DATE: March 29, 1979

TO: R & D Personnel

FROM: Bradford E. Hampson

SUBJECT: Specifications for PRIMOS Condition Mechanism

REFERENCE: None

Abstract

PE-T-468, Rev. 2 documents the PRIMOS Condition Handling Mechanism. Both user-level and implementation-level information is included. The Condition Mechanism represents a major departure from prior methods of handling runtime errors in PRIMOS.

Table of Contents

1 Introduction to the Condition Mechanism

The notion of the "condition" comes from the corresponding concept in the PL/I language. A condition is basically an unscheduled software procedure call (or block activation) which is brought about by the occurrence of some unusual event in a process. Examples of this are hardware-defined faults (such as arithmetic exceptions), detection by a subroutine of a hopeless error situation which cannnot be adequately described to the subroutine's caller through available parameters, and Quits from the user terminal (converted by Primos into a hardware-defined fault).

Conditions are particularly useful in two basic circumstances: when caused by an unpredictable hardware fault, and when the call-return flow of control is not known by the routine detecting the unusual happening.

The implementation of the condition mechanism described here is intended to serve three purposes: to provide a consistent and useful means for system software to handle error conditions and to manage a reentrant/recursive command environment; to provide user programs with the capability to handle error conditions that heretofore forced a return to command level; and to provide support for the condition mechanism of ANSI PL/I.

2 On-Units

An "on-unit" is a handler for a condition, and may either be a prodecure (an "entry variable" in PL/I terminology), or a begin-block. The latter result from execution of the PL/I <on statement>, while the former result from explicit invocation of the system primitives mkonu$ and mkon$f. The only way to cause creation of an on-unit in a non-PL/I program is to explicitly call the system primitives mkonu$ or mkon$f. At various times, system software will create its own on-units for system-defined conditions.

A procedure may also act to invalidate, or "revert", an on-unit it had previously created. In PL/I, this can be done by means of the <revert statement>. The reversion of an on-unit can also be accomplished by calling the system primitives rvonu$ or rvon$f. Note that such a reversion applies to the current activation only; the on-unit(s) for the same condition created by activations earlier in the stack are not affected.

Every on-unit is associated with a given activation, which is simply the particular invocation of the procedure (or begin block) that requested creation of the on-unit. Associated with every on-unit is the name of the condition for which the on-unit is a handler. These condition names are character strings of up to 32 characters, and may represent system-defined conditions if the name is one of those reserved for system use, or it may be a user-defined condition.

The condition mechanism is activated whenever a condition is raised. In PL/I terms, a condition is raised either explicitly as the result of a <signal statement>, or implicitly as the result of some error having been detected during regular computation. Note that in this context, "error" is to be taken loosely: "end of file" is one such "error".

A condition may be raised by the execution of a PL/I <signal statement>, or by an explicit call to the system primitives signl$ or sgnl$f. In some cases, such as conversion of hardware faults to conditions, this call is executed by system software and so is invisible to the user: from the user's point of view, a hardware fault is a condition signal.

3 Invocation of On-Units

When a condition is raised, the Condition Mechanism must first find an on-unit for that condition. It does this by searching the stack backward in time, starting from the activation belonging to the prodecure that raised the condition. If an activation has an on-unit for the specific condition name that was raised, that on-unit is selected. If an activation does not have an on-unit for the specific condition, but does have an on-unit for the special condition ANY$, that activation is said to have a default on-unit, and the ANY$ on-unit is selected. Scanning is temporarily suspended at the first activation containing a selectable on-unit, and that on-unit is invoked.

A selected on-unit is invoked according to the following specification:

```
dcl on_unit entry (ptr) variable;

call on_unit (ptr_to_cond_frame);
```

That is, all on-units are passed a single argument which is a pointer to the Standard Condition Frame Header that describes the condition that was raised. Note that an on-unit operates as if it had been invoked from the activation which created it, so that if the on-unit procedure is internal to that activation, the normal PL/I scoping rules for automatic storage (and all other storage classes) apply.

4 Possible Actions of an On-Unit

In the general case, an on-unit has several options as to what action it can take. It may, of course, perform any desired application-specific tasks, such as closing file units, deleting temporary files, updating databases, doing consistency checks, and so on in order to abort the computation that has been interrupted. In many cases, however, it may be possible for the on-unit to repair the cause of the condition (or even to determine that the condition can be safely ignored), or to decide that the computation's normal flow can be interrupted and the program reentered at some "known" point.

If permitted by the signaller of the condition, the on-unit may simply return, in which case the computation will continue from the point of the signal or hardware fault. If the signaller has forbidden such a return, an attempt to do it will result in a signal of the condition ILLEGAL_ONUNIT_RETURN$. There are information bits in the condition frame header (see below) that inform the on-unit as to whether return is permitted (this information is implicit in the name of most system-defined conditions), and whether it will in general make sense to return without having taken "corrective action".

An on-unit may perform a nonlocal goto to some previously defined label, that will cause the program to restart in some known state. This action may always be taken, as the activation that caused the condition to be raised will usually be aborted by the nonlocal goto, and hence the issue of on-unit return does not arise.

An on-unit may also signal another (possibly the same, but beware of infinite recursion) condition, and it may either permit or deny return from that condition.

The on-unit may decide to get the process to command level, to allow the user to take control. The system default on-unit is an example of an on-unit that does exactly that. If the user enters the "start" command, the new command level will return to the invoking on-unit.

Finally, the on-unit may decide that it has not been able to handle the condition after all, or that it has only partially done so, and needs help from an on-unit established by one of its dynamic ancestors. The on-unit instructs the condition mechanism that it desires to "continue to signal", and then simply returns. The condition mechanism will then continue to scan up the stack for more on-units for the condition.

5 Using the Condition Mechanism from Fortran

Since Fortran is not a block-structured language, the use of on-units (condition handlers) and of nonlocal goto's from Fortran is somewhat restricted. In particular, there are no internal procedures or blocks in Fortran, leaving external procedures (subroutines) as the only possibilities for service as on-units. The fact that a Fortran statement label value does not contain an activation (stack) pointer means that nonlocal goto's work in a way different from PL/I (see below).

To summarize the restrictions:

- Fortran on-units must be SUBROUTINEs, which may not, of course, be internal to the subroutine or main program making the on-unit.
- Nonlocal goto's are defined in Prime Fortran to work only if the target statement label belongs to the caller of the subroutine performing the nonlocal goto. That is, nonlocal

goto's work only to the previous stack level.

5.1 Datatype Incompatibilities

The PL/I interfaces to the condition mechanism utilize the PL/I datatype "character(*) varying". This datatype in not available in Fortran (either 1966 or 1977 ANSI standard). The 1977 ANSI Fortran, however, includes a datatype that is the equivalent of PL/I "character(*) nonvarying".

For these reasons, a set of interfaces to the condition mechanism is provided which utilizes nonvarying character strings. The user is cautioned that these interfaces will not be as efficient as those using the varying character strings. It is possible to simulate varying character strings in Fortran by using appropriate equivalences.

The Prime representation for a varying character string is equivalent to an integer*2 array in which the first element contains the character count, and the remaining elements contain the characters in packed format. To illustrate:

```
PL/I:
  dcl name char(5) varying static initial ('QUIT$');

Fortran:
  INTEGER*2 NAME(4)
  DATA NAME /5, 'QUIT$'/
```

5.2 Interfaces for Nonlocal Goto's

A full-function nonlocal goto requires that the target label identify both a statement and an activation (stack frame) of the program that contains the statement. If such a nonlocal goto is required in a Fortran program, the following interfaces can be used.

The procedure MKLB$F is called by a program to create a PL/I-compatible label pointing to one of its statements. The activation pointer in the label will identify the caller's activation.

The procedure PL1$NL will perform a nonlocal goto to a specified target label. Labels produced by MKLB$F are acceptable to PL1$NL.

The calling sequences for these routines are described below.

6 Default On-Units and Cleanup On-Units

The CLEANUP$ Condition

The special condition CLEANUP$ is used only during processing of a nonlocal goto (or a crawlout from an inner ring). Any activation may make an on-unit for the condition CLEANUP$, which will be invoked only when that activation is about to be aborted by a crawlout or simple nonlocal goto.

The CLEANUP$ on-units of the activations on the stack are invoked in reverse chronological order. Each CLEANUP$ on-unit is expected to return unless it encounters a fatal error. No activation's stack frame is removed from the stack until all activations have been cleaned up (i.e. all CLEANUP$ on-units have returned).

An on-unit for CLEANUP$ may perform any operation desired, but most common will be such things as closing file units, freeing generations of storage that have been allocated in static areas, and so on.

The Condition mechanism cannot guarantee that the CLEANUP$ on-unit in a given activation will not be invoked more than once, but it can and does guarantee that, once a CLEANUP$ on-unit has returned to the Condition Mechanism and that activation has been marked "cleaned up", invocation of any of that activation's on-units (including CLEANUP$), as well as transfer of control into that activation by means of a further nonlocal goto, are prevented.

The ANY$ Condition

An on-unit for the condition ANY$ is called a "default on-unit". A procedure creates an on-unit for ANY$ in the normal manner (PL/I <on statement>, or a call to mkonu$ or mkon$f), whenever it wishes to intercept any condition that might be signalled during its activation.

