execution proceeds as if there is only a single unwind in progress, and the target is the "older" of the previous or currently requested target. For example, assume a thread is unwinding to procedure invocation A and a second unwind request is made to unwind to procedure invocation B. If A is an ancestor of B, the unwind continues to A. If B is an ancestor to A, then the unwind continues to B.

For a second unwind, the handler that is being unwound at the point of the first request is not called again. A second unwind essentially continues the first unwind to the same target or to the target of the second unwind. Unless the first unwind was an exit unwind, the unwind is continued with the second unwind's condition and mechanism arrays.

Unwind in Mica has been further extended to take an optional condition record as a parameter. If present, this parameter is passed to each handler called for the unwind and contains a status value reflecting the nature of the unwind, together with any necessary parameters. Furthermore, this status value is the status returned to the target of the unwind.

Two flags in the primary *condition vector* of a condition array used in an unwind indicate Exit or Unwinding status. These flags are set by the Unwind service, depending on the arguments with which it is called. Condition handlers may examine these flags to determine if an unwind is taking place.

In Mica systems, thread exit is accomplished by using the *exec$terminate* library service, which causes an unwind with an Exit status. In this mode, the unwind algorithm completely unwinds the call hierarchy of a thread, calling any established handlers it finds, and then executes the Exit system service to run exit handlers.

\Comments are invited on the new and different behavior of the Mica Unwind system service. It is intended to provide a simpler and easier-to-understand service, while extending it to allow support for contemporary languages, compilers, and run-time libraries.

The Mica Unwind service is incompatible both in implementation and goals with the VMS Unwind service. Comments are specifically requested concerning the impact these incompatibilities will have on transportability of applications.\

1.2.5.4 Vectored Handler Support

Like VMS, Mica supports vectored handlers. However, the VMS primary, secondary, and last chance vectored handlers have been extended in number and functionality. Mica supports two ordered lists of vectored handlers:

1. The primary vectored handlers: this list is FIFO ordered and searched before any invocation descriptor-based handlers are considered.
2. The last chance vectored handlers: this list is LIFO ordered and searched after all invocation descriptor-based handlers have reraised a condition.

In addition, these lists are securely kept in system space to prevent any accidental corruption.

These lists are finite in size and are kept in the thread control region (TCR) to prevent accidental corruption. Their maximum size is TBD.

1.2.5.4.1 Condition Stack Support

To support and improve Mica's debugger, a user-mode thread condition stack may be specified via a system service. This stack is used to call user-mode vectored handlers only.

If defined for a thread, it is used to call ALL user-mode vectored handlers. Thus, mechanism and condition records are first delivered to the primary handlers on the condition stack. If these handlers resignal, they are transferred to the thread's main stack and the invocation descriptor-based handlers are called. If these resignal, or the main stack is too corrupted, the condition and mechanism records are transferred back to the condition stack and the last chance handlers are called.

\This allows conditions to be delivered to primary vectored handlers without a thread's normal stack being modified. However, by the time conditions are delivered to last chance vectored handlers, the thread stack will have been used.\

1.2.5.4.2 Raising Conditions in Target Threads: Asynchronous Conditions

Mica implements system services that allow one thread to raise a condition in a group of threads (even if they belong to different processes). These services are described in the Internal System Services Manual.

Each service is built out of the capability to raise a condition in a single target thread: when the service is called for a specified target thread, Mica delivers a user-mode AST to the thread. This AST is handled by an AST handler located in system space but user-mode executable. This handler uses the *exec$finish_ast* system service to clear the AST In Progress bit for that thread, then raises a condition using the two AST parameters.

\If at any time in the future there is a need to simulate ULTRIX signals on top of Mica, this mechanism can be used together with a suitably written last chance handler that maintains "signal queues". This simulation is not implemented as a part of Mica.\

1.2.6 Design Description

1.2.6.1 Levels of Handling

There are four possible levels of condition handling:

1. Primary vectored condition handlers
2. Invocation descriptor-based handlers (abbreviated to descriptor-based handlers)
3. Last chance vectored condition handlers
4. The system catchall handler

Vectored handlers may be specified independently for kernel and user mode. Descriptor-based handlers may also be established independently for both modes. All handlers only execute in the mode from which they were established.

The search algorithm for condition handlers considers these possible types in the above order. The first handler found that has been established for the mode in which the condition occurred is used. At any one time, there may be many primary and/or last chance vectored handlers. At any one time, there may be several invocation descriptor-based condition handlers established, each associated with a descriptor for an active procedure.

1.2.6.1.1 Vectored Handlers

For each access mode, there may be an arbitrary number (up to the maximum) of primary and/or last chance handlers established. These may be established only at runtime and are kept in two separate ordered lists. Vectored handlers may only be changed for the mode in which the change request is made.

Primary vectored handlers are intended to be called in FIFO order. Therefore, new primary handlers are appended to the primary handler list. When the debugger is present in an image, it establishes the first primary vectored handler in the list. When this primary vectored handler exists, users are discouraged through documentation from deleting it.

Conversely, last chance handlers are intended to be called in LIFO order. Thus, new last chance handlers must be added to the beginning of the last chance handler list. When the debugger is present in an image, it tries to establish the last chance handler, which is called next to last. Users are discouraged through documentation from deleting this last chance vectored handler.

\RTLs and shared images may establish other last chance handlers at thread creation, such as the ULTRIX last chance handler, the traceback handler, and so on. This is detailed in either Chapter 5, Process Structure, or in the PRISM Calling Standard. Users are discouraged from deleting any of these handlers.\

1.2.6.1.2 The System Catchall Handler

Should all handlers reraise, then the condition is handled by the system-supplied catchall handler.

The catchall handler is a user-executable procedure, mapped in system space, that catches all improperly handled conditions. It simply dumps a portion of the thread's stack and scalar registers out in ASCII to the output device/file defined for the thread's error messages \in VMS, SYS$ERROR\.

1.2.6.1.3 Invocation Descriptor-based Handlers

From the Mica point of view, each procedure invocation may or may not have exactly one invocation descriptor-based handler associated with it. If it exists, this handler's invocation descriptor address is located in the invocation descriptor for its establishing procedure (see the PRISM Calling Standard). The operating system is not capable of directly finding any other handlers that a procedure may establish at runtime.

Thus, for a specific language environment, it is the responsibility of the language RTL to supply an invocation descriptor-based handler that can locate and pass control in the appropriate fashion to any language-specific handlers that may or may not be in existence for the given procedure.

Note that descriptor-based handlers are called for any type of condition. No context is stored in the establisher's descriptor indicating what types of conditions the handler may be capable of handling. It is up to the programmer to code these handlers with this in mind. Therefore, such handlers may choose to handle the condition, pass it on, or (if possible) dismiss it entirely and return to the original instruction stream.

1.2.6.2 Establishing Handlers

A thread establishes or changes a vectored handler using the *exec$establish_vectored_handler* system service. This is described in Section 1.2.9.1.

Dynamic handlers are established or changed either by language semantics, or by explicit procedure calls to LIB$ESTABLISH or LIB$REVERT. As previously stated, these procedures do not exist. Compilers special case such calls, and generate the appropriate code to establish a handler and find it via an associated descriptor-based handler.

This behavior is provided for ease of utility/application porting to Mica systems.

1.2.6.3 Locating a Handler for a Condition

The dispatcher first searches for primary vectored handlers. If this search fails, then the descriptor-based handler search is commenced by examining the invocation descriptor of the procedure in which the condition occurred. If no descriptor-based handler is established there, the dispatcher starts searching backward through the call hierarchy, looking for descriptor-based handlers. It calls all descriptor-based handlers it finds, until one returns a Continue status.

Before calling a descriptor-based handler, the dispatcher checks to see if the associated return address is its unique handler-calling site. If it is, the descriptor belongs to a handler that was active when the condition occurred.

In this case, the dispatcher locates the invocation descriptor for the procedure that established the active handler, and checks to see if the active handler may be reinvoked. If this is allowed, the dispatcher calls the handler again with the new condition's arguments. If reinvoking is not allowed, then the dispatcher continues its search for a handler to call for the new condition from the caller of the active handler's establishing procedure.

Otherwise, the dispatcher calls the handler and on return simply looks next at the caller's descriptor.

The dispatcher continues this search through the call hierarchy until either some handler returns success or it reaches the initial thread invocation. If no suitable descriptor-based handler is found, or none return a Continue status, the dispatcher attempts to call the first last chance handler and, should it reraise, the next last chance handler, and so on. If no last chance handlers are established or all reraise, then the dispatcher calls the system catchall handler. This checks an inhibit message flag in the thread control region, and if FALSE, generates an error message containing the thread's register context and the top of its stack. Thread exit is then requested by using the *exec$exit* system service.

\An exit unwind does not take place in this case because an unhandled condition is a serious problem. The PRISM Calling Standard specifies that last chance handlers be explictly set up at thread initialization to handle such conditions. Their absence, in this case, indicates a severe failure.\

Both descriptor-based and vectored handlers are called with condition and mechanism records as arguments. A condition record describes the condition. A mechanism record describes the environment in which the handler is called and the environment in which the condition occurred.

These records are both fully described in Section 1.2.7.2.1.

1.2.6.4 Returning from a Handler

A condition handler may finish its processing by either a simple return to the dispatcher, or by calling the Unwind system service before returning. If it requests an unwind, the Unwind service checks the validity of the request, and if valid, immediately starts to unwind calls. Note that the handler itself is called with the unwind.

In the first case, a condition handler returns either Continue or Reraise status. The dispatcher only looks at the least significant bit of the return condition value. If this bit is clear, the dispatcher interprets the status as a Reraise. Otherwise, it assumes that the handler is requesting a Continue.

If Continue is returned, the dispatcher first checks the condition record to determine if continuation is allowed by looking at the primary condition's *condition_flags* field. If it allows Continue, the dispatcher then restores from the mechanism record the register context of the procedure in which the condition was raised. The processor status (PS) and program counter (PC) are taken from the primary entry of the condition array and set up on top of the stack.

The dispatcher then completes its processing by executing an REI instruction. This causes a return to the procedure, at the target PC, in the same access mode in which the dispatcher was running: the mode in which the condition was originally raised.

If a Continue is not allowed for the particular condition, then the dispatcher raises a second condition, indicating an attempt was made to continue from a noncontinuable condition. This second condition is also noncontinuable.

