To: Distribution
From: M.G. Smith, R.A. Freiburghouse, M.B. Weaver
Date: March 25, 1973
Subject: A New Multics Signal Mechanism

The current Multics signal mechanism causes several problems for both novice and experienced users. It does not fully support the needs of PL/I and is not well suited to the Multics ring structure. We felt that the number of problems in this area was so large that a complete review of the signal mechanism was necessary. We have completed that review and have identified a number of problems which we propose to solve by a redesigned signal mechanism that is conceptually cleaner and more general than the existing mechanism. We feel that these changes are necessary if we are to provide a suitable standard product for Multics users. We plan to implement these changes during the next few months and install them on the 6180. The purpose of this MSB is to inform you of our plans and to solicit your comments and opinions.

1. PROBLEM:
The deceptively benign appearance of the "ready" message following a QUIT or a message from the default error handler has tricked some users into believing they were talking to a "clean" process, when in fact the suspended stack above the new listener involved enabled condition handlers, initiated segments, yet-to-be-invoked cleanup handlers, etc.

SOLUTION:
The default error handler and QUIT responder will not call the listener directly, but will ask a question: "Do you wish to hold, release, or start?" The question will be asked by the normal "command_query_" method. Only if the answer is "hold" will a new listener be invoked.

2. PROBLEM:
Subsystem writers and other users who desire to stack a new command loop beneath themselves have been unable to call "cu_$cl" because that routine performs a number of unrelated and annoying functions besides reading commands, to wit: performing a reset-read (which kills absentee jobs or loses type-ahead on interactive), laying down an impermeable condition wall, setting an automatic release switch.

SOLUTION:
Move the unrelated functions out of "cu_$cl". The automatic release is discarded, as the question asked by the new default handler will obviate it. The reset-read is per-

formed if necessary when the question is asked by the default handler. The wall is simply discarded--any user wanting one can easily put one in, but one put in by the system is very difficult to remove. See 4. below.

3. PROBLEM:
Some subsystems desire more rapid handling of certain conditions than is provided by the current mechanism, to wit: MMEs in subsystems which use them as supervisor calls, MMEs for debug breaks, traps-before-link and traps-at-first-reference.

SOLUTION:
Since no replacement for "signal_" will provide the performance speed-up these systems desire unless it is tailored to their own requirements, we will provide documentation on how to replace "signal_" but we will not presume to code the replacement.

4. PROBLEM:
Currently some conditions are set up and handled differently than others for no essential reason--the so-called "special handlers". This introduces unneeded complexity into the condition mechanism and also forces programmers to go outside the PL/I language to use the features.

SOLUTION:
The "cleanup" condition will be enabled as any other condition. The "any" condition will be enabled as any other condition, and its meaning will be to catch any condition not specifically named in a separate on-unit in the current frame-i.e., it becomes the default handler or "wall".

5. PROBLEM:
The relationship between a PL/I program and a ring should be respected by the signal mechanism so that users can construct multi-ring PL/I subsystems. The current implementation resignals all conditions originating in the lower ring in a higher ring without invoking the default handler for the condition in the lower ring. This is inappropriate for all of those conditions for which the default handler provides a result suitable for use by the interrupted program. For example, the default handler for the conditions underflow, stringsize and endpage provides a result and continues execution.

SOLUTION:
When the signaller reaches the end of the stack in a given ring, it will invoke the default error handler. If the condition is nonfatal (one of the above), the default error handler will set the result and return to the interrupted program; otherwise, it will effectively either resignal in the next outer ring or will write a message on error-output, do a reset-read, and ask the user if he wishes to hold, start or release. The decision to resignal a fatal error in an outer ring and the choice of which ring is described as part of the solution to the next problem.

6. PROBLEM:
In the present implementation of the signaling mechanism, there is a feature which prohibits effective utiliation of the ring mechanism for certain supervisory, monitoring, metering, and debugging tasks for which it would otherwise be ideally suited. To understand it, envision a lower-ring program (call it the "monitor" for ease of reference) which desires to encapsulate a higher-ring program (call it the "application") so that it can meter it or debug it. The point of using the ring mechanism to do this is to ensure that the act of metering or debugging the application will not in fact change it (putting metering or debugging calls in it moves code around, can affect where page boundaries fall, introduce extra frames into the stack, etc.) and to ensure the integrity of the monitor even if the application program suffers catastrophic failure. This usage of rings is fully in conformity with the Multics philosophy of protection rings: the encapsulated application is intended to be at the mercy of the monitor and to know not what the monitor is up to; the monitor is intended to be protected from any interference by the possibly dangerous application program.

However, the present treatment of reflecting non-supervisor faults directly back to the calling ring is a violation of the Multics ring philosophy, and makes the monitor susceptible to damage from the application. To wit, consider the occurrence of a non-supervisor fault in the application (for the purposes of this discussion, a supervisor fault is one which is handled successfully and transparently by the hardcore supervisor, so that for all practical purposes it did not occur at all--examples are page faults, segment faults, bounds faults). This non-supervisor fault causes a trap to ring zero, ring zero decides it knows nothing of the fault, and so it re-signals the fault as a condition in the faulting ring. Note that this

communication from the higher-ring application program to the hardcore and back out to the higher ring again takes place without the consent or knowledge of the intermediate-ring monitor. The bypassing of the monitor ring in this exchange is the violation of the ring philosophy: it allows the application program to cause fatal process errors and prohibits the monitor from setting breakpoints and taking control away from the upper-ring upon the break.