When a given activation is reached during the stack scan associated with the raising of a condition, it is first examined for an on-unit for that specific condition. That on-unit is selected for invocation if it exists. If the activation has no specific on-unit, but does have an on-unit for ANY$, then the ANY$ on-unit is selected for invocation. The Standard Condition Frame Header passed to the ANY$ on-unit describes the original condition because of which the on-unit is being invoked.

Hence, a procedure's default (ANY$) on-unit is invoked only if the procedure has no specific on-unit for the given condition.

User programs should not include an ANY$ on-unit unless truly necessary. A user ANY$ on-unit should not attempt to handle most system conditions, but rather should "pass them on" by simply returning. The continue switch (cfh.cflags.continue_sw) is set automatically whenever an ANY$ on-unit is invoked. Any user ANY$ on-unit that fails to return with the continue switch still set, may

cause IMPROPER OPERATION of user or Prime software at some future release of the system.

7 System Primitive Interfaces to the Condition Mechanism

The following section documents new dynamically-linked calls that have been made available so that V-mode programs may use the Condition Mechanism. Note that it is not possible for R-mode programs to use any of these interfaces.

---------- Signal Specific Condition ----------
| signl$ | | signl$ |
---------- System Primitive 03-09-79 ----------

Name: signl$

Purpose:

The primitive signl$ is called in order to raise a specific condition in the ring of the caller. The stack is scanned backwards in order to find an on-unit for this condition, or a default (ANY$) on-unit. The first such on-unit found is invoked according to the specification given in PE-T-468, "Invocation of On-Units". The on-unit has the option of signalling another condition; of calling the primitive cnsig$ to request that the stack scan for on-units continue and then returning; of performing a nonlocal goto; and it may have the option of simply returning, in which case the processing of the condition is considered complete and signl$ returns to its caller.

If signl$ is called from a procedure executing in an inner ring (a ring less than 3), and no on-unit or default on-unit is found in that inner ring, an event known as a crawlout occurs. See "PE-T-468, The Crawlout Mechanism" for further details.

Usage:

```
dcl signl$ entry (char(*) var, ptr, fixed bin, ptr, fixed bin,
                  bit(16) aligned);

call signl$ (condition_name, ms_ptr, ms_len, info_ptr,
             info_len, action);
```

condition_name
is the name of the condition to be signalled. (Input)

ms_ptr
is a pointer to an sfh, ffh or cfh structure defining the machine state at the time the event occurred which makes necessary this call to signl$. If ms_ptr is null, signl$ will use a pointer to the cfh produced by the call to signl$. (Input)

ms_len
is the length in words of the structure pointed to by ms_ptr. If ms_ptr is null, the value of ms_len is not examined. (Input)

info_ptr
is a pointer to an arbitrary structure containing auxiliary information about the condition. The format of this structure need be known only to those procedures that will raise or handle this condition. If no auxiliary information is

available, info_ptr should be null. (Input)

info_len
is the length of the structure pointed to by info_ptr, in words. If info_ptr is null, the value of info_len is not examined. (Input)

action
has the following internal structure:

```
dcl 1 action,
      2 return_ok bit(1) unal,
      2 inaction_ok bit(1) unal,
      2 crawlout bit(1) unal,
      2 specifier bit(1) unal,
      2 mbz bit(12) unal;
```

Action.return_ok should be '1'b if the on-unit is to be allowed to return. Action.inaction_ok, if '1'b, informs a potential on-unit that it may return without taking any corrective action and still expect "defined" results. (Return_ok must be '1'b if inaction_ok is '1'b). Action.crawlout is '1'b if this call to signl$ is the result of a crawlout. This bit should never be set by a user program in a call to signl$, but signl$ has no way to enforce this restriction. Action.specifier is '1'b to signal a PLIO condition. The first member of the info structure must be the appropriate specifier pointer. Action.mbz must be '0'b. (Input)

User programs shoud never attempt to signal a PLIO condition (that is, action.specifier should never be '1'b).

The blocks of storage identified by (ms_ptr, ms_len) and (info_ptr, info_len) will be copied onto the outer ring stack if a crawlout occurs. In general, however, a PLIO condition should not be signalled in an inner ring, as the file control block may be inaccessible to the outer ring(s).

mkonu$	Make an On-Unit	mkonu$
	System Primitive 03-09-79	

Name: mkonu$

Purpose:

The primitive mkonu$ is called by a procedure or begin block when it wishes to create an on-unit for a specific condition, or a default on-unit (an on-unit for the condition ANY$).

Usage:

```
dcl mkonu$ entry (char(*) var, entry)
           options (shortcall (18));

call mkonu$ (condition_name, on_unit_entry);
```

condition_name
is the name of the condition for which the activation desires to create an on-unit. If the activation already has an on-unit for this condition, the previous on-unit is overwritten by this new one. (Input)

on_unit_entry
is an entry value representing the on-unit procedure to be invoked when condition_name is raised and this activation is reached in the stack scan. Because mkonu$ does not save the display pointer associated with on_unit_entry, the entry value must be external or internal to the block calling mkonu$. Note that an entry constant that is declared in the block containing the call to mkonu$ must necessarily satisfy these restictions. (Input)

Notes:

The stack frame of the caller is grown, if necessary, to add the descriptor block for the new on-unit.

The value of <condition_name> must not contain trailing blanks.

The caller must guarantee that the generation of storage occupied by condition_name will not be freed until after the caller returns or its activation is aborted by a nonlocal goto. For PL/I callers, this implies that condition_name may not be a constant.

rvonu$	Revert an On-Unit	rvonu$
System Primitive	09-26-78	

Name: rvonu$

Purpose:

This primitive is called by an activation whenever it wishes to revert an on-unit it had previously created. A reverted on-unit is ignored when scanning the stack for an on-unit for a condition that has been raised. The only way to re-instate a reverted on-unit is to issue another call to mkonu$.

Usage:

```
dcl rvonu$ entry (char(*) var);

call rvonu$ (condition_name);
```

condition_name
is the name of the condition whose on-unit in this activation (if any) is to be reverted. (Input)

Notes:

There is no effect if an activation attempts to revert an on-unit for a condition when that activation has no on-unit for the condition, or if that activation had already reverted its on-unit for the condition. In no case will a call to rvonu$ affect on-units in any other activation.

cnsig$	Set Continue-To-Signal Switch	cnsig$
	System Primitive 09-26-78	

Name: cnsig$

Purpose:

The primitive cnsig$ is called by an on-unit when that on-unit has not been able to completely handle the condition because of which it was invoked. After calling cnsig$, the on-unit should return, at which point the Condition Mechanism will resume scanning the stack for more on-units for the condition that was raised.

Usage:

```
dcl cnsig$ entry (fixed bin);

call cnsig$ (status);
```

status
is a standard system error code, and will be nonzero only if there was no condition frame found in the stack in which to set the continue_sw.

Notes:

Multiple calls to cnsig$ by the same on-unit prior to returning to the Condition Mechanism will have the same effect as a single call.

The continue switch is automatically set whenever an ANY$ on-unit is invoked. Such an on-unit, therefore, need not call cnsig$ in order to continue to signal.

---

mkon$f	Make an On-Unit (Fortran)	mkon$f
	System Primitive 03-09-79	

Name: mkon$f

Purpose:

The primitive mkon$f performs the same function as mkonu$; that is, an on-unit for a specified condition name is created for the calling procedure or block. This interface, however, avoids use of varying character strings and so may be more convenient for Fortran and other languages. WARNING: this interface is substantially less efficient than mkonu$, both in terms of stack space and execution time. Applications where these are important should use mkoun$.

Usage:

```
CALL MKON$F (CNAME, CNAMEL, UNIT)
EXTERNAL UNIT
INTEGER*2 CNAME (--), CNAMEL
```

CNAME
is an array containing the name of the condition for which an on-unit is desired. (Input)

CNAMEL
is the length in characters of CNAME. (Input)

UNIT
is an external subroutine (or procedure) which is to become the on-unit handler for this condition. This subroutine will be invoked with one argument as follows:

```
CALL UNIT (CP)
INTEGER*4 CP
```

where CP is a pointer to the condition frame header (cfh) that describes the condition.

Notes:

IMPORTANT: any program compiled by the FTN compiler that makes a call to mkon$f, must include the specification statement "STACK HEADER 34", and be compiled with the -SPO option. This reserves the stack space necessary for on-unit data. If mkonu$ is used, its SHORTCALL specification will reserve the needed space.

The comments in the writeup on mkonu$ apply to mkon$f as well, with the exception that CNAME and CNAMEL may be overwritten by the caller once mkon$f has returned, because they are copied into a stack frame extension by mkon$f.

Note that every call to mkon$f allocates additional stack space to hold a copy of CNAME, even if the caller had previously called mkon$f with the same values of CNAME and CNAMEL.

rvon$f	Revert On-Unit (Fortran)	rvon$f
	System Primitive 03-09-79	

Name: rvon$f

Purpose:

The primitive rvon$f reverts an on-unit for a specific condition in the caller's activation. It is identical in effect to rvonu$. WARNING: this interface is less efficient in execution time (and temporary stack space used) than is rvonu$; time- or space-critical applications may wish to use rvonu$ instead.