A Reraise causes the dispatcher to search for the next available handler, using the same condition record and an updated mechanism record. The dispatcher continues searching for the next handler from the point at which it called the returning handler. If the dispatcher fails to locate a descriptor-based handler, and it encounters a procedure with no prior frame pointer (indicating the beginning of the thread's procedure call tree), then it calls the last chance handlers in LIFO order.

The last chance handlers are expected to handle any type of condition, and to either return Continue, start an unwind operation, or cause the thread to exit. If all attempt to reraise, the dispatcher calls the catchall handler, which eventually forces the thread to exit.

1.2.6.5 Handler Call Unwinding

A handler or any descendant procedure called directly or indirectly by a handler has the option of unwinding procedure invocations and returning to a specified PC in a target ancestral procedure. When this is done, any established descriptor-based handlers that are found between the most recently active handler and the target procedure invocation are called and notified that an unwind is in progress. This allows each procedure being terminated a chance to perform clean-up processing before all context is lost.

The default action for an unwind is to unwind to the caller of the procedure invocation that established the currently active handler. \Note that this default only applies if a handler is active: there is no default in the general unwind case.\

1.2.7 Implementation Description

1.2.7.1 Location of Primary and Last Chance Vectors

The primary and last chance vectors point at the primary and last chance vectored handlers and are used by the dispatcher to locate these handlers. To help ensure that these vectors may not be easily corrupted from user mode by a thread, the lists are kept in the thread control region. This area of the thread's address space has user-mode read access and kernel-mode read/write access.

A system service is provided to make or change vectors in these lists. Note that these lists may be directly read from either access mode.

1.2.7.2 Handling a Condition: The Unified Processing Strategy

Both hardware exceptions and software conditions are handled by the same procedures and data structures. This centralizes the mechanism for locating handlers, and allows the maximum amount of processing to take place in the access mode in which the condition occurred.

1.2.7.2.1 Data Structures

Two data structures are used to hold all information regarding a condition: the condition record and the mechanism record. The condition record contains all condition values and arguments associated with a condition. The mechanism record contains information regarding the environment at the time of the condition, together with the environment of a handler when it is called.

1.2.7.2.1.1 Condition Record

A condition record is defined as:

```
exec$condition_record : RECORD
    vector_number:integer[0..];
    primary_condition : POINTER exec$condition_vector;
    LAYOUT
        vector_number;
        primary_condition;
    END LAYOUT;
END RECORD;
```

Condition vectors each have the following definition:

```
TYPE
    exec$condition_flags : (
        exec$c_condition_unwinding,
        exec$c_condition_noncontinuable,
        exec$c_condition_exit_unwind,
        exec$c_condition_during_ast,
        exec$c_condition_async
        );

    exec$condition_vector(argument_number:integer[0..]) : RECORD
        CAPTURE argument_number;
        condition_name : exec$t_status_value;
        condition_flags : ARRAY [exec$condition_flags] OF bit;
        message_section : POINTER exec$message_section;
        condition_list : POINTER exec$condition_vector;
        arguments : ARRAY [1..argument_number] OF exec$argument_descriptor;
        processor_status : prism$processor_status;
        condition_address : POINTER longword;
        LAYOUT
            condition_name;
            condition_flags;
            argument_number;
            message_section;
            condition_list;
            sbz2 : FILLER(longword,1);
            arguments;
            processor_status;
            condition_address;
        END LAYOUT;
    END RECORD;
```

The *condition_name* field is a Mica status value. The *argument_number* field is the number of condition argument descriptors, not including the processor status (PS) and PC longwords.

The *condition_flags* field is a bit field used to indicate condition characteristics, as follows:

- The *exec$c_condition_noncontinuable* flag is used to indicate whether a handler may return a Continue request to the dispatcher. Thus, hardware or software conditions may be set continuable or noncontinuable when they are raised. At any time, a continuable condition may be set noncontinuable. However, once a condition has been set noncontinuable, it may not be set continuable.
- The *exec$c_condition_unwinding* and *exec$c_condition_exit_unwind* flags are used to indicated to a handler that it has been called by *e$unwind*, the Unwind executive service, and why it has been called.
- The *exec$c_condition_during_ast* flag is used to indicate that the condition was raised during AST processing. Handlers may then decide whether a condition should be reraised for handlers established prior to the AST being delivered.
- The *exec$c_condition_async* flag is set if the condition was raised by a different thread, using one of the system services Mica provides for asynchronous conditions.

The message pointer field, *message_section*, is either zero, or points at a message section or message section file specification through which the status value is to be translated to an ASCII message. \This is provided for local message support.\

The *processor_status* and *condition_address* are stored in the vector when the condition occurred.

All condition-specific arguments present in a condition vector are in descriptor format. This format embeds information as to the type and size of an argument, in addition to the argument, in a condition vector. Thus, condition records become more useful and accessible to the system message procedures and from higher-level languages. \Note that these entries are located on the thread's stack and are quadword aligned.\

\In Mica, condition records are also used to pass information on process or thread exit, and during unwind operations.\

Condition vector argument descriptors have the following layout:

```
exec$argument_descriptor: RECORD
    description: quadword;
    argument: quadword;
    END RECORD;
```

\The nature of these descriptors is currently under discussion with the PRISM Calling Standard Group. Two formats were originally specified: an immediate value descriptor and a calling standard general descriptor. Since the former does not currently exist in the standard, and the latter has changed since the design, this is under review.\

\Mica condition vectors and records are incompatible with those of VMS. This is due to their extended use in the system and their extended functionality.\

1.2.7.2.1.2 Mechanism Record

The Mica mechanism record is defined as follows:

```
TYPE
    exec$mechanism_record : RECORD
        depth : integer;
        stack_valid : boolean[..] SIZE(longword);
        condition_sp : POINTER anytype;
        condition_fp : prism$call_frame_pointer;
        condition_previous_fp : prism$call_frame_pointer;
        condition_ed : POINTER prism$entry_descriptor;
        condition_pc : POINTER prism$instruction;
        volatile_registers : ARRAY[4..31] OF longword;
        establisher_fp : prism$call_frame_pointer;
        establisher_previous_fp : prism$call_frame_pointer;
        establisher_ed : POINTER prism$entry_descriptor;
        establisher_pc : POINTER prism$instruction;
        establisher_registers: ARRAY[4...63] OF longword;
        LAYOUT
            depth;
            stack_valid;
            condition_sp;
            condition_fp;
            condition_previous_fp;
            condition_ed;
            condition_pc;
            sbz1 : FILLER(longword,1);
            volatile_registers;
            establisher_fp;
            establisher_previous_fp;
            establisher_ed;
            establisher_pc;
            establisher_registers;
        END LAYOUT;
    END RECORD;
```

Field names prefixed by *establisher_* refer to the invocation that established the handler being called. Field names prefixed by *condition_* refer to the invocation in which the condition being handled occurred.

The *stack_valid* Boolean is used to indicate if a stack corruption was detected during the handler dispatcher's search for handlers. A value of TRUE indicates a valid stack. The only handlers that may expect to be called after stack corruption are the last chance vectored handlers and the catchall handler. This allows such handlers to detect whether they were called as a result of all preceding handlers reraising, or as a result of stack corruption.

The register area contains the saved values of all registers defined in the PRISM Calling Standard to contain the invocation context for the procedure invocation where the condition occurred.

\The values in the scalar registers R8 and R9, at the time of the condition, are saved in the mechanism record register context area. These registers are defined by the PRISM Calling Standard as possibly returning a function value. Modification of these fields in the mechanism record changes the status returned by an unwind.\ → and on a continue

As in the PRISM Calling Standard, if a frame pointer modulo 8 is zero, the associated procedure has a stack frame; otherwise, the frame pointer, after clearing the low-order three bits, points at the procedure's invocation descriptor.

Unlike condition records, mechanism records are fixed length.

When a condition is delivered to the handler dispatcher, only the *volatile_registers* and *condition_* fields are filled in; all other fields contain zero. This is called a prototype mechanism record. Since the remaining fields depend on what handler is being called, they are filled in by the handler dispatcher just prior to calling handlers.

Note that, with the exception of the continuation flag, handlers may modify any of the fields of a condition or mechanism record.

\This record is incompatible with the VMS mechanism array. This is due to the increased complexity of dealing with procedures that do not possess a stack frame, the concept of an invocation descriptor-based handler, and the need for increased context information by handlers.\

1.2.7.2.2 The Handler Dispatcher Service: e$handler_dispatcher

The *e$handler_dispatcher* executive service is the only service that is called when any type of condition is raised. This service is called either indirectly by a thread to raise a software condition, or directly by the system to notify a thread of a hardware exception. It is called as a procedure with the following arguments:

```
PROCEDURE e$handler_dispatcher(
          IN OUT condition_record: exec$condition_record;
          IN OUT mechanism_record: exec$mechanism_record;
          );
```

This service can be called in either user mode or kernel mode; it is located in system space, executable from both modes. However, the procedure name is not directly exported to user mode, since the procedure may change location between releases. Instead, a procedure variable is reserved at a fixed location in each process control region (PCR).

The condition record in this call is complete, but the machanism record *establisher_* fields are not yet valid. These fields are filled prior to calling handlers. They change for each handler the dispatcher calls.

1.2.7.2.3 Raising Software Conditions

To raise a software condition, a procedure does not directly call *e$handler_dispatcher*. Instead, it calls *exec$raise_condition* with a condition record as the only argument. This condition record's condition vectors all have zero for their PS/PC entries.

The *exec$raise_condition* library service collects its caller's register context to build a prototype mechanism record for *e$handler_dispatcher*, then adds its return address and the caller's PS to each condition vector. Once both records are finished, *exec$raise_condition* transfers control to *e$handler_dispatcher* via a jump, using the procedure variable defined in the PCR for *e$handler_dispatcher*. This avoids an unnecessary full procedure call. \Note that this is not a simple jump; it is a jump past the code in *e$handler_dispatcher* that sets up its frame.\

At the entry to *e$handler_dispatcher*, software and hardware conditions are indistinguishable. In both cases, *e$handler_dispatcher* completes the mechanism record each time a handler is called.

The *exec$raise_condition* library service is called with a condition record:

```
PROCEDURE exec$raise_condition(
          IN OUT condition_record: exec$condition_record;
          )RETURNS status;
```

As many condition vectors may be passed in the condition record to this library service as are required to define the software condition. Note that this service only returns to its caller if a handler returns Continue to the dispatcher, or if an error is detected in its argument.

