MTSS DESIGN NOTEBOOK

Appendix G

FROM: J.H.Saltzer

SUBJ: User Interrupts, Commands Console I/Ø, and User Option Switches

Console Button Interrupts

It has been well established that there is a need and a use for a way for a user to control his program by originating an interrupt signal from the console in certain special cases, such as to cut off printing when too much was requested, or to return to a debugging supervisor when a loop is suspected. The scheme developed in CTSS has some defects. The problem with the CTSS interrupt scheme is that a program cannot effectively control when an interrupt will occur. Writing a program which is capable of accepting an interrupt at any time during its execution is somewhat tricky, as is well known. What is needed is some way of allowing interrupt signals to be remembered without the trap occuring, in much the way that data channel trap signals are handled by the 7094 hardware. The following supervisor subroutines would be needed:

EXECUTE SETBRK.(LABEL)

would cause two actions to take place:

1. Console button interrupt signals will be remembered.
2. Statement label LABEL is defined as a trap return location.

EXECUTE RELBRK.

will cause remembered interrupt signals to be lost, and later interrupt signals to be ignored.

EXECUTE ENBBRK.

will "enable" an interrupt signal trap, either on a remembered interrupt signal or a new one occuring after the subroutine was executed.

EXECUTE DISBRK.

will disable interrupt button traps, although trapping signals will still be remembered for a later enable call. When an interrupt signal trap actually occurs, further traps should be automatically disabled (and remembered), until another call to ENBBRK. has been given.

Depending on what other traps are available to the time-sharing programmer, the above-described scheme should be carefully incorporated into the overall trap environment.

A detailed proposal for such an overall environment on the 7094 was suggested in programming staff note 25, a copy of which is attached, and the general principles outlined there still apply to a new computer system.

Storing of Programmer Options

Recently, a number of public commands have provided several user options. Examples are the erase and kill characters of TYPSET; the brief modes of ED, E (Professor Samuels' editor), and TYPSET; the (LIST) and (SYMT) modes of the new MAD translator, and the (LIST) mode of the FAP translator. Several other examples may be found, but these few illustrate well the point.

Often, these options are invoked by some users every time they use a command; the brief mode of the editors and the (LIST) mode of FAP are examples. This constant usage is really more a property of the user than of the particular files he is working with. In the case of the editors, a simple brief mode switch is not really enough; many users would like to suppress certain program responses but not others; again the particular desired pattern of usage is primarily a property of the user and only secondarily a property of the particular usage of the command. (Note that the elaborate conventions needed to specify a particular pattern every time that a command is used serve to inhibit development of commands allowing such tailor-made patterns of usage.)

It would seem sensible, therefore, to provide some means for any command to store permanently, in a group of program settable switches, the particular modes desired by this user. These switches would be preserved from one use of a command to another, and even from one logged in session to another. Thus, once a user builds up a familiar pattern of usage with a command, he may continue to use it the same way without having to respecify a flock of options each time he uses the command.

Commands as Subroutines

It would seem logical to program all supervisor commands as subroutines with a standard calling sequence; in this form they would be easily accessible to user programs wishing to use command programs. The typing of a command at a console with parameters, e.g.,

```
COMMAND  A B C D E F
```

would be tatamount to requesting the supervisor to execute the following program:

```
EXECUTE CØMMAND.($A B C D E F$)
EXECUTE DØRMANT.
```

Since the parameter string is reflected directly to the command program as a hollerith string, there should be no restrictions on the format of the parameter string. One could thus create a desk calculator command, for example, which was called into action by

```
CØMPUTE 3.14159*2.0/17.5 + 31.55775
```

All commands would normally terminate with the statement

```
FUNCTIØN RETURN
```

thereby allowing the calling program to decide what should be done next. This convention eliminates the need for the elaborate command chaining procedures which were developed for CTSS.

Console I/Ø as seen by the User's Program

The user should be allowed precise control over when his program is needed in core memory to look at his typed input. This may be achieved by providing an input subroutine with the following specifications: A program may read console input by:

N = RDFLX.(BUFF,M)

where BUFF is an array in the user's program large enough to hold M characters. M is chosen by the user on the basis of his application, and need not be a multiple of four. Thus, an INPUT command might make M = 4000, while an interactive user program might set M = 5. The supervisor will not call the program back into operation until either M charachters have arrived, or a break character which may be specified by the user has arrived. In either case, N is set to the actual number of characters placed in BUFF. Although the characters may be initially stored in a floating buffer pool in a supervisor segment, the user should not have to be aware either of this fact or the maximum size of the supervisor's buffer. (If the user may gain efficiency by making his buffer size the same as the supervisor, this should be at his option.)

On the subject of program-specifiable break characters, there should be an easy technique for making any one or a group of the available character set a break character. (For special applications, every character could be a break character. This effect can be cheaply achieved by setting M = 1, above.) A common, easy to use convention which has been used in several CTSS commands is that two consecutive carriage returns comprise a break character. The ability to specify two carriage returns as a break character should therefore also be established in the supervisor.

With the advent of the 8-bit character set, conventions for erase and kill characters must be worked out again. It is fairly clear that as long as otherwise legitimate console characters are used for these functions, there must be some way of changing which actual characters are used for the purpose. It also seems desirable to have the supervisor perform this processing, as otherwise every subprogram which uses console input will have to provide the coding, and a multiplicity of conventions will spring up.

It is clear that both break-character and erase-character processing must be done after a user's call to the RDFLX subroutine, not before, since he may want to change the break- or erase-character set just before the call.