Usage:

```
CALL RVON$F (CNAME, CNAMEL)
INTEGER*2 CNAME(--), CNAMEL
```

CNAME
is the name of the condition whose on-unit in the caller's activation is to be reverted. (Input)

CNAMEL
is the length in characters of CNAME. (Input)

Notes:

All comments that apply to rvonu$ also apply to rvon$f.

---

sgnl$f	Signal Specific Condition (Fortran)	sgnl$f
	System Primitive 03-09-79	

Name: sgnl$f

Purpose:

The primitive sgnl$f is used to signal a specific condition, supplying optional auxiliary information with the signal. A call to sgnl$f is equivalent in effect to a call to signl$. WARNING: this interface is less efficient in execution time (and temporary stack space used) than signl$; time- or space-critical applications may wish to use signl$.

Usage:

```
CALL SGNL$F (CNAME, CNAMEL, MSPTR, MSLEN, INFOPT,
             INFOLN, FLAGS)
INTEGER*2 CNAME(--), CNAMEL, MSLEN, INFOLN, FLAGS
INTEGER*4 MSPTR, INFOPT
```

CNAME
: is the name of the condition to be signalled. CNAME is an integer array containing the character string. (Input)

CNAMEL
: is the length of CNAME in characters. (Input)

MSPTR
: is an integer*4 datum containing a hardware Indirect Pointer to a stack frame describing the machine state at the time the condition was detected. User callers will not usually know this value, and if not should pass the null pointer value 7777/0, which as an octal constant is :1777600000. (Input)

MSLEN
: is the length in words of the machine state stack frame header. (Input)

INFOPT
: is a pointer (same format as MSPTR) to a user-supplied information array. This array can be in any format. If the array is contained in the variable X, a pointer to it is passed by the nonstandard expression LOC(X). Callers should pass the null pointer (see above) if no information array is being supplied. (Input)

INFOLN
: is the length in words of the information array pointed to by INFOPT. (Input)

FLAGS
: is an integer datum specifying certain control actions to

SGNL$F. If bit 1 (:100000) is set, the on-unit may return. If bit 2 (:040000) is set, the on-unit need not take any corrective action before returning. All other bits will usually be 0 (but see the writeup on signl$ for a full description). (Input)

---

mklb$f	Make Label Value (Fortran)	mklb$f
	System Primitive 03-09-79	

Name: mklb$f

Purpose:

The primitive mklb$f is called to convert a Fortran statement label or integer variable that has been assigned a statement label value, into a PL/I-compatible label value. This value can then be used to cause a full-function nonlocal goto in a Fortran program.

Usage:

```
CALL MKLB$F (STMT, LABEL)
INTEGER*2 STMT
REAL*8 LABEL
```

STMT
: is either a variable to which a statement number has been assigned by an ASSIGN statement, or else is a statement number constant of the form $xxxxx. (Input)

LABEL
: will be set to a PL/I-compatible label value identifying the statement STMT in the activation of the caller of MKLB$F. (Output)

pl1$nl	Generate Nonlocal Goto	pl1$nl
	System Primitive 03-09-79	

Name: pl1$nl

Purpose:

This primitive performs a full-function nonlocal goto to the activation and statement identified by a supplied label value. Label values created by calls to mklb$f are suitable arguments to pl1$nl.

Usage:

```
CALL PL1$NL (LABEL)
REAL*8 LABEL
```

LABEL
is a PL/I-compatible label value (such as is produced by mklb$f). Pl1$nl will cause a nonlocal goto to the statement and activation identified by LABEL. (Input)

8 Data Structure Formats

The sections below describe the data structures associated with the Condition Mechanism. Any user program that uses these structures should examine the version number in the structure (if one is provided); if the format of a structure changes, the version number will be incremented. The user program can then take appropriate action if it is presented with structures of different formats.

8.1 The Condition Frame Header (CFH)

The following declaration shows the format of the Standard Condition Frame Header:

```
dcl    1 cfh based,  /* standard condition frame header */
           2 flags,
             3 backup_inh bit(1),
             3 cond_fr bit(1),
             3 cleanup_done bit(1),
             3 efh_present bit(1),
             3 user_proc bit(1),
             3 mbz bit(9),
             3 fault_fr bit(2),
           2 root,
             3 mbz bit(4),
             3 seg_no bit(12),
           2 ret_pb ptr,
           2 ret_sb ptr,
           2 ret_lb ptr,
           2 ret_keys bit(16) aligned,
           2 after_pcl fixed bin,
           2 hdr_reserved(8) fixed bin,
           2 owner_ptr ptr,
           2 cflags,
             3 crawlout bit(1),
             3 continue_sw bit(1),
             3 return_ok bit(1),
             3 inaction_ok bit(1),
             3 specifier bit(1),
             3 mbz bit(11),
           2 version fixed bin,
           2 cond_name_ptr ptr,
           2 ms_ptr ptr,
           2 info_ptr ptr,
           2 ms_len fixed bin,
           2 info_len fixed bin,
           2 saved_cleanup_pb ptr;
```

flags.backup_inh
will always be '0'b in a condition frame. It is used in

regular call frames to control program counter backup on crawlout from an inner ring.

flags.cond_fr
identifies this frame as a condition frame, and will thus be '1'b.

flags.cleanup_done
is '1'b when this activation has been "cleaned up" by the procedure unwind_, which helps to effect nonlocal goto's. When this flag is set, the value of cfh.ret_pb no longer describes the return point of the activation; that information is available in cfh.saved_cleanup_pb.

flags.efh_present
will always be '0'b in a condition frame. It is used in a regular call frame to indicate that an extended stack frame header containing on-unit data is present.

flags.user_proc
identifies stack frames belonging to "non-support" procedures, and hence will be '0'b in a condition frame.

flags.mbz
is reserved and will be '0'b.

flags.fault_fr
will always be '00'b in a condition frame.

root.mbz
is reserved and must be '0'b.

root.seg_no
is the hardware-defined stack root segment number, and indicates which segment contains the stack root for the stack containing this fault frame.

ret_pb
points to the next instruction to be executed following the call to signl$ that caused this condition to be raised, unless flags.cleanup_done is '1'b, in which case cfh.ret_pb will point to a special code sequence used during stack unwinds, and cfh.saved_cleanup_pb will contain the former value of cfh.ret_pb.

ret_sb
is the hardware-defined stack base of the caller of signl$. Thus, this value also points to the previous stack frame on the stack.

ret_lb
is the hardware-defined linkage base of the caller of signl$.

ret_keys
is the hardware-defined keys register of the caller of signl$.

after_pcl
is the hardware-defined offset of the first argument pointer following the call to signl$ that raised this condition.

hdr_reserved
is reserved for future expansion of the hardware-defined PCL/CALF stack frame header, of which the totality of cfh is a further extension.

owner_ptr
is reserved to point to the ECB of the procedure that owns this stack frame (usually signl$).

cflags.crawlout
is '1'b if this condition occurred in an inner ring (a ring number lower than the ring in which the on-unit is executing), but could not be adequately handled there; else it is '0'b.

cflags.continue_sw
is used to indicate to the Condition Mechanism whether the on-unit that was just invoked (or any of its dynamic descendants) wishes the backward scan of the stack for on-units for this condition to continue upon the on-unit's return. The system primitive cnsig$ is used to request that cflags.continue_sw be turned on; user programs should NOT attempt to set it directly. This switch is cleared before each on-unit is invoked (except ANY$ on-units).

cflags.return_ok
is '1'b if the procedure that raised the condition is willing for control to be returned to it by means of the on-unit simply returning. If '0'b, an attempt by an on-unit for this condition to return will cause the special condition ILLEGAL_ONUNIT_RETURN$ to be signalled. Note, however, that the on-unit may return regardless of the state of cfh.cflags.return_ok if cfh.cflags.continue_sw has previously been set by a call to cnsig$. This is because, in this case, the on-unit return does not cause a return to the procedure that raised the condition, but instead causes a resumption of the stack scan.

cflags.inaction_ok
is '1'b if the procedure that raised the condition has determined that it makes sense for an on-unit for this condition to return without taking any corrective action. If '0'b, the on-unit must take some corrective action before returning, or else continued computation may be undefined. Cflags.inaction_ok will never be '1'b unless cflags.return_ok is '1'b as well. No user program should change the state of this or any other member of cfh.cflags.

cflags.specifier
: if '1'b, indicates that this condition is a PL/I I/O condition (PLIO condition) that requires a specifier pointer as well as a condition name to completely identify it. This specifier is usually a pointer to a PLIO file control block. The specifier must be the first member of the info structure.

cflags.mbz
: is reserved for future expansion and must be '0'b.

version
: identifies the version number (and hence the format) of this structure, and will currently always be 1.

cond_name_ptr
: is a pointer to the name (char(32) varying) of the condition because of which the on-unit is being invoked.

ms_ptr
: is a pointer to a structure which defines the state of the CPU at the time the condition occurred. In the case of hardware faults, ms_ptr will point to a Standard Fault Frame Header (ffh). In the case of software-initiated conditions, ms_ptr will point to a cfh. The two cases can be distinguished by the value of ms_ptr -> cfh.flags.fault_fr: if '00'b, the software case obtains; otherwise, the hardware case obtains.

info_ptr
: is a pointer to an arbitrary structure containing auxiliary information about the condition. If null, no information is available. This pointer is copied directly from the corresponding argument to signl$. If cflags.specifier is '1'b, the format of this structure is partially constrained as described above.

ms_len
: is the length in words of the structure pointed to by ms_ptr.

info_len
: is the length in words of the structure pointed to by info_ptr.

saved_cleanup_pb
: is valid only if flags.cleanup_done is '1'b, and if valid is the former value of cfh.ret_pb (which has been overwritten by the nonlocal goto processor).