1.2.7.2.4 Raising Hardware Conditions

Hardware conditions result from a processor generating an exception, causing its execution of the current thread to be interrupted, and to continue in kernel mode at an address in the executive. Each type of hardware exception has an associated address, specified in a processor's system control block (SCB).

When a hardware exception occurs, if it is not a user-visible exception, the executive deals with the exception directly and subsequently returns to the previous mode. Otherwise, if the previous mode was user mode, the executive checks to see if the thread has defined a condition stack. If one exists and it is not the current stack, the thread's previous-mode stack is switched to the condition stack, and the thread control region (TCR) is changed to reflect this state.

The executive then allocates a condition record on the previous-mode stack. The hardware exception arguments are moved to the condition record and, except for the PS/PC pair, the arguments are removed from the current-mode stack.

A mechanism record is allocated on the previous-mode stack and the call-context registers of the procedure invocation in which the hardware exception occurred are saved in this record.

The invocation descriptor for *e$handler_dispatcher* is located, and a stack frame is allocated and set up for the simulated call to *e$handler_dispatcher*.

Scalar registers are then set up as defined in the calling standard for the entry to *e$handler_dispatcher*. Finally, the return PC on the delivery-mode stack is changed to the entry address of *e$handler_dispatcher*, and an REI is executed. This has the effect of "calling" the dispatcher in the previous mode.

\Invisible hardware exceptions within a thread do not result in a call to *e$handler_dispatcher*. System routines directly handle such exceptions.\

1.2.7.2.4.1 Stack Overflow

Should a thread's user-mode stack overflow, causing a hardware exception, the executive grants read/write access to the guard page and raises a stack overflow condition in user mode.

The executive may detect a potential user-mode stack overflow while attempting to allocate a condition/mechanism record and/or *e$handler_dispatcher* stack frame on a user-mode stack in response to a hardware exception. In this case, it first grants read/write access to that stack's guard page, completes the allocations and data transfers necessary for the hardware exception, then allocates a second set of condition/mechanism records and call frame on that user-mode stack. This second set is filled in to indicate a user-mode stack overflow condition, which is then raised when the REI is executed.

This stack overflow condition is a continuable condition: an RTL condition handler will try to extend the stack when it receives control. Note that if this condition is delivered because the condition stack overflowed, then a primary vectored handler is responsible for dealing with the condition, since the RTL handler may be invocation descriptor established.

Should the executive attempt to allocate these data structures on a user-mode stack and discover this stack has overflowed and there is no longer a guard page to raise a stack overflow condition, then the particular user-mode stack is reset to its starting value for the thread, and only a stack overflow condition is raised on the stack.

Note that in this case of user-mode stack overflow, the condition is not continuable.

\These algorithms apply to either a thread's normal user-mode stack or a condition stack if one is available.\

Should a hardware exception occur and the kernel stack overflow during subsequent processing and synthesizing of the call to *e$handler_dispatcher*, a system bugcheck occurs.

12.7.2.4.2 Kernel Mode Processing

The above description of kernel mode processing is illustrated by the following algorithm:

```
thread causes hardware exception
   hardware vectors the exception through SCB
   kernel receives control at delivery of exception in kernel mode
   if exception EQ thread invisible
      directly handle it
    else
      find invocation descriptor for e$handler_dispatcher
      set stack-flag = valid
      if previous mode = user mode and thread established condition
                                                          stack
         if not already running on condition stack
            if exception = stack overflow
               if guard page available
                  unprotect guard page
                else
                  reset thread main stack to initial value
               end
            end
            save previous mode stack pointer in TCR
            switch previous mode stack to condition stack
            indicate this in TCR
          else
            if exception is stack overflow
               if guard page available
                  unprotect guard page
                else
                  reset condition stack to its starting value
                  set stack-flag = invalid
               end
            end
         end
      end

      if not enough space on previous mode stack for subsequent
                                                       allocates
         if previous mode = kernel
            system bugcheck
          else
            if guard page available
               unprotect user guard page
               if not sufficient space for current exception and overflow
                                                          condition
                  reset stack to initial starting point
                  set stack-flag = invalid
               end
             else
               reset stack to initial starting point
               set stack-flag = invalid
            end
            allocate condition record on user stack
            move exception name and argument count into record
            copy typed exception arguments from kernel stack to condition
                                                            record
            allocate mechanism record on previous mode stack
            if stack-flag = invalid
               set mechanism record to show invalid stack
             else
               set mechanism record to show valid stack
            end
            save registers in the mechanism record
            enter exception site information in mechanism vector
```

```
            allocate call frame for call on previous mode stack
            set the call frame's previous frame pointer to point
               back to the previous frame on the previous mode stack
            set the call frame's return address to the PC value for
                                                    the exception
            allocate condition record on user stack
            copy typed stack failure arguments to condition record
            allocate mechanism record on previous mode stack
            save registers in the mechanism record
            enter exception site information in mechanism record
            if stack-flag = invalid
               set mechanism record to show invalid stack
               set primary condition noncontinuable in condition
                                                          record
             else
               set mechanism record to show valid stack
            end
            allocate call frame for call on previous mode stack
            set the call frame's previous frame pointer to point
               back to the previous frame on the previous mode stack
            set the call frame's return address to the PC value for
                              the entry point of e$handler_dispatcher
          end
        else
          if previous mode = kernel mode
             expand exception arguments into typed exception arguments
             push exception name and argument count onto stack and
                              set up condition record data structure
           else
             allocate condition record on user stack
             move exception name into condition record
             move argument count into condition record
             move typed exceptions arguments from kernel stack into
                                                    condition record
          end
          allocate mechanism record on previous mode stack
          save registers in the mechanism record
          enter exception site information in mechanism record
          set mechanism record to show valid stack
          allocate stack frame for call on previous mode stack
          set the call frame's previous frame pointer to point
                back to the previous frame on the previous mode stack
          set the call frame's return address to the PC value for
                                                  the exception
        end
        set register context for previous mode e$handler_dispatcher call
        if exception occurred in kernel mode
           jump to e$handler_dispatcher
         else
           clean the current stack back to PS/PC
           change the return PC to the entry point of e$handler_dispatcher
           REI
        end
    end
```

\Note that this algorithm assumes that a condition stack is always at least as large as would be required to deliver the "largest" hardware exception together with a stack overflow condition. This is legislated by the system service used to create a condition stack.\

1.2.7.2.4.3 Format of the Condition Record for Hardware Exceptions

As described above, the hardware arguments are transferred to the previous-mode stack and formatted into a condition record containing a single condition vector for the call to *e$handler_dispatcher*. During this copying process, the arguments are converted to a descriptor format. For all but one hardware exception, each descriptor provides exactly one of the hardware arguments specified in the PRISM SRM.

In this vector, the *processor_status* field contains the contents of the processor status register when the hardware exception took place. The *condition_address* field contains either the address of the instruction where the exception took place, or the address of the next instruction in virtual memory. This depends on what the hardware exception is, as shown in the PRISM SRM.

The FLBC (as discussed earlier in Section 1.2.3.2.2) is the only hardware exception for which the arguments that handlers receive in the associated condition record are not exactly as stated in the PRISM SRM. For this fault, instead of a zero longword and the instruction which caused the fault being the two arguments, the interrupt code for this fault converts the zero longword into the unpacked displacement from the instruction. Thus, the kernel mode processing described above receives two fault-specific arguments: the unpacked displacement argument and the instruction which caused the fault, which it then packages into descriptor format in the condition vector.

\This is done to localize the software that unpacks the fields of this instruction and thus remove a dependency on the instruction format from all levels of software in the system.\

1.2.7.3 Locating a Handler from the Dispatcher

\In the following algorithm, FPSAV and FLAGS are as defined for frame descriptors in the PRISM Calling Standard.\

Once entered, the dispatcher attempts to find a condition handler as described.

The dispatcher uses the following algorithm:

```
enter_dispatcher:
if primary condition vector indicates continuation not possible
   set stop = TRUE
 else
   set stop = FALSE
end
point at 1st primary vector in the FIFO list in the TCR
while primary list not empty
   get entry in list
   extract stored depth parameter ! defined in system services section
   set depth to be the extracted value
   set handler pointer for this handler
   call handler_call
   position to next entry in list
end

if we are running on the condition stack
   call exec$switch_stack
end
```

```
set current_descriptor to address of e$handler_dispatcher
set depth = -1
set stack-flag to reflect contents of mechanism record stack information
repeat
  if stack-flag = valid
     perform sanity check on stack
     if check fails
        set stack-flag = invalid stack
      else
        set caller of current_descriptor (using FPSAV register)
                                          as new current_descriptor
        depth = depth + 1
     end
  end
  if current_descriptor address EQ 0 or stack-flag =  invalid
     if the thread established a condition stack
        call exec$switch_stack
     end
     if stack was invalid
        set mechanism vector to indicate invalid thread stack
     end
     point at 1st last chance vector in the LIFO list in the TCR
     while last chance list not empty
        get entry in list
        extract stored depth parameter
        set depth to be the extracted value
        set handler pointer for this handler
        call handler_call
        position to next entry
     end
     call catchall handler
     if stack-flag = invalid
        cause thread exit using exec$exit  ! note lack of unwind here
      else
        call exec$switch_stack
*  ------>  call exec$terminate(condition_record)
     end
  end

  if return address of invocation of current_descriptor
                                     is within dispatcher
     find mechanism record for this handler
     get establisher's descriptor from record
     if FLAGS = 3 for establisher's descriptor
        repeat
           if current_descriptor establishes a descriptor-based handler
                        AND FLAGS = 3 for current_descriptor
              get handler's invocation descriptor
              call handler_call
           end
           if current_descriptor EQ establisher's
              break out of repeat loop
           end
           set caller of current_descriptor
             (using FPSAV register) as new current_descriptor
        end  ! repeat
     end
   else
     if current descriptor establishes a descriptor-based handler
        get its address
        call handler_call
     end
  end
end      ! repeat
```

```
handler_call:
  if stop = TRUE
     set exec$c_condition_noncontinuable bit in condition_flags in
                                          the primary condition vector
  end
  save flags from primary condition vector
  call handler from this single known location
  if condition_flags[exec$c_condition_noncontinuable]
     set stop = TRUE
  end
  restore flags to primary condition vector
  if lsb of return status EQ 0       ! reraise case
     return
   else
     if stop = FALSE
        restore registers from mechanism record
        set up e$handler_dispatcher frame to have the previous
                              frame as its caller
        return     ! note this will clean the stack
      else
        raise error condition continue not allowed
     end
  end
```

\The dispatcher directly accesses data structures defined in the calling standard. It also directly manipulates its own stack and argument records.\

1.2.7.4 Calling a Handler from the Dispatcher

Handlers are always called from the same site in the dispatcher. This allows subsequent identification of a handler while searching backward through a call hierarchy.

