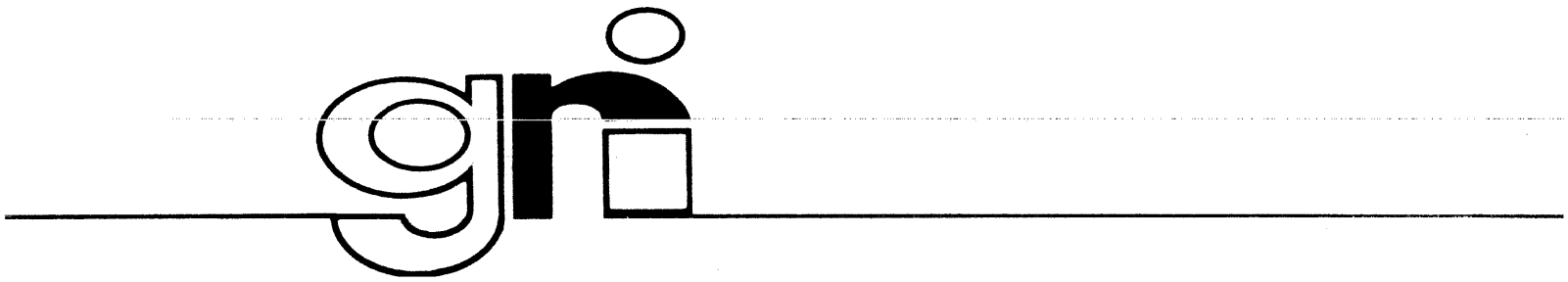


real time executive (% rtx)

GRI Computer Corporation
320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

GRI

REAL TIME EXECUTIVE

% RTX

GRI Computer Corporation, 320 Needham Street, Newton, Massachusetts 02164

70-44-001
0772-0300

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER THREE - continued

LIST OF FIGURES

GRI Real Time Executive

Introduction:

The GRI Real Time Executive (%RTX) is a generalized controller which can be utilized to structure a particular real-time system aimed at a particular application. It utilizes only 224_{10} core memory locations and has a completely modular organizational approach. The use of %RTX, in most cases, simplifies and shortens the tasks involved in designing and coding such a real-time applications system; and in many cases, allows the user options which would not be economically available to him in terms of time spent in designing and implementing such a controller himself.

In order to understand the specifications of the various modules which can be assembled into an executive applicable to the task at hand, it is necessary to understand how %RTX expects a real-time system to be structured. The overall structure may be separated into four categories: 1) Control Programs, 2) Interrupt Service Routines, 3) Task Processors, and 4) Shared Subroutines. The actual functions of modules in these four categories in any given system may overlap somewhat, but their basic definitions in functional terms is as follows:

1) Control Programs

These are the programs which do the bookkeeping and keep track of machine states necessary to coordinate the activities of the other units in the system. It is intended that these functions be provided for by %RTX and its modules. (It should

be pointed out here that this is not the same as what is usually referred to as a monitor although one possible applications of %RTX might be to serve as the nucleus of a monitor.)

2) Interrupt Service Routines

These are the routines which are entered in response to an interrupt, usually a unique routine per unique interrupt. They can range in complexity from a simple acknowledge to a complete task depending on the system design and/or timing requirements. $RTX requires certain minimal communication with these routines in order to save and restore registers and flags correctly.

3) Task Processors

These are the modules which actually perform the tasks required for the user's application. %RTX also requires certain communication with these modules in order to preserve system integrity. Normally, the Task Processors will comprise the major programming effort for a typical user, although special devices may make it necessary for the user to write, in addition, some of his own Interrupt Service Routines.

4) Shared Subroutines

These are subtasks which need to be performed by more than one of the Task Processors in the system. Generally speaking, if a subtask is relatively short, it should be incorporated "in line" with each of the processors which require it. If it is a lengthy routine, however, core space limitations may require that it be utilized as a shared subroutine. The coding of such a subroutine is no different than that of a normal subroutine not used in real-time. The call to it from the Task Processors which use it, however, involves the

use of the $ATCH and $SREL routines supplied in the %RTX package. These routines attach and release shared subroutines to or from the processor requiring them.

A very general pictorial representation of the relationship between these four categories in a typical system is offered in Figure 1.