Notes

Any procedure attempting to interpret the data contained in a cfh structure should be aware that, in the case of a crawlout, cfh.ms_ptr describes the machine state at the time the condition was generated. The stack history pertaining to that machine state has been lost as a result of the crawlout.

The machine state extant at the time the inner ring was entered is available, and is pointed to by cfh.ret_sb. This machine state will be a cfh or an ffh according to whether the inner ring was entered via a procedure call (cfh) or a fault (ffh). The value of cfh.ret_sb -> cfh.flags.fault_fr can be used to distinguish these cases.

In the case where a crawlout has not occurred, cfh.ms_ptr points to the proper machine state, and no assumptions can be made concerning cfh.ret_sb.

8.2 The Extended Stack Frame Header

Any procedure (or begin block) that desires to make one or more on-units must reserve space in its stack frame header for an extension that contains descriptive information about those on-units. This space is allocated simply by including in such procedure the proper declaration for the system primitive mkonu$.

The format of the stack frame header (with extension) is as shown below.

```
dcl    1 sfh based,  /* stack frame header */
          2 flags,
            3 backup_inh bit(1),
            3 cond_fr bit(1),
            3 cleanup_done bit(1),
            3 efh_present bit(1),
            3 user_proc bit(1),
            3 mbz bit(9),
            3 fault_fr bit(2),
          2 root,
            3 mbz bit(4),
            3 seg_no bit(12),
          2 ret_pb ptr,
          2 ret_sb ptr,
          2 ret_lb ptr,
          2 ret_keys bit(16) aligned,
          2 after_pcl fixed bin,
          2 hdr_reserved(8) fixed bin,
          2 owner_ptr ptr,
          2 tempsc(8) fixed bin,
          2 onunit_ptr ptr,
          2 cleanup_onunit_ptr ptr,
          2 next_efh ptr;
```

flags.backup_inh
: is examined only if this stack frame is the "crawlout frame" on an inner ring stack, and a crawlout is taking place. If '1'b, it indicates that sfh.ret_pb is to be copied to the outer ring as-is, so that the operation being aborted by the crawlout will not be retried. If '0'b, sfh.ret_pb will be set to baserel

(sfh.ret_pb, sfh.after_pcl - 2), so that the inner ring call may be retried.

flags.cond_fr
will be '0'b unless the frame is a condition frame (and is hence described by the structure "cfh").

flags.cleanup_done
is '1'b if the nonlocal goto processor has "cleaned up" this frame by invoking its CLEANUP$ on-unit if any, and resetting its sfh.ret_pb to point to a special code sequence to accomplish the unwinding of this stack frame. When '1'b, the former value of sfh.ret_pb may be found in sfh.tempsc(7:8) provided sfh.flags.efh_present is set.

flags.efh_present
is '1'b if the extension portion of this frame header has been validly initialized. In the present implementation, this implies that at least one call to mkonu$ has been made, since mkonu$ is responsible for performing the initialization. If '0'b, members of this structure below marked (EFH) are not valid and may be used by the procedure for automatic storage.

flags.user_proc
is '1'b if this stack frame belongs to a "non-support" procedure; else is '0'b. If flags.user_proc is '1'b, sfh.owner_ptr is guaranteed to be valid, and to point to an ecb which is followed by the name of the entrypoint.

flags.mbz
is reserved and will be '0'b.

flags.fault_fr
is '00'b if this frame was created by a regular procedure call; or '10' if this frame is a fault frame (ffh) with valid saved registers; or '01'b if this frame is a fault frame (ffh) in which the registers have not yet been saved.

root.mbz
is reserved and must be '0'b.

root.seg_no
is the hardware-defined segment number of the stack root of the stack of which this frame is a member.

ret_pb
points to the next instruction to be executed upon return from this procedure.

ret_sb
contains the stack base belonging to the caller of this procedure, and hence also points to the immediate predecessor of this stack frame.

ret_lb
contains the linkage base belonging to the caller of this procedure.

ret_keys
contains the hardware-defined keys register belonging to the caller of this procedure.

after_pcl
is a value such that baserel (sfh.ret_pb, sfh.after_pcl) points to two words beyond the procedure call (PCL) instruction that invoked this procedure.

hdr_reserved (EFH)
is reserved for future expansion of the hardware-defined PCL stack frame header.

owner_ptr (EFH)
points to the Entry Control Block (ECB) of the procedure that owns this stack frame. This member must be initialized by the called procedure itself, as the PCL instruction does not do it.

tempsc (EFH)
is a fixed-position block of eight words to be used as temporary storage by procedures called by this procedure that have a "shortcall" invocation sequence and hence have no stack frame of their own.

onunit_ptr (EFH)
points to the start of a chain of on-unit descriptor blocks for this activation. If onunit_ptr is null, this activation has no onunit blocks, except possibly for the condition CLEANUP$ as described below.

cleanup_onunit_ptr (EFH)
If nonnull, this activation has an on-unit for the special condition CLEANUP$, and cleanup_onunit_ptr points to the ECB for that on-unit procedure (it does not point to an on-unit descriptor block).

next_efh (EFH)
points to the first on a chain of additional stack frame "header" blocks, so that these do not have to be allocated at the beginning of the stack frame. Presently, next_efh will always be null.

8.3 The Standard Fault Frame Header

Whenever a hardware fault occurs, the so-called Fault Interceptor Module (FIM) is expected to push a stack frame with the standard format shown below. In addition, a register-save protocol must be followed by all but ring three fault interceptors. The only inner ring FIM which

is permitted to violate these rules is the FIM for the Unimplemented Instruction fault.

The standard fault frame header structure is as follows:

```
dcl 1 ffh based,  /* standard fault frame header */
      2 flags,
        3 backup_inh bit(1),
        3 cond_fr bit(1),
        3 cleanup_done bit(1),
        3 efh_present bit(1),
        3 user_proc bit(1),
        3 mbz bit(9),
        3 fault_fr bit(2),
      2 root,
        3 mbz bit(4),
        3 seg_no bit(12),
      2 ret_pb ptr,
      2 ret_sb ptr,
      2 ret_lb ptr,
      2 ret_keys bit(16) aligned,
      2 fault_type fixed bin,
      2 fault_code fixed bin,
      2 fault_addr ptr,
      2 hdr_reserved(7) fixed bin,
      2 regs,
        3 save_mask bit(16) aligned,
        3 fac_1(2) fixed bin(31),
        3 fac_0(2) fixed bin(31),
        3 genr(0:7) fixed bin(31),
        3 xb_reg ptr,
      2 saved_cleanup_pb ptr,
      2 pad fixed bin;
```

flags.backup_inh
will be ignored by the Condition Mechanism for fault frames.

flags.cond_fr
will be '0'b in a fault frame.

flags.cleanup_done
is set to '1'b by the stack unwinder when it has "cleaned up" this fault frame. The old value of ffh.ret_pb has been placed in ffh.saved_cleanup_pb, provided flags.fault_fr is '10'b.

flags.efh_present
will be '0'b in a fault frame, implying that FIM's may not make on-units.

flags.user_proc
will always be '0'b in a fault frame.

flags.mbz
: is reserved and will be '0'b.

flags.fault_fr
: will be '10'b if this frame is indeed a standard format ffh and the registers have been validly saved in ffh.regs; else will be '01'b.

root.seg_no
: is the hardware-define stack root segment number.

ret_pb
: points to the next instruction to be executed following a return from the fault. This will frequently also be the instruction that caused the fault (the case for those faults defined by the CPU reference manual as "backing up" the program counter). If flags.cleanup_done is '1'b, ret_pb will point to a special "unwind" code sequence, and its former value will have been saved if possible in ffh.saved_cleanup_pb.

ret_sb
: contains the value of the SB register at the time of the fault, and hence will usually point to the predecessor of this stack frame.

ret_lb
: contains the value of the LB register at the time of the fault.

ret_keys
: contains the value of the KEYS register at the time of the fault. This can be used to determine in what addressing mode the fault was taken.

fault_type
: is set by each FIM to the offset in the fault table corresponding to the fault that occurred (e.g. a Process Fault results in a fault_type of '04'b3). This datum cannot be guaranteed valid, as it is not set indivisibly with the hardware-defined header information. Since FIM's usually set fault_type just after saving the registers, it is very unlikely for fault_type to be invalid.

fault_code
: is the hardware-defined fault code produced by the fault that was taken.

fault_addr
: is the hardware-defined fault address produced by the fault that was taken.

hdr_reserved
: is reserved for future expansion of the PCL/CALF header.

regs
is valid if flags.fault_fr is '10'b, and if valid contains the saved machine registers at the time of the fault, in the format produced by the RSAV instruction.

saved_cleanup_pb
is valid only if flags.fault_fr is '10'b and flags.cleanup_done is '1'b, and if valid contains the value that was in ret_pb before the latter was overwritten by the stack unwinder.

pad
exists only to make the size of this structure an even number of words.