When a condition handler of any type is called by the dispatcher, it is passed two arguments: the original condition record and a mechanism record. Both of these structures are usually quadword aligned and stack based.

By the time the dispatcher has been invoked, the condition vector is complete, and the mechanism vector lacks only the information relating to the establisher of the handler. Once a handler has been located, the dispatcher fills in the *establisher_* fields and the *depth* argument.

Thus, a condition handler is a procedure of type:

```
TYPE
    exec$t_condition_handler : PROCEDURE(
            IN OUT condition_record : exec$condition_record;
            IN OUT mechanism_record : exec$mechanism_record;
            )RETURNS status;
```

For vectored handlers, the *establisher_* fields in the mechanism record contain zero. The *depth* parameter for vectored handlers is recalled from the control block filled in when the handler was established. It has a default value of −1.

For descriptor-based handlers, the *depth* argument is defined to be the number of procedure invocations between the invocation that had the condition raised and the invocation that established the handler. This excludes any invocations that were not searched due to multiply active conditions. For vectored handlers, this argument is set to whatever *depth* parameter was specified when the vectored handler was established. (See Section 1.2.9.1 for a description of the *exec$establish_vectored_handler* system service.)

1.2.7.5 Unwinding

A procedure uses the *e$unwind* executive service to unwind calls. There are three possible unwind requests:

1. Unwind to a specified procedure invocation.
2. Unwind to the establisher of a currently active condition handler.
3. Perform an exit unwind: unwind every procedure invocation until the beginning of the calling hierarchy is reached, at which point an Exit system service is executed. (Note that in this case, any overlapping unwinds are ignored.)

The *e$unwind* service is mapped in system space and is both user-mode and kernel-mode executable. The procedure definition is not exported to user mode. As in the case of the handler dispatcher, a procedure variable is reserved in each thread's PCR for *e$unwind*.

For user-mode calls, the *exec$unwind* system service is provided as part of the shareable image FMSHR to provide access that is invariant in virtual address across Mica releases. This procedure has the same definition and arguments as *e$unwind*. When called, *exec$unwind* finds the thread's PCR procedure variable for *e$unwind*, sets up its call frame to look like an *e$unwind* call frame, and jumps to *e$unwind*. \As is the case for the handler dispatcher, this is not a simple jump.\

The *e$unwind* service may be called directly from within kernel mode, since all system components are relinked for each release.

The *e$unwind* executive service has the following definition:

```
PROCEDURE e$unwind(
          IN frame: prism$call_frame_pointer = NIL;
          IN caller_of_establisher: boolean = false;
          IN exit_unwind: boolean = false;
          IN target_pc: POINTER anytype CONFORM = NIL;
          IN condition_record: exec$condition_record OPTIONAL;
          IN collapse_stack: boolean = true;
          )RETURNS status;
```

The *frame*, *caller_of_establisher*, and *exit_unwind* arguments allow three mutually exclusive methods of expressing the target procedure invocation for the unwind. If more than one is used, an error is returned. This service may be used at any time if either the *frame* or *exit_unwind* arguments are specified. However, using the *caller_of_establisher* argument requires that a condition handler already be active. In this case, if no handler is active, an error condition is raised.

The *frame* argument specifies the stack call frame pointer of the target procedure invocation. Thus, it may only be used to unwind to a procedure invocation that possesses a stack frame. Since this form of the service call may be used at any time, it provides a way of implementing nonlocal GOTOs.

The *caller_of_establisher* argument specifies that the target of the unwind is the caller of the establisher of the currently active condition handler.

The *exit_unwind* argument specifies that the unwind request is for an exit unwind and all procedure invocations will be unwound. The *exec$c_condition_exit_unwind* flag will also be set in the *condition_flags* field of the primary condition vector.

If an unwind request is made specifying neither of the three possible unwind types, an error is returned.

The optional *target_pc* argument specifies the return address within the target invocation at which to continue execution. It may be used with either the *caller_of_establisher* or *frame* arguments. If omitted, the original return address to the target procedure invocation is used.

The *e$unwind* service returns an error if the *target_pc* is specified in the *exit_unwind* case.

The *condition_record* argument may be used to specify an optional condition record. If specified, this parameter is used as the condition record to each handler called for the unwind. If no condition record is specified, then *e$unwind* allocates a condition record whose first condition vector has the status value *exec$_unwinding*.

The caller of *e$unwind* may use the *collapse_stack* boolean argument to cause *e$unwind* to either collapse the stack back to the point at which the target procedure last left it, or leave the stack unchanged (other than removing the *e$unwind* frame).

The *e$unwind* service sets the *exec$c_condition_unwinding* bit in the *condition_flags* field of the condition record's primary condition vector prior to calling all handlers.

1.2.7.5.1 Implementation Details

The *e$unwind* service executes in the access mode of its caller. It needs direct access to the environment of its caller and makes procedure calls using the calling mode's stack. There are two fundamental uses of this service: implementing nonlocal GOTOs and an exit unwind (essentially a nonlocal GOTO with its target before the first procedure invoked in the thread).

When called, if all arguments are valid, *e$unwind* never returns to the calling procedure. Instead, the unwind takes place and, for the non-exit-unwind case, the return from this service occurs in the target procedure invocation at the target or original return PC.

Should *e$unwind* detect an error condition, then two mechanisms are used to notify the thread:

1. If the error is an inconsistency in the arguments (for example, requesting both an exit unwind and unwind to specific target), an error status value is returned to the caller of *e$unwind*.
2. All other possible errors can only be detected during the actual unwind and this notification is given by raising an error condition. These errors include: not finding a specified target, no condition handler being active, stack corruption, unable to find target, and so on.

If *e$unwind* is successful in performing a non-exit unwind, the status returned to the target procedure invocation is:

1. If an optional condition record was specified, the condition value from the primary condition vector in the record.
2. If there is no condition record specified, but the unwind has resulted after one or more conditions were raised, the status returned comes from the saved R8 and R9 values in the most recent condition's mechanism record.
3. If neither 1 or 2 are present, then a success status is returned.

\Note that by using the optional condition record with *e$unwind* it is no longer necessary to jam a return status into a mechanism record.\

In the exit unwind case, when the beginning of the calling hierarchy is reached, *e$unwind* executes an *exec$exit* system service with the optional condition record as an argument.

An unwind is performed by executing a scan backward through the procedure invocation hierarchy. The scan is started with the caller of *e$unwind*. As each procedure invocation is processed, the following actions occur:

1. If the frame is that of the first condition handler encountered during the scan, and if this unwind request is to the caller of the establisher of the currently active handler, this handler's establisher's caller is set as the target of the unwind.

   \Note that if this condition handler establishes a condition handler, that condition handler is called.\

2. If this unwind request has a specific target, the current invocation is compared to that target and if it matches, the unwind is finished with status returned in the above manner.

\Note that if the target establishes a condition handler, that handler is NOT called.\

3. For all other invocations, any established handlers are called.

   \Note that this is the only possible action in the case of an exit unwind, until the beginning of the hierarchy is reached.\

During this scan, *e$unwind* determines:

- If this unwind has collided with a previous unwind

  \This can be detected since a frame pointing at the invocation descriptor of *e$unwind* is found on the stack.\

- If an attempt is being made to unwind out of an exit handler
- If the target procedure frame descriptor does not exist
- If the stack is corrupt

The last three possibilities are error conditions: in all cases the appropriate condition is raised.

If the unwind collided with an earlier unwind, then the following rules are applied:

1. The arguments of the previous unwind are found in the previous *e$unwind* stack frame. If a condition record was specified for that call, the primary condition value is checked to determine if the previous unwind was an exit unwind. If so, *e$unwind* raises an error condition.

   \Colliding unwinds are not allowed once an exit unwind has started.\

2. If the current target is the same as the previous target, an error condition is raised.
3. If the current target is an ancestor of the previous target, the current target remains unchanged.
4. If the current target would have been unwound by the previous unwind, then the previous target replaces the current target.

Note that in the last two cases, the following takes place:

- The current unwind effectively replaces the previous unwind: the previous unwind's condition record is effectively superseded, possibly along with its target.
- All invocations are unwound between the invocation requesting the current unwind and the previous *e$unwind* frame. When that frame is reached, the current unwind proceeds from the next invocation which the previous unwind would have next unwound.

  \Note that this is very subtle but extremely important to prevent incorrect unwind invocations and to prevent a possible infinite loop of unwind requests.\

Prior to starting the scan, *e$unwind* takes the following actions, since it will be calling condition handlers:

- If no condition record was specified in the call to *e$unwind*, a condition record is allocated and filled in to indicate one condition: *exec$_unwinding*.
- A mechanism record is allocated and partially filled in to reflect the site of the unwind request. The mechanism record associated with the first condition handler encountered (if any) replaces this allocated record.

As each invocation is processed, the following actions are taken during the scan, in the following order:

1. The *e$unwind* service modifies its own frame pointer to indicate that this invocation was its caller.

\This maintains the stack in a consistent state: stack space associated with unwound procedure invocations is now associated with the invocation under consideration. This effectively hides invocations that have been unwound, and thus prevents calling any handlers that have already processed the unwind, should a condition be raised during the unwind.\

2. If the invocation is an invocation of *e$unwind*, then the saved context of that invocation is examined to find the invocation it would have next unwound. That invocation is found and used in place of the *e$unwind* invocation.

   \This effectively ignores the *e$unwind* invocation and skips back to where it would have continued from, implementing the semantics discussed above for colliding unwinds.\

3. If the invocation establishes a descriptor-based handler, the mechanism record is completed with the establisher's environment information, and the handler is called with the condition and mechanism records.

4. The previous frame pointer is retrieved. If it is nonzero and is not the target of the unwind, then steps 1, 2, and 3 are repeated. Otherwise, *e$unwind* carries out the following actions:

   a. For non-exit unwinds, when this frame is the target frame:

   The *e$unwind* service modifies its environment to indicate that this latest frame is its caller. It then returns to either the original return address for this frame or the optional return address, if one was specified. This returns control to the target invocation and, depending on the state of the *collapse_stack* argument in the call to *e$unwind*, collapses the stack back to its last position for the target procedure.