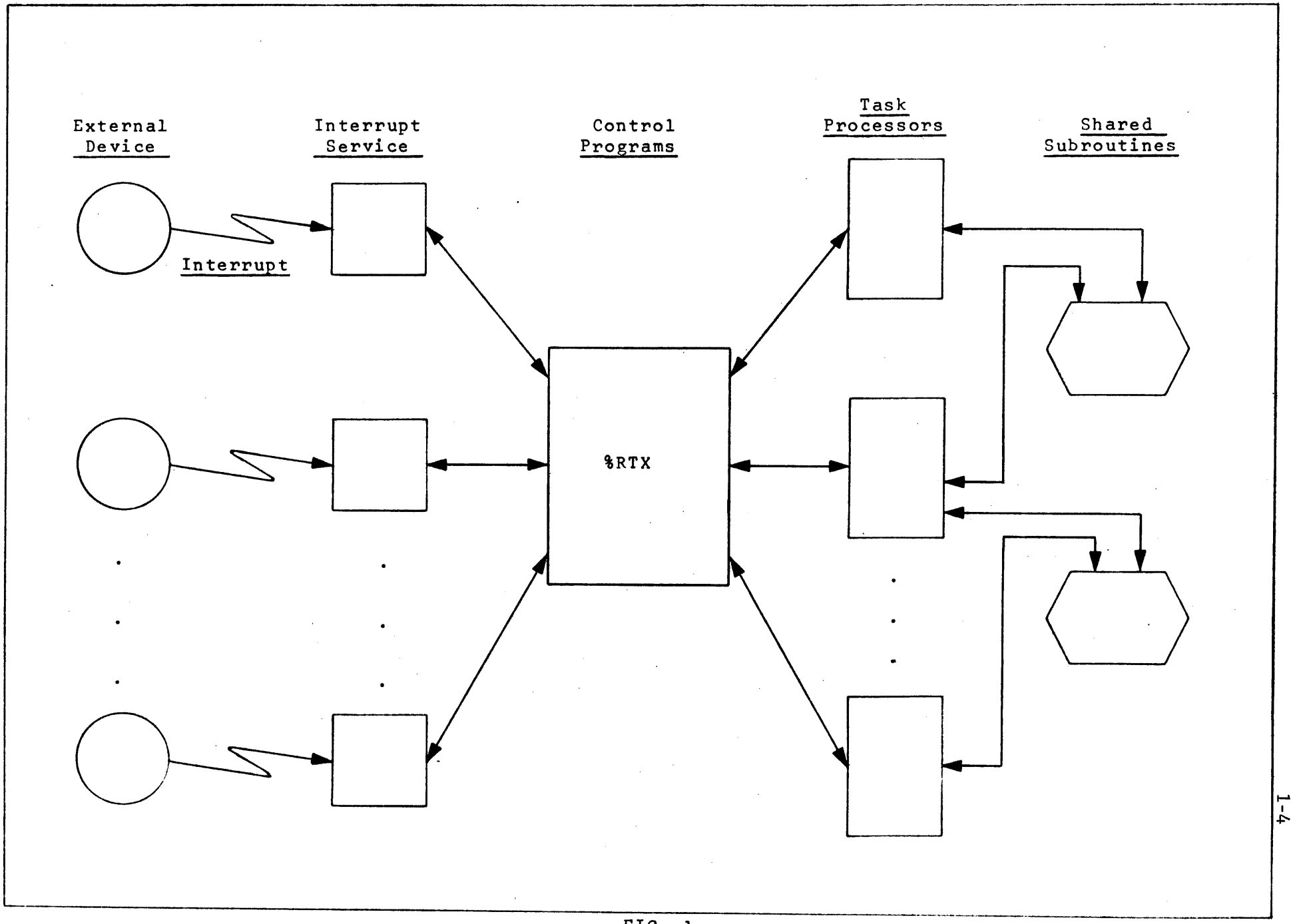

External Device
Interrupt Service
Control Programs
Task Processors
Shared Subroutines
Interrupt
%RTX

FIG. 1

System Structure:

After initialization and startup of a system, as well as during the periods when a real-time system is idle, %RTX enters a scan or polling mode. In this mode, %RTX continually and sequentially invokes each user written Task Processor. In order for %RTX to know where each processor in the system is, and in what state its operations are in, it must reference a processor list which is supplied by the user during startup. This list also defines the priority of the processors in the system since the entries are polled from top to bottom and the entries toward the top of the list have a tendency to be polled more often. The highest priority processor is at the top of the list, and successively lower priority processors occupy successively lower positions on this list.