8.4 The On-Unit Descriptor Block

Each on-unit created by an activation is described to the Condition Mechanism by a descriptor block (except for the special condition CLEANUP$, which has no descriptor). These descriptor blocks are threaded together in a simple linked list, the head of which is pointed to by sfh.onunit_ptr. The format of an on-unit descriptor is as follows.

```
dcl    1 onub based,  /* standard onunit block */
           2 ecb_ptr ptr,
           2 next_ptr ptr,
           2 flags,
             3 not_reverted bit(1),
             3 is_proc bit(1),
             3 specify bit(1),
             3 snap bit(1),
             3 mbz bit(12),
           2 pad fixed bin,
           2 cond_name_ptr ptr,
           2 specifier ptr;
```

ecb_ptr
points to the Entry Control Block (ECB) which represents the procedure or begin block to be invoked when this on-unit is selected for invocation.

next_ptr
points to the next on-unit descriptor on the chain for this activation, or else is null if at the end of the list.

flags.not_reverted
is '1'b if this on-unit is still valid and has not been reverted, and is '0'b if the on-unit has been reverted and is to be ignored by the condition raising mechanism.

flags.is_proc
: is '1'b if this on-unit was made via a call to the primitive mkonu$, and '0'b if it was made via the PL/I <on statement>.

flags.specify
: is '1'b if the condition name does not fully identify which condition this on-unit block is to handle: onub.specifier is a further qualifier in this case.

flags.snap
: is '1'b if the <snap option> was specified in the PL/I <on statement> that created this on-unit; else it is '0'b.

flags.mbz
: is reserved and must be '0'b.

pad
: is reserved and must be 0.

cond_name_ptr
: is a pointer to a varying character string containing the condition name for which this on-unit is a handler. This name may be an incomplete specification if onub.flags.specify is '1'b.

specifier
: is valid only if onub.flags.specify is '1'b, and if valid qualifies the condition name that is pointed to by onub.cond_name_ptr. The primary use of onub.specifier is for PL/I I/O conditions, in which the specification of the condition requires both a name and a file descriptor pointer.

9 System-Defined Conditions

The following table lists all current standard system-defined condition names, lists the meaning of each, and describes what information is available from the cfh structure produced by each condition.

In the descriptions below, "software" means that the machine state frame pointed to by cfh.ms_ptr is a cfh frame; "hardware" means that this frame is an ffh frame. The notation "ffh." and "cfh." below refers to the ffh or cfh that is pointed to by cfh.ms_ptr. The "info structure"s referred to below are pointed to by cfh.info_ptr.

Unless otherwise noted below, the system default on-unit for each condition prints an appropriate diagnostic message on the user's terminal, and calls a new command level.

ERRRTN$

software not returnable

A non-ring-0 call to the ring 0 entry ERRRTN was made, as the result of an ERRRTN SVC or a call to ERRPR$ with certain values of the key.

No info structure is available.

The default on-unit for this condition simulates a call to EXIT; hence, this condition should be signalled only while executing in a Static Mode program.

ACCESS_VIOLATION$

hardware returnable

The process has attempted to perform a CPU instruction which has violated the access control rules of the processor. No information is readily available to differentiate between Write Violation, Read Violation, Execute Vioaltion and Gate Violation.

ffh.fault_type has the value '44'b3.
ffh.fault_addr contains the virtual address the access to which is improper.
ffh.ret_pb points to the instruction causing the violation.

No info structure is available.

POINTER_FAULT$

hardware returnable

The process has referenced through an indirect pointer (IP) whose fault bit is on, but that pointer did not appear to be a valid unsnapped dynamic link.

ffh.fault_type has the value '64'b3.
ffh.fault_addr points to the faulting IP.
ffh.ret_pb points to the faulting instruction.

No info structure is available.

LINKAGE_FAULT$

hardware returnable

The process has referenced through an indirect pointer (IP) which is a valid unsnapped dynamic link, but the desired entry point could not be found in any of the dynamic link tables.

ffh.fault_type has the value '64'b3.

ffh.fault_addr points to the faulting IP.
ffh.ret_pb points to the faulting instruction.

Info structure:

```
dcl 1 info based,
      2 entry_name char(32) var;
```

info.entry_name is the name of the entry point that could not be found.

RO_ERR$

software returnable

A ring-0 call to ERRPR$ or ERRRTN has been made, as the result of some fatal error condition having been detected.

No info structure is available.

The default on-unit for this condition prints no diagnostic, but calls a new command level.

ARITH$

hardware returnable

The process has encountered an Arithmetic Exception Fault.

ffh.fault_type has the value '50'b3.
ffh.fault_code is the hardware-defined Exception Code, and partially identifies the cause of the fault.
ffh.ret_pb points to the next instruction to be executed upon return. There is no way in general to obtain a pointer to the faulting instruction.

No info structure is available.

The Static Mode default on-unit for this condition will simulate Prime 300 fault handling for Arithmetic Exception if the appropriate word of segment '4000 is non-zero (see the System Architecture Guide for the exact location). If a Static Mode program is not in execution when the fault occurs, or if the Prime 300 vector word is zero, the standard default handler for this condition will resignal the ERROR condition with the appropriate info structure.

SVC_INST$

hardware returnable

The process has executed an SVC instruction, but the system has not been able to perform the operation. If the user is in "SVC virtual" mode, all SVC instructions result in this condition being

raised.

ffh.fault_type has the value '14'b3.
ffh.ret_pb points to the location following the SVC instruction.

Info structure:
```
dcl 1 info based,
      2 reason fixed bin;
```

info.reason has one of the following values: 1 (bad SVC operation code or bad argument(s)); 2 (alternate return needed but was zero); or 3 (virtual SVC handling is in effect in this process).

For the case of virtual SVC's only (info.reason code of 3), the Static Mode default on-unit will simulate the Prime 300 fault handling for the SVC fault if the appropriate word of segment '4000 is non-zero; if this word (see the System Architecture Guide for the exact location) is zero or if there is no Static Mode program in execution, the standard default handler prints a diagnostic and calls a new command level.

ILLEGAL_ONUNIT_RETURN$

software not returnable

An on-unit for some condition has attempted to return, when that has been disallowed by the procedure that raised the condition.

Info structure: the standard-format cfh that describes the condition whose on-unit has illegally attempted to return.

STACK_OVF$

hardware returnable

The process has overflowed one of its stack segments, but the Condition Mechanism was able to locate a stack on which to raise this condition.

ffh.fault_type has the value '54'b3.
segno (ffh.fault_addr) is the last stack segment in the chain of stack segments of the stack that overflowed. It is this segment that contains the zero extension pointer that caused the stack overflow fault.
ffh.ret_pb points to the faulting instruction.

No info structure is available.

The Static Mode default on-unit will attempt to simulate the Prime 300 fault handling for Stack Overflow fault if the appropriate word (see the System Architecture Guide) of segment '4000 is nonzero. If this word is zero or if no Static Mode program is in execution, the standard default handling occurs.

QUIT$

hardware|software returnable

The user has actuated the quit button (Break Key or Control-P) on his terminal.

If this is a hardware signal, then ffh.fault_type has the value '04'b3.
cfh.ret_pb or ffh.ret_pb points to the next instruction to be executed in the faulting procedure.

No info structure is available.

The default on-unit flushes the input and output buffers of the user's terminal, prints the message "QUIT." on the terminal, and calls a new command level.

UII$

hardware returnable

The process has executed an unrecognized instruction that nevertheless caused an Unimplemented Instruction Fault, or else the system UII handler detected an error in processing the valid UII.

The ffh that accompanies this condition is nonstandard in that ffh.regs is not valid.

ffh.ret_pb points to the next instruction to be executed in the faulting procedure.

CLEANUP$

software returnable

The nonlocal goto processor (unwind_) is in the process of invoking on-units for the condition CLEANUP$ in each activation on the stack, prior to actually unwinding the stack. The on-unit for this condition should return, unless it encounters a fatal error. Calls to cnsig$ from a CLEANUP$ on-unit have no effect.

No info structure is available.

LISTENER_ORDER$

software (varies)

This condition is used internally by the command loop to manage its recursion. Users should NEVER make on-units for this condition, and user default on-units (ANY$) should always pass this condition on by returning.

The format of the info structure that accompanies this condition is described in the writeup on the Primos Command Environment.

BAD_NONLOCAL_GOTO$

software not returnable

The nonlocal goto processor has been asked to transfer control to a label whose display (stack) pointer is invalid, or whose target activation has already been cleaned up. There is also a possibility that the user's stack may have been overwritten.

Info structure:

```
dcl 1 info based,
      2 target_label label,
      2 ptr_to_nlg_call ptr,
      2 caller_sb ptr;
```

info.target_label is the label to which the nonlocal goto was attempted. info.ptr_to_nlg_call is a pointer to the call to pl1$nl that requested this nonlocal goto. info.caller_sb is a pointer to the activation (stack frame) requesting this nonlocal goto.