   For non-exit unwinds, if this pointer is zero, this means the target was not found; therefore, unwind raises an error condition.

   b. For exit unwinds:

   The *e$unwind* service resets the thread stack to its initial value by moving its own frame to the beginning of the stack and calls *exec$exit* with the condition record.

Note that any *e$handler_dispatcher/exec$raise_condition* frames encountered are skipped over during the scans. They are not checked against a possible *frame* argument.

If, during the scan, *e$unwind* discovers that the thread's stack is now corrupt, it raises a stack corrupt condition.

1.2.7.5.2 Unwinding Through ASTs

If an AST is present on the stack during an unwind, it is unwound through by the mechanism described below.

When the AST was delivered, a frame was placed on the stack encapsulating the saved registers. This frame established a descriptor-based condition handler whose sole purpose is to dismiss the AST in the event of an unwind.

This handler uses the *exec$finish_ast* system service, which simply clears the thread's AST In Progress bit. It then performs any required frame cleanup and returns to *e$unwind*.

This handler and the special invocation descriptor are supplied in system space, user-mode executable.

\Note that even though this handler is located in system space, it does not need to be found via the PCR, as do the handler dispatcher and the unwind procedure. It is never directly called by users and, therefore, does not need to be invariant across system releases.\

1.2.7.5.3 The Unwind Algorithm

The actions of *e$unwind* may be represented as follows:

```
enter e$unwind

!  Check arguments:

if (more than one of frame, caller_of_establisher or exit_unwind
       arguments specified) OR (no frame AND no caller_of_establisher
                                  AND no exit_unwind argument specified)
   return argument error condition
end
end
if condition_record argument specified
   check argument for existence and consistency
   if error
      return argument error condition
   end
 else
   allocate condition record
   fill in exec$_unwinding status value
   set condition_record to this record
end

set frame_pointer = previous frame pointer

if frame specified
   set target_frame = frame
 else
   set target_frame = 0
end

set handler_found = false
frame_pointer = caller's frame pointer
allocate mechanism record on stack
copy to condition's context
save R8 and R9 in record

while frame_pointer NE target_frame
   call get_invocation_context
   if this is an invocation of e$unwind
      find this invocation's condition_record argument
      if previous unwind was an exit unwind
         raise already exiting error condition
      end
      find this unwind invocation's target
      if NOT(exit_unwind)
         if (ancestor of current target) OR target_frame EQ 0
             set target_frame = previous target
          else
             if previous target EQ current target
                raise error condition
             end
         end
      end
      set current invocation to next invocation which would have
                     been unwound by the previous e$unwind
       call get_invocation_context
   end
   if return_address in e$user_exit AND target_frame EQ 0
      raise condition attempt to unwind through exit handler
   end
   if return_address EQ dispatcher handler call site
           AND NOT(handler_found) AND caller_of_establisher
      set handler_found = true
      if target_frame EQ 0
```

```
            set target_frame = caller of the establisher of this
                                                              handler
            replace allocated mechanism array with this handler's
                                                    mechanism record
         else
            if target_frame EQ caller of the establisher of this
                                                              handler
               raise error condition
            end
        end
    end
    if caller_of_establisher AND return_address EQ dispatcher
       find establisher's frame pointer from mechanism record
                                       for this handler invocation
       frame_pointer = establisher's frame pointer
       call get_invocation_context
    end
    if descriptor establishes a descriptor-based handler
       fill in establisher's context into mechanism record
       restore R8 and R9 from mechanism record
       set exec$c_condition_unwinding bit in condition_flags of
                                         primary condition vector
       if exit_unwind
          set exec$c_condition_exit_unwind in condition_flags of
                                           primary condition vector
       end
       call handler with these arguments
       clean stack
    end
    if return address NE dispatcher
       frame_pointer = previous frame pointer
     else
       frame_pointer = frame pointer of dispatcher's caller
    end
    if frame_pointer EQ 0       ! exit unwind case
       if exit_unwind
          move e$unwind frame to beginning of stack, with
                                                  previous FP = 0
          exec$exit(condition_record)
        else
          raise error condition target not found
    end
end       ! while

if handler_found
   set return_status to R8 and R9 from mechanism record
                       of the condition unwinding
 else
   if optional condition array specified
      set return_status to primary condition vector status
    else
      set return_status to success
   end
end

!  Note that the next step involves a lot of work with the register
!  context and stack context of e$unwind :
!
```

```
set up context to indicate the target frame was the caller
                                                of e$unwind
set return PC to target_PC
if collapse_stack
   set up for collapse stack on return
 else
   set up for intact stack, minus e$unwind call frame on return
end

return with return_status


!  This procedure returns the context of a procedure invocation as
!  needed by the unwind algorithm. Note that it also checks for the
!  consistency of the invocation's environment and raises a condition
!  if the environment has been corrupted.

get_invocation_context:
   check this invocation's environment for inconsistency
   if inconsistent
      raise stack invalid condition
   end
   if frame_pointer mod 8 EQ 0
      descriptor = (frame_pointer)
      previous frame = 8(frame_pointer)
      return_address = 4(frame_pointer)
    else
      descriptor = frame_pointer cleared by 7
      previous frame = contents of saved FPSAV register
      return_address = contents of saved PCSAV register
   end
   set e$unwind previous FP = frame_pointer
   return success
```

For the above algorithm, the following points are true:

1. This service directly manipulates its calling environment. It also examines storage of any previous versions of itself it finds active.
2. The implementation of this procedure is highly dependent on the PRISM Calling Standard definitions and conventions.
3. Since the *e$unwind* invocation frame is always present during the entire unwind operation and uses standard procedure calling and return sequences, ASTs may safely occur during an unwind.

1.2.7.6 Actions at Thread Creation

When a thread is created, the RTL initialization associates a creator-defined last chance handler with a thread. This occurs immediately after thread startup.

Note that a thread always has the system catchall handler associated with it. This is not a vectored handler; it is a system-space procedure executable from user or kernel mode and called by the handler dispatcher if all handlers reraise.

Should this handler be called either as a result of the absence of other handlers, or as a result of all other handlers reraising, it causes the thread to exit. Note that it may attempt to print out a diagnostic message, depending on the condition value CNTRL field setting or the setting of a message override bit in the thread control region (TCR). (See Chapter 3, Status Codes and Messages, for a description of the condition value CNTRL field).

1.2.8 The PRISM Calling Standard

This design and the PRISM Calling Standard depend on each other. This chapter is currently in agreement with Version 0.5 of the PRISM Calling Standard.

1.2.9 Miscellaneous Mica System Services

1.2.9.1 exec$establish_vectored_handler

The *exec$establish_vectored_handler* system service allows a thread to specify or change any vectored handler in either of the two vectored handler lists. Changes are only allowed to the lists kept for the mode in which the service is called. The service is called from kernel mode using the *e$* entry point.

This procedure causes a vector cell in the thread control region (TCR) to be filled in, modified, or cleared. The cells are linked in a list, with the listheads located in the TCR. The service is very general, allowing either the primary or last chance lists to be modified. Vectored handlers may be replaced in a list, removed from a list, or added as a new entry to a list, either at the beginning or end of the list.

If a handler is being replaced, then its associated vector cell in the list is reused with the *handler* procedure variable and *depth* fields changed. If a handler is being removed, then the list's links are reset around the vector cell, and the cell is cleared.

```
PROCEDURE exec$establish_vectored_handler(
          IN vector: exec$t_vector_id;
          IN handler: exec$t_condition_handler;
          IN OUT vector_cell: POINTER exec$t_vectored_handler_cell OPTIONAL;
          IN depth: integer = -1;
          IN remove: boolean = false;
          IN beginning: boolean = false;
          )RETURNS status;
```

The *vector* argument is used to indicate which of the two lists is being modified. The *handler* procedure variable and *depth* arguments are used for either filling in a new vectored handler cell or changing an existing one. The optional *vector_cell* argument may be used to indicate the cell to reuse or to identify the cell to be removed when the optional Boolean flag indicates remove. Note that the default for *depth* is: -1.

The *beginning* argument is used to indicate whether a new vector cell is to be added to the beginning or to the end of the vectored handler list. The default is to add it to the end of the list *(beginning = FALSE)*.

All status returns are documented in the Internal System Services Manual.

1.2.9.2 exec$create_condition_stack

A thread may establish a stack for the delivery of conditions to vectored condition handlers with the *exec$create_condition_stack* system service. The service has one optional argument:

```
PROCEDURE exec$create_condition_stack(
          IN stack_size: integer = 0;
          )RETURNS status;
```

The *stack_size* argument is used to specify how large this stack should be. The default is a stack large enough to deliver the largest hardware condition and a stack overflow condition. Specifying a size smaller than this default causes the default to be used.

If the stack cannot be allocated, an error is returned.

This service modifies the TCR to indicate the presence, location, and last saved stack pointer (SP) for this stack.

An error is also returned if a condition stack has already been created for the issuing thread.

All status returns are documented in the Internal System Services Manual.

1.2.9.3 exec$switch_stack

\This service is not a general service. It is only used by the handler dispatcher from user mode and thus returns an error if called from any other location.\

The *exec$switch_stack* system service switches between the two possible user-mode thread stacks. This service may only be used from user mode. Its action is to first move the area on the current stack between the base of the current call frame and the stack pointer prior to the service call onto the target stack. It then adjusts the TCR data structures to reflect the target stack as the current stack and stores the new base address of the old stack.

When this service returns, the SP is set up for the target stack and the frame pointer (FP) is set to the correct address on the target stack, computed from the old FP relative to the information pushed onto the new stack.

```
PROCEDURE exec$switch_stack(
          )RETURNS status;
```

\Note that after the service returns successfully:

```
FP = ( target top of stack ) + ( old FP - top )
```

Since the PRISM Calling Standard specifies that FP is always in R2, we may reliably do this manipulation, since the frame in question always belongs to the handler dispatcher.\

Note that this service returns an error if a thread has not established a condition stack.

All status returns are documented in the Internal System Services Manual.