Each of the Task Processors has four basic parts: 1) a save area, 2) prologue, 3) main body, and 4) exit. The save area is used by %RTX to save registers and processor status when a Task Processor is interrupted. Thus, the save area must be at least six words long -- one word for processor status and one each for the AX, AY, MSR, TRP, and SC registers. If the user's system has other registers that need to be saved, sufficient space must be provided to save each one, and the user must supply the code to save and restore them -- this procedure is covered in detail under the specifications sheets for $USAV and $URES. The prologue is generally a series of tests which determine whether the task for this processor needs to be performed, i.e. whether the main body should be entered. If the prologue determines that the task does not need to be performed, it simply issues a return to %RTX which causes it

to invoke the next lower priority processor in the list. If it decides the task needs to be performed, it continues on into the main body of code. The latter, when completed, must exit to %RTX. There are two choices of exit -- one which causes %RTX to invoke the next lower priority processor (if any) on the list, and another which causes it to begin scanning at the top of the list again.

Generally, most or all of the code within a Task Processor -- including any Shared Subroutines, as well as portions of %RTX itself, are interruptable. This means that the "interrupt active" flip-flop within the GRI-909 is in the "on" state (via an FOI ICO instruction). Exactly which device or devices are allowed to interrupt the system during such times depends on the setting of the interrupt status register (ISR) and on which devices have been selected. These conditions are under user control within his Task Processors since they may initiate input/output or otherwise start devices. The only restriction applicable to this manipulation is that whatever is done to ISR must also be done to the contents of a location within %RTX labeled $ISR.

When an interrupt occurs, the "interrupt active" flip-flop is automatically turned off by hardware so that it is as if an FOI ICF had been issued just prior to entry to the interrupt service routine which handles the interrupt. There are two types of Interrupt Service Routines -- those which operate with the interrupt active off (or send a zero to the ISR and turn interrupt active on, thereby allowing the power fail interrupt only), and those which operate with interrupt active on to allow other devices to interrupt. The former are referred to as "ICF type" or "non-interruptable" Interrupt Service Routines, and

the latter as "ICO type" or "interruptable" Interrupt Service Routines. %RTX must be informed which type of Interrupt Service Routine is currently operating. For an ICO-type Interrupt Service, %RTX is so informed by a call to $ICO in %RTX before entering the routine, and by exiting with either a call to $EICO or $NICO. For an ICF-type, %RTX is so informed by the exit from the interrupt service which is a call to either $EICF or $NICF.

The differences between the two types of returns to $RTX from an Interrupt Service Routine are analogous to the two returns to %RTX available to a Task Processor. The $EICF or $EICO are called "end-mode" returns and cause %RTX to save all the registers for the Task Processor which was interrupted, and when all current interrupts have been taken care of, resume scanning at the top of the priority list which may result in invoking a processor other than the one which was interrupted. The "end-mode" returns are generally taken when the interrupt signals a significant change in the system. For instance, suppose the Interrupt Service detected a carriage return from teletype which usually indicates the end of a line of input. this instance, an entire line is now in the computer as opposed to the previous teletype interrupts which delivered only one more character into the line. The state of the system has changed significantly since the information in the input line can now be processed instead of merely collected. If the Task Processor which takes care of this is near or at the top of the priority list and the Interrupt Service Routine takes an "end-mode" return, the line will be processed as soon as possible. On the other hand, the $NICF and $NICO

return from an Interrupt Service Routine cause no alteration of the scan pointer in %RTX so that when all current interrupts are taken care of, %RTX resumes by invoking the Task Processor which was interrupted.

The following running commentary of the flow of control through a hypothetical real-time system will give some idea of the bookkeeping and control maintained by %RTX. Suppose there are three Task Processors A, B, and C in the processor list and there are three Interrupt Service Routines I, J, and K where I and J are ICO types and K is an ICF. Further suppose that when I has been entered, the ISR is set to allow both J and K to interrupt it and when J has been entered, it allows only K.

Let us say that the system has just been loaded and started so that %RTX begins by invoking processor A whose task might be to start up the three devices which correspond to I, J, and K. When this is done, A exits and %RTX invokes B, and then C (both of which might have nothing to do until some device interrupts). The processor list being exhausted, %RTX returns to the top and invokes A again, which since it has started the devices now has nothing to do. It then invokes B, C, A, etc. until finally somebody interrupts, say the device which goes to K. Let us say that K issues an end-mode return forcing %RTX's scan pointer to the top of the list to invoke processor A. This processor sees that K must be restarted and so enters its main body code to accomplish this, then exits to %RTX. Now %RTX invokes processor B, which, say, has something to do in response to the interrupt from K. But before B can finish, an interrupt comes from the device which goes to Interrupt Service Routine I. %RTX is informed by I that it is an ICO type and to allow interrupts