NONLOCAL_GOTO$

software returnable

This condition is signalled by the PL/I nonlocal goto processor pl1$nl just prior to setting up the stack unwind (and hence prior to the invocation of any CLEANUP$ on-units). This condition exists to enable certain overseer software (such as the debugger) to be informed that the nonlocal goto is occurring. The default handler for this condition simply returns. When a procedure handling this condition wishes to let the nonlocal goto occur, it should simply return (without continue-to-signal set).

Info structure: same as for the BAD_NONLOCAL_GOTO$ condition.

ANY$

(pseudo-condition)

An activation's on-unit for ANY$ is invoked if that activation does not have a specific on-unit for the condition that was raised. The condition frame header for the condition ANY$ will describe the original condition directly; there is no separate condition frame header for the condition ANY$ unless ANY$ has been explicitly raised by a call to signl$ (not a recommended practice).

ILLEGAL_SEGNO$

hardware returnable

The process has referenced a virtual address whose segment number is out of bounds.

ffh.fault_type has the value '60'b3.
ffh.ret_pb points to the faulting instruction.
ffh.fault_addr is the virtual address that is in error.

No info structure is available.

NO_AVAIL_SEGS$

hardware returnable

The process has referenced a virtual address that refers to a segment that has not yet been created. At the moment, the system has no free page tables to assign to the segment. If the on-unit for this condition returns, the reference will be retried, with some possibility of success if this or some other process has in the meantime deleted a segment.

ffh.fault_type has the value '60'b3.
ffh.ret_pb points to the faulting instruction.
ffh.fault_addr is the virtual address that is causing the attempted segment creation.

No info structure is available.

NULL_POINTER$

hardware returnable

The process has referenced through an IP or base register whose segment number is '7777'b3. This is considered to be a reference through a null pointer, although user software should always employ the single value 7777/0 for the null pointer.

ffh.fault_type has the value '60'b3.
ffh.ret_pb points to the faulting instruction.
ffh.fault_addr contains the null pointer through which a reference was made.

No info structure is available.

The default on-unit for this condition resignals the ERROR condition with the appropriate info structure.

UNDEFINED_GATE$

software not returnable

The process has called an inner ring gate segment at an address within the initialized portion of the gate segment, but there is no legal gate at that address. This results from the fact that gate segments must be padded to the next page boundary with "illegal" gate entries.

No info structure is available.

ILLEGAL_INST$

hardware returnable

The process has attempted to execute an illegal instruction.

ffh.fault_type has the value '40'b3.
ffh.ret_pb points at the faulting instruction.

No info structure is available.

RESTRICTED_INST$

hardware returnable

The process has attempted to execute an instruction whose use is restricted to ring 0 procedures. Certain of these instructions (in the I/O class) can be simulated by ring 0. An instruction which causes this condition to be raised could not be simulated by this mechanism.

ffh.fault_type has the value '00'b3.
ffh.ret_pb points to the faulting instruction.

OUT_OF_BOUNDS$

hardware returnable

The process has referenced a page of some segment that has been defined as not referencible in any ring (i.e. no main memory or backing storage is allocated for that page, and allocation is not permitted).

ffh.fault_type has the value '10'b3.
ffh.ret_pb points at the faulting instruction.
ffh.fault_addr contains the offending virtual address.

No info structure is available.

PAGE_FAULT_ERR$

hardware returnable

The process has encountered a page fault referencing a valid virtual address, but due to a disk error, the page control mechanism has not been able to load the page into main memory. If the on-unit for this condition returns, the reference will be retried, and there is some likelihood that the disk read will succeed and the reference thus be completed.

ffh.fault_type has the value '10'b3.
ffh.ret_pb points at the faulting instruction.
ffh.fault_addr contains the virtual address the page for which cannot be retrieved.

No info structure is available.

EXIT$

software returnable

The process has made a call to the EXIT primitive, via a direct call or an EXIT SVC. This condition should not be handled by user programs, since it is used by certain Primos software to monitor the execution of Static Mode programs.

No info structure is available.

The default on-unit for this condition simply returns.

STOP$

software not returnable

The process has executed a <stop statement> in a higher-level-language program. This condition should not be handled by user programs, as it is used by Prime software to ensure the proper operation of the <stop statement> in the various languages.

No info structure is available.

The default on-unit for this condition performs a nonlocal goto back to the command processor which invoked the procedure which (or one of the dynamic descendants of which) executed the <stop statement>.

PAUSE$

software returnable

The process has executed a <pause statement> in a Fortran program. This condition should not be handled by user programs since it is used by Prime software to ensure the proper operation of the Fortran <pause statement>.

No info structure is available.

The default on-unit for this condition prints no diagnostic, but calls a new command level.

REENTER$

software returnable

This condition is raised by the Primos REN (REENTER) command, and is used to reenter a subsystem that has been temporarily suspended due to another condition (such as a quit signal). The details of operation are left to the individual subsystems.

No info structure is available.

The default on-unit for this condition simply returns. The REN command will print a diagnostic if control returns to it after it signals REENTER$, since this implies that there was no subsystem on the stack willing to accept reentry.

BAD_PASSWORD$

software not returnable

This condition is raised by the ATCH$$ primitive when attempting to attach to a passworded directory with an incorrect password. This condition is signalled non-returnably in order to increase the work function of machine-aided password penetration.

No info structure is available.

ENDFILE (file)

software (PL-I) returnable

This condition is raised when an end-of-file is encountered while reading a PL/I file with PL/I I/O statements. The value of the onfile() builtin function identifies the file involved.

The standard PL/I condition info structure is provided (see below). The value of info.oncode_value is undefined, and info.file_ptr identifies the file on which end-of-file occurred.

The default on-unit for this condition prints a diagnostic and then resignals the ERROR condition with an info.oncode_value of 1044.

ENDPAGE (file)

software (PL-I) returnable

This condition is raised when end-of-page is encountered while writing a PL/I file using PL/I I/O statements. The value of the onfile() builtin function identifies the file on which the end-of-page was encountered.

The standard PL/I condition info structure is provided (see below). The value of info.oncode_value is undefined; info.file_ptr identifies the file in question.

The default on-unit for this condition performs a "put skip" on the file, and then returns.

KEY (file)

software (PL-I) returnable

The KEY condition is raised when reading or writing a keyed PL/I file with PL/I I/O statements, and the supplied key does not exist (read) or already exists (write). The value of the onfile() builtin function identifies the file in question; the value of the onkey() builtin function contains the key in error.

The standard PL/I condition info structure is supplied. The value of info.oncode_value is undefined; the value of info.file_ptr identifies the file in question.

The default on-unit prints a diagnostic and resignals the ERROR condition, with an info.oncode_value of 1045.

ERROR

software (PL-I) (varies)

This condition is a catch-all error condition defined in PL/I. The default on-unit for most PL/I-defined conditions (such as KEY) result in the ERROR condition being resignalled. Hence, the programmer has the choice of handling a more- or less-specific case of the condition.

Many I/O and conversion operations in PL/I can result in the raising of the ERROR condition. The standard PL/I condition info structure is supplied. Each distinct error has been assigned a unique value of info.oncode_value. The info structure is:

```
dcl 1 info based,
      2 file_ptr ptr,
```

```
        2 info_struct_len fixed bin,
        2 oncode_value fixed bin,
        2 ret_addr ptr;
```

info.file_ptr
: when valid, is a pointer to a PL/I file control block that identifies the PL/I file in question in the error at hand.

info.info_struct_len
: is the length, in words, of this structure. It identifies both the version of the structure and the extent to which any optional items at the end of the structure are filled in. The present value is 6.

info.oncode_value
: is the unique code assigned to this particular error case. This integer is also the value of the oncode() builtin function when the latter is used in an on-unit invoked in response to this error.

info.ret_addr
: is a pointer to the instruction (statement) in the user's program whose execution has caused this error.

The default on-unit for this condition prints a diagnostic corresponding to the value of info,oncode_value, and calls a new command level, unless the error is one of the arithmetic errors that is handled "without comment" (such as underflow), in which case the appropriate action is taken, and the on-unit returns control to the point of interruption.

The specific values of info.oncode_value, and their corresponding messages, are documented separately.

10 The Crawlout Mechanism

An event known as a crawlout occurs whenever the Condition Mechanism reaches the end of an inner ring stack (a ring other than 3) without finding a selectable on-unit for the condition that has been raised. Note that a crawlout can occur even when the inner ring has an on-unit for the condition, if that on-unit signals another condition, or if the on-unit calls cnsig$ and returns, causing a resumption of the stack scan.

The following terminology is used to describe the action of the crawlout mechanism. The target ring is that ring from which the present (inner) ring was entered. The condition frame is that (inner ring) condition frame that describes the condition as it was raised in the inner ring. The crawlout frame is the topmost frame on the inner ring stack; it is the crawlout frame whose return-conditions describe the point in the outer ring from which this inner ring was entered. Thus, we have target ring = ring (crawlout_frame -> sfh.ret_pb).

The first step in performing a crawlout is to determine how large a stack frame must be allocated on the target ring stack. This size is just the sum of the size of the crawlout frame header, the machine state header from the inner ring, the condition name, the info structure if any, and a special auxiliary pointer area. The target ring stack root segment number is obtained from a static table compiled into crawl_.