1.2.9.4 exec$change_unwind_vector

\This service is not intended as a general service. It is used by the debugger to get notification of unwind requests before they start. The situations in which the debugger needs to do this are well defined. This approach has been taken so that there is no interaction with the debugger for the majority of unwind requests, so that performance is maintained.\

The *exec$change_unwind_vector* service allows the unwind procedure vector in the process control region (PCR) to be changed. This allows another procedure to gain control when a request is made to *exec$unwind*, but leaves direct calls in kernel mode to *e$unwind* unaffected. The service returns the current contents of the unwind vector (for later replacement, if desired):

```
PROCEDURE exec$change_unwind_vector(
          IN new_procedure: PROCEDURE;
          OUT old_procedure: PROCEDURE;
          )RETURNS status;
```

The contents of the unwind vector in the PCR at the time of the call to this service are returned in the *old_procedure* output argument. The procedure *new_procedure* is used to replace the contents of the PCR unwind vector. Note that this procedure must expect the same arguments as *e$unwind*.

All status returns are documented in the Internal System Services Manual.

1.2.10 Miscellaneous Mica Executive Services

1.2.10.1 e$get_next_condition_handler

This service executes in the access mode of the caller. Like the *e$unwind* and *e$handler_dispatcher* executive services, it needs to examine the current stack and environment. It too is associated with a procedure variable in each thread's process control region (PCR) and a procedure provided in the system shareable image, FMSHR.

The function of the *e$get_next_condition_handler* executive service is to return the address of the invocation descriptor of the next condition handler that the handler dispatcher will call, should the currently active handler resignal. It has two output arguments:

```
PROCEDURE e$get_next_condition_handler(
          OUT handler: exec$t_condition_handler;
          OUT handler_type: exec$t_condition_handler_type;
          )RETURNS status;
```

The *handler* argument returns a procedure variable for the next handler found. A warning status is returned if no handler is found.

The *handler_type* argument identifies the type of handler.

This service is called from user mode by using the *exec$get_next_condition_handler* system service, which locates the associated procedure variable in the thread's PCR, changes its own call frame to look like an *e$get_next_condition_handler* call frame, and jumps to the entry point of *e$get_next_condition_handler*.

This service returns an error status if no condition handler was active at the time it was called.

\This service is provided for facilities such as the debugger. It is intended to localize all code that implements the condition handling algorithm for finding condition handlers. It is implemented as a part of the condition dispatcher to keep the handler locating algorithm in one place in the system.\

1.2.11 Miscellaneous Mica Library Services

Other than *exec$raise_condition*, no Mica library services are currently proposed for condition handling.

1.2.12 Condition Values

\This section has been moved from this chapter to Chapter 3, Status Codes and Messages.\

Condition names/values are a subset of status values.

1.2.12.1 Condition Values Specific to Condition Handling

When a handler returns to the dispatcher, it returns the status of *exec$_continue* if it is possible to continue from the condition. If a handler wishes to reraise, it returns *exec$_reraise*.

When *e$unwind* calls a handler, if no condition record is specified, it allocates a small condition record with the primary condition value *exec$_unwinding*. This indicates that an unwind is taking place.

1.3 Exit Handling

1.3.1 Introduction

Mica exit handling is a facility that gives threads the capability of specifying and executing procedures in user mode during thread rundown. This facility provides a controlled method for threads to perform cleanup or emergency actions.

Exit handling is both thread and process based. Each thread has an independent list of thread exit handlers which only that thread may modify at any time prior to thread rundown. For a process, there is a list of process exit handlers which may be modified by any thread in that process at any time prior to overall process rundown. A process exit handler list is not associated with any thread in that process or with any thread's individual exit handler list.

Note that the last thread to exit a process is used to execute the process exit handlers.

Typical use of exit handling is file cleanup, network cleanup, resetting terminal characteristics, and so on.

1.3.2 Goals

1. Exit handling is thread and process based:
   a. No thread within a process may influence the exit handling of any other thread within that or any other process. Thread exit handlers are managed and executed on a per-thread basis.
   b. Any thread within a process may make changes to the process-wide exit handler list. No other process's threads may influence this list.
2. All established thread exit handlers are always called when a thread exits.
3. All established process exit handlers are always called when the last thread in a process has finished calling its own thread exit handlers.
4. A thread's exit handler list is securely maintained, with strict control of access by the owning thread.
5. Exit handlers and lists exist for and run in user mode only; there are no kernel-mode exit handlers.
6. At any one time, a thread may have as many thread exit handlers established as it has system resources available to allocate and manage them.
7. At any one time, a process may have as many thread exit handlers established as it has system resources available to allocate and manage them.
8. For a forced exit, it is impossible for a thread to continue execution of exit handlers indefinitely.
9. Exit handling must be usable by the Mica debug facility.

1.3.3 Design Description

When a thread's exit has been requested using one of the Exit system services, that thread's exit handlers are called. Exit handlers are procedures called by the system in user mode using the standard call interface. They may be established at any time during normal thread execution and are managed as entries in a linked list held in a thread's TCR (for thread exit handlers) or in the PCR (for process exit handlers).

Once all thread exit handlers have been called, if the thread is not the last thread in the process, it continues with the rest of thread rundown. Otherwise, when the last thread in a process has completed calling its thread exit handlers, that thread is used to call the process exit handlers. Once these have been called, the thread continues with thread rundown, and if it is the last thread, process rundown.

The mechanism for calling exit handlers is complicated by the existence of two forms of thread exit: normal exit and forced exit.

1.3.3.1 Normal Exit

A thread normally requests its own exit by using the *exec$terminate* library service. This service causes an exit unwind to take place with either an optional input condition record describing the exit, or a manufactured condition record with the single status value: *exec$_exiting* (denoting normal thread exit). This procedure is described in Section 1.3.5.

Once the unwind has completed, *e$unwind* calls *exec$exit* with the condition record as an argument. The *exec$exit* system service locates the first exit handler entry in the TCR (or the PCR, if none exists in the TCR and this is the last thread in the process) and calls the exit handler in user mode.

If that exit handler completes normally and returns, it returns to user-executable code in system space, which calls *exec$exit* with the original condition record. This causes the next exit handler to be dequeued and called. The algorithm is continued until the exit handler list is exhausted.

At any time, a handler may also call *exec$exit* to finish its processing. This allows the handler to replace the condition record being used for the exit.

All handlers called are allowed to consume as much of the thread's remaining CPU quota as they require. Should a handler exceed the quota, it is terminated, the thread object marked that a forced exit is taking place and the next exit handler dequeued from the appropriate list and called in user mode. The exit then proceeds as a forced exit.

Should any handler enter a wait state, the exit stalls until either the wait is satisfied, or a forced exit for the thread is requested. Any number of forced exits may be requested: each terminates the currently running exit handler (or the unwind) and causes the next handler to be called (if there are no more, then rundown continues).

1.3.3.2 Forced Exit

A thread may be forced to exit by a call to either the *exec$force_exit_thread* or *exec$force_exit_process* system services. The *exec$force_exit_process* service essentially performs the actions of *exec$force_exit_thread* for each thread in the target process.

The *exec$force_exit_thread* service causes a user-mode AST to be queued to the target thread. If the target is the calling thread, a user-mode AST is immediately to be queued to the thread, and a timer started to ensure delivery of the AST. If another thread has called *exec$force_exit_thread*, then a kernel-mode AST is queued to the target thread which, when delivered, queues a user-mode AST to the thread and sets up the timer. When the user=-mode AST is queued, the thread object is marked to indicate a forced exit is in progress.

The original status value argument to the *exec$force_exit_thread* system service is the first AST argument, and the second AST argument is set to the status value *exec$_forced_exit*.

If the AST is not delivered within the timeout period, the thread's user-mode AST queue is flushed, and the AST is requeued with the second parameter set to *exec$_forced_exit_asts_flushed*. The thread's user-mode ASTs are then enabled and the elapsed timer is reenabled.

The user-mode AST is delivered to a special system procedure. This procedure is mapped in system space, but is user-mode executable. On execution, this procedure allocates a condition record with two condition vectors, each containing one of the AST arguments as condition values. It then requests an exit unwind with this condition record.

When the unwind completes, *e$unwind* calls *exec$exit*, causing the exit handlers to be called. Unlike the normal exit case, each exit handler is given a finite CPU quota to use. Exhaustion of this quota or a subsequent forced exit causes the next exit handler to be dequeued and called, until all exit handlers have been dequeued and called.

Note that the thread is also given a finite CPU quota to perform its unwind. Should the thread fail to complete the unwind due to quota exhaustion or a second forced exit taking place, the first exit handler is dequeued and the handler called (with a renewed but finite quota).

1.3.3.3 Restrictions when Executing Exit Handlers

Once thread rundown has commenced, a thread is not allowed to establish any more exit handlers. This restriction is necessary so that once rundown has started, it may not be prolonged by a pathological thread that continuously establishes new exit handlers.

For forced exits, threads also may not create any new threads within their process.

The *e$unwind* service returns an error if it detects an attempt to unwind to an ancestor of an exit handler invocation.

Calling the Exit system service during exit handler processing has the effect of returning from the active handler and replacing the condition record that the handler may have been given as an input argument. The system then looks for the next eligible exit handler to call. Note that this includes any thread exit requests that might occur as a result of raising an exception during exit handling. This is described in Section 1.3.5.

1.3.4 Implementation Description

1.3.4.4 Establishing and Deleting Exit Handlers

1.3.4.4.1 exec$establish_exit_handler

Exit handlers are established by allocating a controlling data structure (the exit handler cell) from the TCR or PCR, filling it in with a pointer to the procedure variable defining the handler (and optionally a user-supplied record pointer), and linking it into the exit handler list maintained in either of those two regions. This is all done by the *exec$establish_exit_handler* system service:

```
PROCEDURE exec$establish_exit_handler(
          IN process_handler: boolean = false;
          IN handler: exec$t_exit_handler OPTIONAL;
          IN OUT exit_cell: POINTER exec$t_exit_handler_cell OPTIONAL;
          IN context_record: POINTER anytype CONFORM = NIL;
          IN beginning: boolean = true;
          )RETURNS status;
```

The caller specifies an exit handling procedure and, if requested, is returned a pointer to the storage used to establish that exit handler in the appropriate control region. The region is determined by the Boolean *process_handler* (FALSE represents the TCR; TRUE represents the PCR).

The optional *context_record* argument may be used to supply a pointer to a user-defined record. If this argument is supplied, the exit handler is called; the pointer is supplied as a parameter.

The optional *beginning* argument specifies whether the handler is to be added to the beginning (that is, called first) or to the end (that is, called last) of the exit handler list. The default, as shown, is the beginning, which (unless overridden) causes handlers to be called in LIFO order).