Hence, it is proposed that, instead of signaling a fault in the faulting ring, the hardcore ought to (conceptually, at least) signal the fault in ring 1. Ring 1, in the default case, would also have no interest in the fault and would resignal it in ring 2. The ripple-up of the fault would continue until it reached either the faulting ring or else a ring which had some interest in it. In fact, the ripple-up will be made more efficient by not actually signalling in the rings which have no interest in intercepting such signals.

This mechanism could then be used very effectively to debug a number of troublesome kinds of things: fatal process errors due to stack overflow (there would always be space in the monitor's stack to catch faults), fatal process errors due to a fault in the signaler, clobbering the debugger's linkage section or on-conditions (these would all be in a lower ring.)

There is one final additional enhancement which is planned in connection with the ripple-up, that is to gate into ring zero and ripple-up all signals, not just ones automatically trapped by the hardware. The rational here is that the line between hardware-detected and software-detected conditions is very fuzzy--indeed, sometimes the very same condition is caught the one way and sometimes the other--so the signaling mechanism should not attempt to preserve or invent a distinction. The uniform signaling of all conditions in a lower ring assures that the monitor be kept abreast of and in full control of whatever contingencies arise in the application program. Again, this will not be inefficient in the case where all lower rings are uninterested.

A new ring zero gate will be provided to allow a user to establish a signaller for each ring. This gate will keep a vector of pointers in the "pds", one for each ring. The initial value of this vector will make the system behave as it does today.

All signals will be directed to ring zero where the "pds" will be inspected to see which ring should receive the signal. A signaller called to respond to a signal originating in a higher ring would be a user-written signaller interested in trapping certain signals before they reach the ring of occurrence. It would either handle the signal or would return to ring zero via a gate that would select the next interested ring. A signaller called to respond to a signal originating in its own ring or a lower ring would look for a handler in its own ring, finding none it would invoke the default error handler. The default error handler would either sucessfully return to the interrupted program or would call back to ring zero via the gate that will select the next ring to receive the signal, normally this would be the calling ring. If no more rings are interested, the standard error messages are written and the user is asked to indicate whether he wants to start, hold or release. (program interrupt may be a fourth alternative, but its effect is achieved by "hold" followed by "pi").

7. PROBLEM:
Scattered throughout the system are routines that try to identify the procedure that owns the stack frame from which a signal originated. These routines are not always consistant and must all be updated when changes are made to system conventions or data formats.

SOLUTION:
We propose two levels of routines to perform this function. A high-level routine will return, among other things, the character-string representation of the entry name used the invoke the procedure that owns the first standard stack frame preceding the most recent"signal_"stack frame. It will also return an integer that gives the word offset of, the instruction or the statement number of the statement that caused the signal. Lower level routines will also be provided.

The PL/I compiler will be modified to accept a new procedure option ('support'). This option will cause a flag to be set in the stack frames created by activations of the procedure. The two routines used to perform the

the identification of the signalling procedure will ignore stack frames containing this flag and return the information corresponding to the most recent stack frame without the flag.

This option can then be used on language support subroutines whose signals should appear to have originated in their calling procedure.

When these changes have been made PL/I will properly support the "onloc" built-in function, and error messages issued by the default error handler will always contain an identification of the offending procedure and statement.

8. PROBLEM:
The arguments passed to a condition handler and the values accessible via the PL/I builtin functions onsource, ondata, onfile, oncode, onfield, onkey, and onloc are essentially the same kind of data and should be handled in the same manner.

SOLUTION:
Since each condition handler may be interested in only some of the data, and since new condition data may be introduced into the system together with new conditions, passing these values as arguments is impractical. Furthermore, PL/I on-units cannot receive arguments. Consequently, we propose to make the values now passed to condition handlers as arguments accessable via functions. The pll_ondata used to support PL/I condition built-in functions will be extended to accomodate the new information. Ondata will be kept in signal_ automatic storage and will be correctly set for all signals regardless of their source.

9. PROBLEM:
The fim does not sort combined hardware detected signals into the distinct signals required by PL/I. This causes incorrect execution when a user-supplied default handler has been established, unless it knows how to separate these conditions. On the 6180 an illegal procedure fault should sometimes signal the stringsize condition and sometimes behave as a nop. This sort of dirty work should be hidden in the fim, not spread around in everybody's default error handler.

SOLUTION:
The fim will separate the arithmetic conditions into: underflow, overflow, fixedoverflow, and zerodivide. It

should only signal zerodivide when the divisor is zero. (The fault also occurs for some nonzero divisors). A fixedoverflow resulting from a EIS decimal instruction should signal the size condition. An illegal procedure fault resulting from an EIS move whose second operand is a null string should signal the stringsize condition if the detection bit of the instruction is on, and do a nop if it is off.

10. PROBLEM:
The snap and system options of PL/I are not properly supported by the signal mechanism.

SOLUTION:
The on-unit data placed in the stack frame for a given condition handler will contain two flags, indicating the presence of these two options. On encountering the snap flag, the signaller will call debug and then call the handler. On encountering the system flag, signal_ will call the default error handler. This latter feature allows a programmer to establish the default error handler for a specific condition rather than for all conditions.

This MSB is an outline of the proposed changes. More complete design documentation is being prepared. The detailed design and implementation will minimize the inconvenience to users of the existing signal mechanism by providing compatible interfaces or write-arounds whenever possible. Some changes will probably be required, but the increased facility provided by the new mechanism should be worth the price.