The target ring stack frame has the following format:

```
dcl 1 orf based, /* outer ring frame */
      2 crawlout_frame(ffh_size) fixed bin,
      2 cofim,  /* special data for crawlout FIM */
        3 cond_name_ptr ptr,
        3 ms_ptr ptr,
        3 info_ptr ptr,
        3 ms_len fixed bin,
        3 info_len fixed bin,
        3 action,
          4 return_ok bit(1) unal,
          4 inaction_ok bit(1) unal,
          4 crawlout bit(1) unal,
          4 specifier bit(1) unal,
          4 mbz bit(12) unal,
      2 cond_name char(32) var,
      2 ms_frame(orf.ms_len) fixed bin,  /* machine state */
      2 info_struc(orf.info_len) fixed bin;
```

Since ffh_size (the size of a standard ffh) is a constant, orf.cofim is at a known offset in orf. Hence, a crawlout FIM module may find it without requiring that it be passed as an argument. The pointers and lengths in orf.cofim identify the location and size of the other members of orf, and are used by a crawlout FIM in order to re-invoke signl$ in the outer ring.

If sufficient space for orf cannot be found in the target ring stack, a fatal error ("crawlout stack overflow") occurs.

The remaining steps in the crawlout process are as follows (quits are inhibited until the start of step (11)):

(1) The crawlout frame is copied to orf.crawlout_frame. If the crawlout frame is an ffh that does not contain valid saved registers, then crawl_ uses the "earliest valid registers" pointer passed to it to locate and copy those saved registers into orf.crawlout_frame.

(2) The condition name is copied to orf.cond_name. Space for a 32 character name is always allocated.

(3) The inner ring machine state (pointed to by cfh.ms_ptr) is copied to orf.ms_frame. Because this data is always simply copied

through, a multiring crawlout will carry the machine state at the time the condition was originally raised.

(4) The info structure (if any) is copied to orf.info_struc.

(5) orf.action is set by copying return_ok, inaction_ok and specifier from the inner ring cfh.cflags; orf.action.crawlout is set to '1'b. orf.action will be passed to signl$ by the crawlout FIM.

(6) Unless the crawlout frame's cfh.flags.backup_inh is '1'b or the crawlout frame is an ffh, the ret_pb in orf.crawl_frame is backed up to point at the PCL that entered the inner ring. In this way, if signl$ returns, the call into the inner ring is re-executed unless prohibited by the inner ring.

(7) The pointers and lengths in orf.cofim are set to identify the appropriate members of the orf structure.

(8) The inner ring crawlout frame is then modified to make it appear as though the inner ring was called by some procedure (the entry value passed to crawl_) whose stack frame is orf.

(9) An unwind to this "activation" of step (8) is set up by a call to unwind_.

(10) If we are crawling out from ring zero, unlkf$ is called to unlock all ring zero locks.

(11) Then crawl_ returns, causing the unwind to take place. Control reappears in the target ring, with orf as the current stack frame, and the procedure of step (8) in control.

(12) If the procedure of step (8) is the standard crawlout FIM crfim_, it will simply call signl$ with the values in orf.cofim as arguments. This raises the same condition in the target ring. If signl$ returns, crfim_ reloads the registers if needed and simply returns.

11 Internal Interfaces

WARNING: These internal interfaces are documented for completeness and ease of understanding only; the interfaces are subject to change in calling sequence, functionality, and existence.

11.1 CRAWL_

crawl_ Perform a Crawlout from an Inner Ring

```
dcl crawl_ entry (entry, ptr, ptr);

call crawl_ (crawl_fim, crawl_frame, regs_frame);
```

crawl_fim
is an entry value representing the procedure to which control is to be passed when the crawlout is complete. This procedure must satisfy certain restrictions described in Notes below. The entry value must represent an external entry since no display pointer will be passed. (Input)

crawl_frame
is a pointer to the topmost frame on the inner ring stack, to be used as the "crawlout frame" as defined above. (Input)

regs_frame
is a pointer to the earliest frame on the current ring stack that contains valid saved registers. The value of regs_frame will not be used if the crawlout frame is not an ffh. Usually, the value of crawl_frame and regs_frame will be identical; however, the raise_ internal interface returns the proper value to pass as regs_frame. (Input)

The crawlout operation is performed as described above. Control is transferred to the procedure described by crawl_fim, in the environment of the target ring.

Remember that the caller of crawl_ must have a stack frame that is a condition frame; that is, the procedure calling crawl_ cannot expect to be able to overlay the cfh structure with its own automatic storage. Mechanically, this is accomplished by including in the caller of crawl_ a declaration of the form "dcl dummy_ entry () options (shortcall (cfh_size))" as explained in the insert file for the cfh structure.

Restrictions on Crawlout FIM's

A crawlout FIM will not be given control via a regular hardware PCL instruction. Therefore, it must not matter which stack root is used; the procedure must be able to be entered in 64V addressing mode; the procedure must expect no arguments; and the procedure must not use any automatic storage except the words comprising orf.cofim, unless it first extends its stack frame. These restrictions will probably make it impossible to write a crawlout FIM in PL/I.

11.2 CSTAK$

cstak$ Manipulate Concealed Stack (Ring 0)

```
dcl cstak$ entry (fixed bin, 1, 2 ptr, 2 bit(16) aligned,
                  2 fixed bin, 2 ptr,
                  bit(1) aligned, ptr);

call cstak$ (depth, cs_data, eog, pb_value);
```

depth
is the offset in concealed stack (CS) frames from the most recent entry of the CS, of the CS frame to read. The value of depth must be between 0 and (maximum CS depth - 1), inclusive, lest a fatal process error result. (Input)

cs_data
is a copy of the CS frame specified by depth. The ret_pb in cs_data has the modified value if modification was requested. (Output)

eog
is '1'b if the CS entry read represents an "end of group" (EOG) frame according to the definition of a CS group; else it is '0'b. (Output)

pb_value
if null, is ignored and no modification of the CS occurs; else, the ret_pb in the CS entry just read is modified to have the value pb_value. (Input)

See the section, "Stack Unwind Protocol", for a detailed discussion of the concealed stack.

11.3 FATAL_

fatal_ . Generate Fatal Process Error

```
dcl fatal_ entry (fixed bin) options (shortcall (4));

call fatal_ (error_code);
```

error_code
is a standard system error code, the corresponding message for which will be printed on the user's terminal by ring zero as the process' ring three environment is re-initialized. (Input)

Fatal_ is used to insulate the other interfaces of the condition mechanism from detailed knowledge of how to generate a fatal process error. The present fatal_ calls the ring zero gate fatal$ (which is at a known location in the ring zero gate segment, and so requires no linkage fault).

11.4 FNONU$

fnonu$ Find On-Unit in Given Frame

```
dcl fnonu$ entry (ptr, char(*) var, ptr, ptr, ptr)
           returns (bit(1) aligned);
```

```
cond_found = fnonu$ (frame_ptr, condition_name,
      onunit_or_last_ptr, catch_all_ptr, spec_ptr);
```

frame_ptr
: is a pointer to the stack frame of the activation whose list of on-units is to be searched. A stack frame that has no extended header will be considered to have an empty on-unit list. (Input)

condition_name
: is the name of the condition whose on-unit is desired. This name may be qualified by spec_ptr, as explained below. (Input)

onunit_or_last_ptr
: is a pointer to the specific on-unit descriptor block (if cond_found is '1'b), or else is a pointer to the last on-unit descriptor block on that activation's list, making it easy to thread a new on-unit descriptor block onto the list. If the activation's on-unit list is empty, onunit_or_last_ptr will be set such that using it to set a next-pointer will still work (i.e. the stack frame header is considered to contain a "phantom" first descriptor on the on-unit list). Onunit_or_last_ptr is null if there is no extended stack frame header present, or if the activation is "dead" by virtue of its sfh.flags.cleanup_done being '1'b. (Output)

catch_all_ptr
: if valid, is a pointer to the activations's ANY$ on-unit descriptor block. Catch_all_ptr is null if no ANY$ on-unit exists in this activation. Catch_all_ptr is invalid if cond_found is '1'b, since a specific on-unit has been found. (Output)

spec_ptr
: if non-null, this pointer qualifies condition_name, and is used to search for on-units for PL/I builtin conditions that have file control blocks associated with them. If spec_ptr is null, it has no effect; otherwise, only an on-unit whose name matches condition_name, and whose specifier pointer is valid and equal to spec_ptr, will be reported as found. (Input)

cond_found
: is '1'b if a specific on-unit for condition_name was found in the activation, and '0'b otherwise. (Output)

Note: fnonu$ cannot be used to find an activation's on-unit for the special condition CLEANUP$. The item sfh.cleanup_onunit_ptr points directly to the ECB for this on-unit, if any.