The *exec$establish_exit_handler* service allows very general manipulation of the exit handler list specified by the *process_handler* Boolean. An entry may be added to either end of the list, reused by specifying the *exit_cell*, or deleted by specifying the *exit_cell* and omitting the *handler* argument. An error is returned if an attempt is made to delete a currently active exit handler.

A procedure may be declared as an exit handler as often as desired. Each declaration generates a new exit handling cell linked in the specified list associated with the thread.

This system service returns an error if issued after a forced exit has started.

1.3.4.5 Requirements on Thread and Process Control Structures

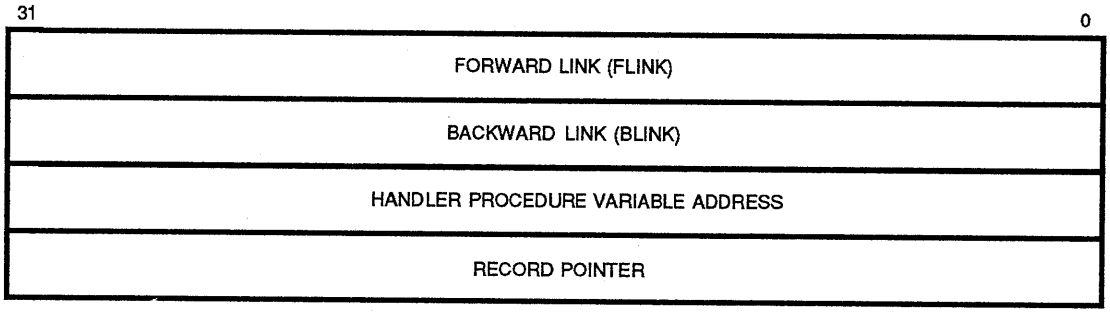
1.3.4.5.1 Thread Object

Exit handling requires that one status flag be defined in a status field within a thread object. This flag is used to indicate that a forced exit is in progress.

1.3.4.5.2 TCR and PCR

Exit handling requires a listhead in both the TCR and PCR regions. The listheads each consist of a list of pointers for each list. Data structures are allocated within these regions and placed in these lists.

Figure 1–1: Exit Handler Cell



The Handler Procedure Variable Address field is used to store a pointer to the procedure variable for the associated exit handler.

The FLINK and BLINK entries contain the virtual addresses of the next and previous exit handler cells, except for the first and last entries in the list(s).

The Record Pointer field holds the optional pointer argument of the system service. Note that if no argument is specified, this field is zero.

1.3.4.6 Finding and Calling Exit Handlers

When a thread enters rundown, the system first searches the thread exit handler list. As each handler is found, it is removed from the front of the list and called in user mode, and the storage is deallocated back to the region. Once the list is empty, if this is the last thread in the process, the process-wide exit handler list is handled in the same way.

Exit handlers are defined as the following procedure type:

```
TYPE
    exec$t_exit_handler : PROCEDURE(
            IN condition_record: exec$condition_record;
             IN context_record: POINTER anytype CONFORM = NIL;
             );
```

The *condition_record* argument is used to pass the handler the condition record associated with this exit. The *context_record* argument is used to pass in a record pointer, if one was specified when the exit handler was established.

When an exit handler finishes processing, it either executes an *exec$exit* system service, or returns. If it returns, it returns to an address in system space where an *exec$exit* system service is called.

Calling *exec$exit* simply finds the next exit handler and calls it. No status is kept between calls to exit handlers. Once both lists are empty, the last call to *exec$exit* causes the exit process to continue with the next phase of thread rundown.

Exit handlers are always called with ASTs disabled.

Summarizing, the algorithm is:

```
exit_handling:
!  User-mode exit handlers:

   set no handler cell found
   if either the thread or process handler queues are not empty
      if thread exit handler queue is not empty
         get pointer to handler cell from head of queue
         remove handler cell from queue
        else
         if process exit handler queue is not empty AND this is the
                                          last thread in this process
            get pointer to handler cell from head of queue
            remove handler cell from queue
         end
      end
      if handler cell found
         if a forced exit
            set finite time for thread's CPU quota
         end
         allocate frame on thread user-mode stack for call to
                           the e$user_exit system procedure
         disable user-mode ASTs
         allocate argument list for this call on thread user-mode
                                                            stack
         put exit handler procedure variable in argument list
         set up condition record as input argument for this call
         get optional context_record point from cell and place
                                          in call argument list
         deallocate storage of cell back to region
         push PS/PC pair onto kernel stack to REI to the
                                 entry point of e$user_exit
         REI
      end
   end
   clear thread control block exit handling in progress

   clear thread object of exit handling in progress
return  ! to rest of thread rundown

!  Procedure e$user_exit is a procedure mapped in system space but
!  user-mode executable. It has the following definition:
!
!  PROCEDURE e$user_exit(
!            IN handler : exec$t_exit_handler;
!            IN condition_record: exec$condition_record;
!            IN context_record : POINTER anytype CONFORM OPTIONAL;
!            );
!
!  This procedure is implemented using the following algorithm and
!  exists only to call the condition handler, perform an exit system
!  service should the handler return, and to ensure that the user-mode
!  stack always has a consistent state should an AST be delivered.
!
!  This procedure is not documented, is not necessarily at the same
!  virtual address for each Mica release, and is not vectored to via
!  the PCR since no user-written code will ever directly call it.
!
!  It's presence on the stack alerts e$unwind should an attempt to
!  unwind out of exit handling occur.
```

```
e$user_exit:
      call handler with the condition_record and context_record
                                                          arguments
      exec$exit(condition_record)       ! uses exit service
                                  ! to get to go back for
                                  ! next handler
```

\The following facts currently apply to the above algorithm:

1. The "finite time" setting in the above algorithm has yet to be specified. It may vary depending on the underlying PRISM processor speed.
2. The above algorithm uses the *exec$exit* system service to continue the exit handler search in kernel mode.
3. Part of the exit handling dispatcher runs in user mode, but is located in system space. All threads require execute access to this procedure from user modes.\

1.3.4.7 Returning from Exit Handlers

Exit handlers always eventually return to the exit handler dispatcher in kernel mode. They return no status.

1.3.5 Exit System Services

The Exit system services are discussed in Chapter 5, Process Structure. However, threads should never use these services to cause their own exit. There is a library service, *exec$terminate*, which should always be used to request normal thread exit.

Unlike the system service, the library service simply causes an unwind to take place using either an input condition record, or manufacturing a condition record with a single condition vector containing the single status value: *exec$_exiting*.

The procedure declaration is as follows:

```
PROCEDURE exec$terminate(
          IN condition_record: exec$condition_record OPTIONAL;
          )RETURNS status;
```

If a condition record is specified, the primary status value must be *exec$_exiting*. An error is returned in all other cases.

Given a successful call, this service does not return to its caller. Once the unwind has finished, the unwind algorithm calls the system service *exec$exit*, which then calls the first established exit handler.

If a call to *exec$exit* occurs after the system has called an exit handler, it cleans the stack to before the first exit handler frame it finds (only one may be active at a time), and returns to the exit handler dispatcher to search for the next exit handler. If none is found, the rest of thread rundown is started.

The *exec$exit* system service uses the following algorithm:

```
exec$exit:
   remove exec$exit call frame from user-mode stack
   if call frame left on user-mode stack = e$user_exit frame
      remove e$user_exit frame from user-mode stack
   end
   call exit_handling

   [remainder of thread rundown code]
```

1.4 User-Mode AST Handling

1.4.1 Introduction

Mica allows a thread to establish procedures to be called in response to events asynchronous to the thread's execution. When such an association has been made and the event occurs, the thread's normal execution is interrupted and the procedure is called. This is called an *asynchronous system trap* (AST) and the procedure is called an AST handler. When the handler is called, the AST is said to have been delivered.

AST handlers may be established for a variety of events in the system. They may be established during kernel-mode or user-mode thread execution. Establishing an AST handler causes an AST object to be allocated in system space. When the associated event occurs, the object is placed in the thread's FIFO AST pending queue for the mode from which the handler was established. Each thread in the system has two queues, one for kernel-mode ASTs and one for user-mode ASTs, and each queue entry represents a pending AST for a thread. AST handlers always execute in the mode from which they were established.

Kernel-mode ASTs are delivered and handled prior to user-mode ASTs. A pending kernel-mode AST will interrupt an AST being handled in user mode. A pending user-mode AST will not interrupt the handling of a kernel-mode AST, nor will it interrupt the execution of a user-mode AST handler.

The behavior of the two kinds of kernel-mode ASTs, handling and exiting from such ASTs, and the details of the precise queuing and locking strategies implemented by the Mica kernel for AST objects are described in Chapter 6, The Kernel.

This chapter is only concerned with describing the delivery, behavior, and completion of user-mode ASTs.

1.4.2 Implementation Description

1.4.2.1 AST Handlers

When a user-mode AST handler is established, an optional argument may be specified. When the AST is delivered, if the argument was specified, the handler is called with this as its first argument. Depending on the AST, there may also be a second argument, which is supplied by the facility which requested the delivery of the AST.

These arguments have the following type:

```
exec$t_ast_argument : longword ;
```

\Note that a type definition is supplied for AST arguments, since the definition of the argument should be independent of whether the underlying machine is a 32-bit or 64-bit PRISM system.\

A user-mode AST handler is defined as follows:

```
exec$t_ast_handler : PROCEDURE(
            IN ast_arg_1 : exec$t_ast_argument = DEFAULT;
            IN ast_arg_2 : exec$t_ast_argument = DEFAULT;
            );
```

There are no restrictions on the system facilities available to an AST handler. All services are available, including the Exit service. The only indication to the Mica kernel/executive that a thread is executing an AST handler is a flag that is set in the thread object during the delivery of an AST. When the handler completes, this flag is reset, and the pending AST queues for both kernel and user mode are checked. Any pending ASTs are immediately delivered, using the same delivery mechanism that was used for the completing AST.

1.4.2.2 Delivering and Completing a User-Mode AST

A single thread may have many different procedures established as user-mode AST handlers for many different types of events. At any given time, a thread may have many user-mode ASTs pending. They are held in a queue and are not delivered unless the thread is executing normally (that is, the thread is not already handling an AST) and has user-mode AST delivery enabled. Thus, ASTs are mutually exclusive, and at any time a thread may disable their delivery.