from J and K. Before I can finish servicing its interrupt, J interrupts it. J finishes and takes an "end-mode" return to %RTX. Since J interrupted I which is an Interrupt Service Routine, %RTX simply remembers the "end-mode" condition and resumes I at the point it was interrupted. When I finishes, %RTX resumes scanning at the top of the processor list due to the "end-mode" return from J (even if I made a "normal" return). Thus, %RTX invokes processor A, which restarts devices I and J. Then %RTX invokes B. Since B was in the middle of some process when it was interrupted, %RTX invokes B by restoring all the registers to what they were when I interrupted it and continues. And so forth....

Active and Inactive States:

Note that in the above description, there are two different ways in which %RTX invokes a processor. In one case, it gives control to the processors prologue which decides whether to enter the main body of code. In the other case, %RTX resumes the main body of code at the point it was interrupted. These two different modes of entry will be called inactive entry or active entry respectively. Also, a processor is in the inactive state, or simply inactive, if it will be reinvoked via an inactive entry should it be interrupted. Similarly a processor is in the active state, or simply active, if it will be reinvoked via an active entry should it be interrupted.

The difference between the active and inactive states is that if a processor is interrupted while it is active, all registers are saved, and when it is invoked again, the registers are restored and processing continues at the point of

interrupt; whereas if a processor is interrupted when it is inactive, no registers are saved, and when it is invoked again, it is entered at a specified location rather than continuing at the point of interrupt. The address -1 of this specified location is in the first word of the processor's save area. This word is Ø if the processor is in the active state.

The state (active or inactive) of a processor at a given point in its operations is extremely important to the successful completion of its assigned task. When the user is writing a section of code for a processor in the system, he should make sure which of the two states he wants that section to be operating under and set up the first word of his save area accordingly. A general rule for making this decision is that if the section of code is referencing things that can change via an interrupt (either by an Interrupt Service routine or a higher priority processor which might gain control via an "end-mode" return) and if such a change would make it meaningless to continue to the completion of the code, that section should be inactive. Otherwise, the active state is generally desirable. Typically, a prologue is inactive since it usually involves testing things that change, and the first instruction of the main body would be to set the processor active (by zeroing the first word of its own save area). For this reason, whenever %RTX enters a processor's prologue, it is in the inactive state.

CHAPTER TWO

REAL TIME EXECUTIVE (%RTX)

Length: $340_8(224_{10})$

Entry Points: Detailed descriptions follow

Name	Function
$STRT	start scan, main entry
$PRCL	contains address -1 of processor list
$NEXT	return from inactive processor (e.g. prologue) which allows next lower priority processor to be invoked.
$RSME	return from inactive processor which allows resumption of another specified processor.
$NREL	return from active processor which forces processor scan to begin at top of priority list.
$EXEC	return from active processor which allows next lower priority processor to be polled.
$ACTV	subroutine callable by processor or interrupt service routine to set itself active.
$ICO	must be called at beginning of ICO-type interrupt service routines.
$SAVA	contains address of current save area.
$TRP	save location for TRP register prior to calling $ICO, $EICF, or $NICF.
$EICO	end mode return from ICO-type interrupt service routine, forces processor scan pointer to top of list.
$NICO	normal return from ICO-type interrupt service routine, processor scanner not altered.

Name	Function
$ISR	- system interrupt status word.
$PROL	- contains address -1 of current processor's prologue entry.
$SCAN	- contains address of second word of current entry on processor list.
$REST	- contains processor save area address during restoring registers.
$EICF	- end mode return from ICF-type interrupt service routine (see $EICO).
$NICF	- normal return from ICF-type interrupt service routine (see $NICO).
$INTL	- contains address -1 of interrupt location for ICF-type interrupt service.

The most commonly used entry points are discussed first. The others are used generally by the special sharable subroutines supplied with %RTX such as $ATCH or $GETM, etc.