11.5 PREVSB_

prevsb_ Get Previous Stack Frame

```
dcl prevsb_ entry (ptr, bit(1) aligned, bit(1) aligned, fixed bin)
      returns (ptr);

previous_frame = prevsb_ (current_frame, crawl_flag, fix_flag,
                          cs_depth);
```

current_frame
: is a pointer to the stack frame whose immediate predecessor is desired. (Input)

crawl_flag
: is '1'b if the end of the stack has been reached and this is an inner ring stack (stack in a ring less than three); else it is '0'b. (Output)

fix_flag
: is '1'b if prevsb_ is to repair the stack discontinuity that occurs when the Primos Command Loop invokes a so-called Static Mode program, and is '0'b if prevsb_ is merely to "hide" this discontinuity instead of actually fixing it. This argument should have the value '1'b only when called from Primos Command Loop procedures that explicitly know when to perform this function, and from the procedure unwind_. (Input)

cs_depth
: is the frame offset of the next frame on the Concealed Stack to consider. The first call to prevsb_ should pass cs_depth equal to 0. Thereafter, prevsb_ will update cs_depth as appropriate. See the section, "Stack Unwind Protocol" for a description of how the Concealed Stack is used. (Input/Output)

previous_frame
: is a pointer to the stack frame that is the immediate predecessor of current_frame, except when the end of the stack has been reached, in which case null is returned. (Output)

Note that prevsb_ has explicitly been given knowledge that enables it to make it appear that a Static Mode program's stack is part of the ring three stack (even though, physically, this may not be so).

11.6 RAISE_

raise_ Raise a Specific Condition

```
dcl raise_ entry (bit(1) aligned, ptr, ptr);

call raise_ (crawlout_needed, crawl_frame, regs_frame);
```

crawlout_needed
: is '1'b if the scan of the stack for an on-unit for the condition has reached the end of the stack, indicating the need for a crawlout. If '0'b, indicates that the on-unit was invoked and has now returned. Procedure raise_ does not check cfh.cflags.return_ok; this must be done by the caller of raise_. (Output)

crawl_frame
: points to the topmost frame on the stack of the current ring, when crawlout_needed is '1'b. This value is suitable to be passed to the crawl_ procedure to perform the crawlout. (Output)

regs_frame
: is a pointer to the earliest frame on the current ring stack that contains valid saved registers. This value is suitable to be passed to the entry crawl_ to effect a crawlout. (Output)

The stack frame of the caller of raise_ must be a condition frame and must describe the condition to be signalled. Raise_ does not check to be sure that its caller's frame meets these requirements. Raise_ will, however, process cfh.cflags.continue_sw, so that a single call to raise_ is sufficient to invoke all on-units for the condition in that ring that are logically required to be invoked.

11.7 UNWIND_

unwind_ — Set Up Stack Unwind for Nonlocal Goto

```
dcl unwind_ entry (label) returns (bit(1) aligned);

unwind_ok = unwind_ (target_of_nl_goto);
```

target_of_nl_goto
: is the label variable representing the statement and the activation to which it is desired to transfer control. Note that this mechanism cannot be used to transfer control into an inner ring from an outer ring, although it can be used to perform a crawlout to the most recent activation of the ring that called the current ring. (Input)

unwind_ok
: is '1'b if the unwind has been properly set up, and '0'b if the label target_of_nl_goto has an invalid activation (stack) pointer or if the stack in the current ring has been damaged, or if the target activation has already been cleaned up. (Output)

Unwind_ performs only the manipulation of the stack necessary to precipitate the unwind operation, and then returns to its caller. In addition, however, unwind_ will invoke the CLEANUP$ on-unit, if any, in every activation being aborted by the nonlocal goto. After each CLEANUP$ on-unit (if any) returns, the activation's sfh.flags.cleanup_done is set to '1'b.

When the caller of unwind_ itself returns, the stack will be unwound and control will arrive at the point specified by target_of_nl_goto.

12 Stack Unwind Protocol

Whenever it is necessary to unwind a stack (either partially or completely), it must be realized that the Concealed Stack may actually be a part of the process' stack history even though it is not physically threaded into the stack. This occurs when a CALF instruction encounters a fault (such as page fault, segment fault, process abort, and so on) before the fault frame on the real stack has been constructed. It is in fact possible for there to be multiple such instances (e.g. the CALF for pointer fault encounters a segment fault, whose CALF in turn encounters a page fault).

When unwinding the regular stack, therefore, the Concealed Stack must be unwound as well, if that is appropriate. The following algorithm is used when contemplating a stack frame, to decide if that frame resulted from such a faulted CALF.

First, we define a Concealed Stack Group (CSG) as the one or more CS frames that intervene between two frames on the real stack. It can be seen that all frames in a CSG are in a sense "caused" by the original CALF instruction that did not complete. Therefore, when unwinding, the next CSG must be unwound as a group, since there are no other intervening real stack frames between members of the CSG.

The end of a CSG (called the "end of group" or EOG frame) has been reached when that frame's ret_pb points to a higher ring, or does not point to a CALF instruction.

The unwinding algorithm is as follows:

(1) Consider the next frame (working backward in time) on the stack, starting with the caller of unwind_. If this is the son of the target frame, invoke its CLEANUP$ on-unit if any, and goto step (5).

(2) If this stack frame is not a fault frame, there can be no CSG between it and the previous real stack frame, so clean this frame up by invoking its CLEANUP$ on-unit if any, and setting its sfh.ret_pb to point at a PRTN instruction. Goto (1).

(3) If ffh.ret_pb does not point at a CALF instruction, then clean up

the frame as in step (2), and goto step (1).

(4) Else set ffh.ret_pb to point to the CALF_ entry point, which will simply remove the CSG from the CS and then execute a PRTN. Goto step (1).

(5) [This is the son of the Target Frame] If this frame is not a fault frame, then set sfh.ret_pb to point at the target label. Exit.

(6) If ffh.ret_pb does not point at a CALF instruction, or if ring (ffh.ret_pb) is greater than the current ring, then reset sfh.ret_pb as in step (5), and exit. (No inner ring CS entries exist).

(7) Else if this is a crawlout, modify the son of the EOG frame in this CSG so that its ret_pb points to the target label. Then point this ffh.ret_pb at the entry point CALFT_, which will remove all but the EOG frame from the CS and transfer to the target label (which is now contained in the son of the EOG frame). Exit.

(8) Else this is not a crawlout. Perform step (7), except modify the ret_pb of the EOG frame itself to point at the target label. This will result in the entire CSG being removed from the CS. Exit.

Note that there is currently no way for software other than the unwinder to automatically trace the stack history through the concealed stack.

13 A PL/I Example

The hypothetical problem: provide a program with an on-unit for the condition POINTER$_FAULT that will fix the faulting pointer to point at a (possibly long-integer) zero, and retry the instruction that faulted.

Solution:

```
problem: proc;

dcl mkonu$ entry (char(*) var, entry) options (shortcall (18)),
    long_zero fixed bin(31) static init (0),
    ptr_fault_ char(14) var static init ('POINTER_FAULT$');

$INSERT dcl_for_ffh
$INSERT dcl_for_cfh

/* Set up the on-unit for POINTER_FAULT$. */

        call mkonu$ (ptr_fault_, ptr_handler);

/* Now perform whatever computations might pointer-fault. */
```

```
/* Having done them, return. */

        return;

/* On-unit for POINTER_FAULT$.  Correct the faulting pointer to
   point at long_zero, and restart at the point of interruption.  */

ptr_handler: proc (cp);

dcl cp ptr;  /* pointer to cfh */

dcl msp ptr;  /* local copy of machine state ptr */
dcl based_ptr ptr based;

        msp = cp -> cfh.ms_ptr;
        msp -> ffh.fault_addr -> based_ptr = addr (long_zero);

/* The above uses the hardware-saved pointer to the faulting pointer,
   which is found in the machine-state ffh, to reset the bad pointer.
   We then simply return, causing the instruction to be re-executed.  *
/

        return;

        end;  /* ptr_handler */

    end;  /* problem */
```

14 A Fortran Example

The hypothetical problem: provide a subroutine A with a handler for the QUIT$ condition, which will set a particular common variable QUITX to 1 and return to the point of interruption. Presumably, at some later time, A or some other routine in the subsystem would interrogate QUITX to see if a quit happened.

Solution:

```
      SUBROUTINE A
C
      EXTERNAL B   /* ON-UNIT FOR QUIT$
      STACK HEADER 34
      COMMON /COM/ AA, BB, QUITX, CC, DD
      INTEGER*2 AA, BB, QUITX, CC, DD
C
C.. SET UP THE ON-UNIT.
C
      CALL MKON$F ('QUIT$', 5, B)
C
C.. COMPUTE UNDER PROTECTION OF B.
C
C.............
```

```
C
C.. INTERROGATE QUITX TO SEE IF WE GOT A QUIT.
C
      IF (QUITX .NE. 1) GOTO 1000   /* NO QUIT
C
C.. A QUIT OCCURRED, DO SOMETHING ABOUT IT.
C
C..............
C
1000  CONTINUE
      RETURN
      END

      SUBROUTINE B (CP)   /* ON-UNIT FOR QUIT$
C
      INTEGER*4 CP   /* PTR TO CONDITION FRAME
C
      COMMON /COM/ AA, BB, QUITX, CC, DD
      INTEGER*2 AA, BB, QUITX, CC, DD
C
      QUITX = 1   /* SET QUIT-SEEN FLAG
      RETURN   /* AND RETURN TO POINT OF INTERRUPT
      END
```