Disabling or enabling the delivery of user-mode ASTs on a PRISM system is accomplished by using the SWASTEN instruction. This instruction is used to store the current AST delivery state (enabled or disabled) and to change the state. This is accomplished efficiently without a context switch into kernel mode. The instruction is fully described in Chapter 4 of the PRISM SRM.

A user-mode AST is queued to a thread in kernel mode. When an AST object is queued, if the thread is not already executing a user-mode AST handler (determined by examining the state of a flag in the thread object), delivery of the AST is started by setting the user-mode AST Interrupt Request bit in the thread's ASTRR register. Note that for threads not currently executing on a processor, this bit is set in the saved context for the thread.

When the thread next runs in user mode, with user-mode ASTs enabled, the AST interrupt immediately occurs and is delivered in kernel mode. A search of the AST queue takes place in kernel mode and, eventually, the AST is delivered in user mode. Immediately prior to the delivery in user mode, a flag is set in the thread object to indicate that a user-mode AST handler is executing.

If the thread was already executing a user-mode AST handler, then the AST object is simply left in the queue. No further action need be taken since, as part of the current handler completing, a check is made for pending user-mode ASTs, and therefore, an AST interrupt request is made.

In summary, delivering a user-mode AST involves:

1. Dequeuing the AST object
2. Saving the thread's current user-mode register context
3. Marking the thread object to indicate an AST is being processed
4. Causing the AST handler to be called in user mode

Conversely, completing a user-mode AST involves:

1. Resetting the user-mode AST In Progress flag in the thread object
2. Checking for any pending user-mode ASTs and, if there are any, requesting a user-mode AST interrupt
3. Restoring the previous user-mode register context and continuing thread execution in user mode from where it was interrupted

Several events and mode transitions occur during AST delivery: an interrupt from user to kernel mode; effecting a call in user mode to the AST handler; and subsequently dismissing the AST in kernel mode, with an associated check for other pending ASTs.

Mica implements these actions by providing the AST dispatcher procedure and a single system service used by the AST dispatcher, *exec$finish_ast*.

1.4.2.3 AST System Services

1.4.2.3.1 exec$finish_ast

The *exec$finish_ast* system service is used by the AST dispatcher to complete an AST. It has no arguments:

```
PROCEDURE exec$finish_ast(
          )RETURNS status;
```

This service uses the *k$exit_ast* procedure to carry out the following functions in kernel mode:

1. Clear the user-mode AST In Progress flag in the thread object.
2. Check if the user-mode pending AST queue is empty for this thread; if not empty, request a user-mode AST interrupt using the ASTRR register.

   Note that this requires obtaining a lock on the kernel dispatcher's database.
3. Complete the system service request by returning to user mode.

This service always returns success. If no user-mode AST was in progress when it was called, it effectively does nothing.

Note that this system service may be called at any time. In particular, an AST handler may allow other ASTs to be delivered by using this service. This effectively allows an AST handler to make itself interruptible.

1.4.2.3.2 The AST Dispatcher Service: exec$ast_dispatcher

The AST dispatcher has the following definition:

```
PROCEDURE exec$ast_dispatcher(
          IN ast_handler: exec$t_ast_handler;
          IN ast_arg_1:  exec$t_ast_argument;
          IN ast_arg_2:  exec$t_ast_argument;
          );
```

The main function of the AST dispatcher is to contain the knowledge of AST delivery and preserve the consistency of the thread's user-mode stack. When delivering a user-mode AST, instead of arranging a call directly to the user-established AST handler, the kernel arranges a call in user mode to the AST dispatcher. The dispatcher then provides the following functions:

1. The dispatcher's call frame on the thread's user-mode stack is used to hold the AST arguments, the saved user-mode context, and the address of the invocation descriptor of the user-mode AST handler to be called. When the frame is built, these arguments are saved as part of the stack space allocated for the frame.
2. Establishes a descriptor-based handler for dealing with possible unwind requests while the AST handler is executing.
3. Calls the user-mode AST handler, which returns to the dispatcher when finished.
4. Completes the AST using the *exec$finish_ast* system service, which resets the user-mode AST In Progress flag in the thread object and causes any pending ASTs to be delivered.
5. Returns to the thread's point of execution when the AST interrupt took place.

The AST dispatcher guarantees that at all times the thread's user-mode stack is consistent with respect to the PRISM Calling Standard. This eliminates the need for any other user-mode system facility (such as condition handling) to be explicitly aware that an AST is in progress. Thus, user-mode AST handling is provided as an integral part of the thread's environment, and all special processing surrounding the algorithms used to implement this facility are self-contained.

The creation of the call frame is critical: it allows the established condition handler to correctly deal with an unwind. When an unwind takes place, the condition handler cleans up the environment of the AST dispatcher, calls *exec$finish_ast* to complete the AST processing, and returns to *e$unwind*. Thus, unwinding can always deal with active AST handlers, and at any time, an AST may be delivered during an unwind. The key element is to always keep the thread's stack in a known state.

1.4.3 Algorithm Description

Given a pending user-mode AST for a thread which has user-mode AST delivery enabled, when the thread next executes in user mode, an AST interrupt is generated. This interrupt is delivered in kernel mode. The thread's AST queues (both kernel-mode and user-mode queues) are examined. Once all kernel-mode ASTs are delivered, the user-mode AST is delivered in the following steps:

1. If no user-mode AST is active: dequeue a user-mode AST object and set the AST In Progress flag in the thread object.

   Otherwise, just REI from this interrupt, since the pending AST is delivered when the thread's current user-mode AST completes.

2. If the user-mode AST established a kernel-mode "piggy-back" AST handler, that handler is called, and the delivery of the user-mode AST continues when that handler completes.

3. To deliver the AST to user mode, carry out the following actions on the thread's current user-mode stack:

   a. Build a call frame on the thread's user-mode stack for the AST dispatcher procedure.

   b. Put the AST arguments together with a pointer to the invocation descriptor for the AST handler specified for this AST in the call frame.

   c. Save all the thread's current user-mode registers in the call frame and set up the user-mode registers for entry into the AST dispatcher upon execution of an REI instruction.

      Note that whatever procedure was executing at the time of the AST interrupt into kernel mode now appears to be the caller of the AST dispatcher, with a return address set to wherever that procedure was interrupted.

4. Clean up the thread's kernel stack. Modify the PC and PS currently on the kernel stack to cause an REI to continue user-mode execution at the entry point of the AST dispatcher.

5. Execute an REI instruction to return to user mode.

At this point, the delivery of the AST continues in user mode:

1. Enter the AST dispatcher as if it had been called by the procedure that has been interrupted by the AST delivery.

2. The AST dispatcher calls the user's AST handler with the AST arguments located in its call frame.

3. When the user handler returns, the AST dispatcher calls the *exec$finish_ast* system service to complete the AST processing in kernel mode.

4. Finally, on return from the system service, the AST dispatcher restores the previous user-mode register context (from its stack frame) and uses an REI instruction to resume the thread's previous execution. (Note that this might in fact be in the system service entry page if the thread was previously in a wait state and the service needed to be restarted.)

GLOSSARY

AST: See asynchronous system trap.

AST handler: A procedure that is intended to receive notification of a user-mode AST. These procedures are part of the program and are associated with a particular event or system service completion notification required by the thread during its execution.

AST handling facility: The Mica AST handling facility provides a mechanism for delivering asynchronous event notification in user mode to threads.

asynchronous system trap (AST): An event that occurs asynchronous to a thread's execution, causing the thread's normal execution to be interrupted and an AST handler to be called. An AST cannot occur unless the thread has established an AST handler for it.

condition: An error state that results from an error encountered during thread execution. When a condition is raised, the thread's execution is interrupted and the thread starts executing a system-supplied dispatch procedure, which locates a condition handler.

condition handler: A procedure written as a part of a program or supplied by a run-time facility to handle conditions if they occur during the execution of that program.

condition handling facility: The Mica condition handling facility provides the mechanism by which condition handlers are found and established (either at runtime or compile time). This facility provides a mechanism by which all error conditions encountered during a thread's execution may be reliably handled by the thread in a controlled manner.

condition record: A data structure that contains all condition values and arguments associated with a condition.

condition vector: Each quadword-aligned entry in a condition record. All condition-specific arguments present in a condition vector are in descriptor format.

dispatcher: A system-supplied dispatch procedure that locates and calls condition handlers. The dispatcher executes as if it had been called immediately after a condition was raised.

exit handlers: There are two types of exit handlers: thread and process exit handlers. Thread exit handlers are called when a thread exits. Process exit handlers are called when the last thread in a process has finished executing the last of its thread exit handlers. Exit handlers are established using a system service and kept as a list in either the TCR (for thread exit handlers) or the PCR (for process exit handlers).

exit handling facility: The Mica exit handling facility gives threads the capability of specifying and executing procedures in user mode during thread rundown. This facility allows threads and processes to perform overall clean-up actions on their environment, deallocation of system resources, or emergency actions.

hardware conditions: Conditions that occur when a thread attempts some action defined as incorrect, impossible, or not yet possible by the hardware. Such action results in a hardware exception interrupting execution, which in turn causes a condition to be raised in the thread which was executing.

invocation descriptor-based handlers: These handlers are located from a procedure's invocation descriptor. They are used to implement a particular language's condition handling semantics. For the Pillar language, they are used to implement structured condition handling. Invocation descriptor-based handlers are established at compile time and may be called multiple times when multiple conditions are active.

mechanism record: A data structure that contains information regarding the environment at the time of a condition, together with the environment of a handler when it is called.

software conditions: Conditions that result from an explicit use of condition handling by a thread. Software conditions may be raised at any point during thread execution. This allows applications or language run-time libraries to notify threads that some action defined as incorrect, impossible, or not yet possible was attempted by the thread. In Mica, software conditions may occur synchronously and asynchronously to thread execution.

system catchall handler: A user-executable procedure, mapped in system space, that catches all improperly handled conditions.

unwind facility: The Mica unwind facility centrally provides the capability to perform nonlocal GOTOs within a thread. It is implemented as a user-mode procedure, mapped in system space, and reached via a procedure variable in the process control region.

vectored handlers: There are two types of vectored handlers: primary and last chance. Primary vectored handlers are the first searched for when a condition is raised. The list of primary handlers is called in FIFO order with respect to when they were established. Last chance vectored handlers are called in LIFO order with respect to when they were established. Vectored handlers may only be established at runtime, by using a system service.