Following the same general philosophy, a subroutine for output on the console should work as follows:

EXECUTE WRFLX.(BUFF,N)

will type out N characters from the buffer at BUFF. N may be any number, not necessarily a multiple of four, and not limited by supervisor buffer sizes. An alternate subroutine, WRFLXA, as in CTSS, should also be provided.

From: J. H. Saltzer, T. N. Hastings, and R. C. Daley

Subject: Unified control of enabled user traps, including memory protection and relocation.

There are now available to a foreground user a number of "traps" which are requested and enabled by the user via a variety of supervisor subroutine calls. These traps include a high speed I/O (data channel) trap, clock trap, interrupt button trap, and the floating trap mode. The various conventions used in these enabling calls can be simply collected into a single call to a supervisor program, including provisions for specifying operation of the protection and relocation mode. The supervisor subroutine which is called would be an interpreter, and would be used typically as follows:

```
TSX   $ENABLE,4
LPI   A
LRI   B
ENB   C
AXT   **,4
TRA   **
```

The calling sequence begins in location 1,4 and ends with the TRA instruction; in between are any number of trap enabling instructions or NØP's in any order. The supervisor subroutine would enable all (permitted) requested traps and modes and interpret the AXT and TRA instruction thereby providing an automatic grace period before any traps or modes take effect. (The AXT is optional.) Following a trap of any kind which was enabled by the ENABLE sequence, all traps are inhibited; if a data channel, clock, or interrupt button trapping signal occurs it is remembered until the next ENABLE sequence, which enables that trap. Instructions permitted in the calling sequence are:

```
LPI   LPTM   ETM
LRI   EPTM   LTM
ENB          IXH   LIST,,N   (the IXH is a pointer word to
                               an enable list of the form
                               specified in CC-226.)
```

Note that a basic philosophy in the specification of the ENABLE subroutine is that the original hardware trapping procedures are imitated as closely as possible. Two new modes have been introduced with this call; their proposed operation is described below.

Two "supervisory" applications of the time-sharing system have need for some sort of core memory protection and collection of other protection mode violations; no doubt other supervisory applications could use these features. The two applications referred to here are the grading and monitoring of student class programs, and the operation of debugging programs such as FAPSUG or MADSUG. In both of these applications, an objective is for a core B supervisory program to maintain control no matter what an undebugged program happens to do by mistake. In the caseof a class an additional objective is to maintain security, say, of grades, and avoid cheating or similar malicious actions by a student.

Memory Protection Mode.

Following a call to ENABLE which specified a setting of the protection mode register, the operation of the computer as seen by the user's program would be modified: all memory protect violations including illegal instruction traps and core-A supervisor calls would cause all traps to be disabled and control to pass to the appropriate lower core location (33_8). Following a trap actual memory protect would still be in operation with memory bound returned to its earlier position and core-A supervisor subroutine calls honored normally.

Since the core-A supervisor is normally in the business of sorting out the meaning of protection traps, it would be convenient if it could do such a preliminary sorting before returning to core-B simulating a trap to the user program; a trap code could be placed in the decrement of location 32 along with the ILC at the point of violation. The following memory protection violations might be distinguished:

1. TIA to a legal supervisor subroutine.
2. Other TIA's and illegal instructions.
3. Core protection violation.
4. HTR encountered. (?)

Relocation Mode

If a call to ENABLE specified a setting of the relocation register, all addresses beginning with the address of the TRA at the end of the calling sequence are relative to the new relocation setting.

If a trap of any kind occurs while in relocation mode, the unrelocated ILC is stored, the relocation register is reset to its value before relocation mode was entered, and the trap made to the user's apparent absolute lower core location. Thus it is possible for a controlling program to get back to the interrupted program by placing the contents of location 32 in the address of the TRA at the end of the next ENABLE sequence.

Other desirable features: Although it would probably require a little more effort to add, the following feature would be very useful to a supervisory program attempting to interpret another program or permit the other program certain supervisor calls. If the TRA instruction in the calling sequence is a TIA to a supervisor subroutine name, this would signify that the supervisor subroutine should be called after IR4 is reloaded from the address of the AXT, but that the return from the supervisor subroutine should be considered to be a protection violation. (That is, a trap occurs after the subroutine has finished). The supervisor subroutine would respect the core-B user's setting of the relocation and protection registers.

When this special kind of "delayed" trap occurs, the ILC location would be filled with the relative location to return in the core-B program following the permitted supervisor subroutine call. It would be useful to add two more trap codes to the decrement of the ILC location:

5. Returning from a supervisor subroutine.
6. Error return from a supervisor subroutine.

Relation to older trap-enabling procedures.

Unfortunately, a number of traps have already been implemented in the time-sharing supervisor, with diverse techniques of specification. Here, therefore, are several suggestions for unification which will have to be reviewed very carefully in the light of the amount of reprogramming of user programs they might cause.

Clock trap.

The clock trap should be a precise imitation of the 7094 interval timer procedure. A call to supervisor subroutine CL¢C¢N causes location 5 to begin incrementing, but unless the clock is enabled overflow only causes a trapping signal, not a trap. This signal is remembered until the next ENABLE sequence which specifies an ENB instruction which enables the clock (bit 17 of the enable word). Supervisor subroutine CL¢C¢F stops further incrementing of cell 5. Thus CL¢C¢N and CL¢C¢F act only as the console on-off switch.

Console interrupt button trap.

With the above techniques in mind, the console interrupt button feature can be easily incorporated into the sequence by enabling it with an unused bit in the enable word, such as bit 18. The trap return locations would be set by a call to SETBRK, and such a call would cause all future interrupt signals to either cause a trap or be remembered, depending on whether or not an appropriate ENB instruction had been received. Since the user program has both memory protection and the ability to remember and put off interrupt button traps until it is able to handle them, the need for interrupt levels is eliminated.