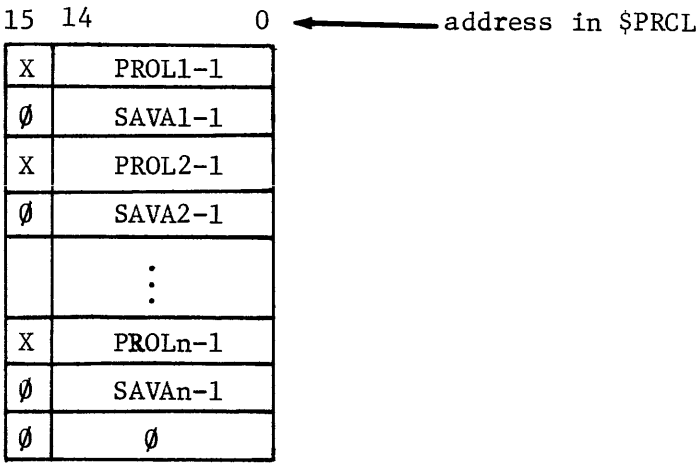
1) $STRT

This is the call to %RTX which starts the scan after the user has submitted his processor list (see below) the interrupt active flip-flop should be off, as will normally be the case during start-up since depressing the Start switch on the console automatically turns it off.

2) $PRCL

During initialization, the user must submit a processor list to %RTX. This is done by storing the address -1 of the beginning of the list in $PRCL. The format of the list is as follows:

Processor List



PROL1 is the address of the highest priority processor's prologue, and SAVA1 is the address of its save area. PROL2 and SAVA2 are for the next lower priority processor, etc. The list ends with a zero word.

Bit 15 of the PROL entries are normally = Ø. Should Bit 15 be set to 1, %RTX will skip that processor entirely during the scan - this allows the user to exercise some extra control over which processors can be invoked at any given time.

Also, when a processor list is initially submitted, the first word of each processor's save area should contain the address -1 of its prologue (i.e. the same address as is in the list). This is a required initial condition, the actual contents of this word will change during operation. It is suggested that this initial condition be taken care of during the assembly of the processors so they will be loaded with the proper value in the first word of their save areas.

3) $NEXT

This is a return to %RTX which causes %RTX to bump its scan pointer to the next processor on the list. This return can only be made from an inactive processor -- if this is called from anywhere else, %RTX will blow up.

The most common use of this return is if a processor's prologue finds that the processor has nothing to do. It returns control to %RTX via a JU $NEXT (or a JC, etc.), allowing the next processor to be scanned.

4) $NREL

This is one of two types of return to %RTX that can be made from either an active or inactive processor. The processor which issues this return will be set inactive in such a way that the next time it is invoked, it will be entered, inactive, at its prologue. The scanner in %RTX is forced to the top of the processor list by this return.

5) $EXEC

This is the normal exit from an active processor. It does the same thing as $NREL, except the scanner is bumped to the next lower priority processor in the list. This exit can also be called from an inactive processor.

6) $ICO and $TRP

$ICO is called by an ICO-type interrupt service routine after saving the TRAP in $TRP and before it does any actual processing. The call should look like:

```
            RM      TRP,$TRP        ;SAVE TRAP REGISTER
            JU      $ICO
            WRD     INTLC           ;INTERRUPT LOCATION
            WRD     IRMSK           ;ISR MASK
ISAVA:      LOC     .+10            ;8 WORD SAVE AREA
                                    ;BEGIN INTERRUPT SERVICE
```

Where INTLC is the address of the location where the SC was stored when the interrupt occurred, IRMSK is used by %RTX to "AND" against the ISR, so IRMSK should have 1's in all bit positions except those for which interrupts from the corresponding

device are not to be allowed. (In particular, another interrupt from the same device should not be allowed, i.e. IRMSK should have at least one Ø bit position corresponding to the device which just interrupted.)

The area at ISAVA is used as follows:

ISAVA:

Cascade information
Saved ISR
Active/Inactive
Save area for
AX, AY, MSR, TRP,
SC should this
interrupt service
be interrupted

When %RTX returns from the $ICO call, the interrupt active flip-flop will be on, the ISR will contain the result of "ANDing" the old ISR with IRMSK and the interrupt service routine will be in the active state (ISAVA+2 will be set to Ø).

Furthermore, %RTX will have set itself in "interrupt mode" which has certain consequences for saving and restoring registers if further interrupts occur, the main one being that if an ICO-type interrupt service routine is interrupted, only AX, AY, MSR, TRP, and SC are saved in its save area -- no provisions are made for saving additional registers as is the case with processors.

An ICO-type interrupt service routine can set itself inactive the same way a processor can -- and for the same reasons. It does so by setting its ISAVA+2 location to the address -1 of the inactive entry.

The "cascade information" location is set by %RTX to the address of the previous (if any) interrupted interrupt service routine's ISAVA area. A diagram showing pictorially how cascading operates is shown in Figure 2.

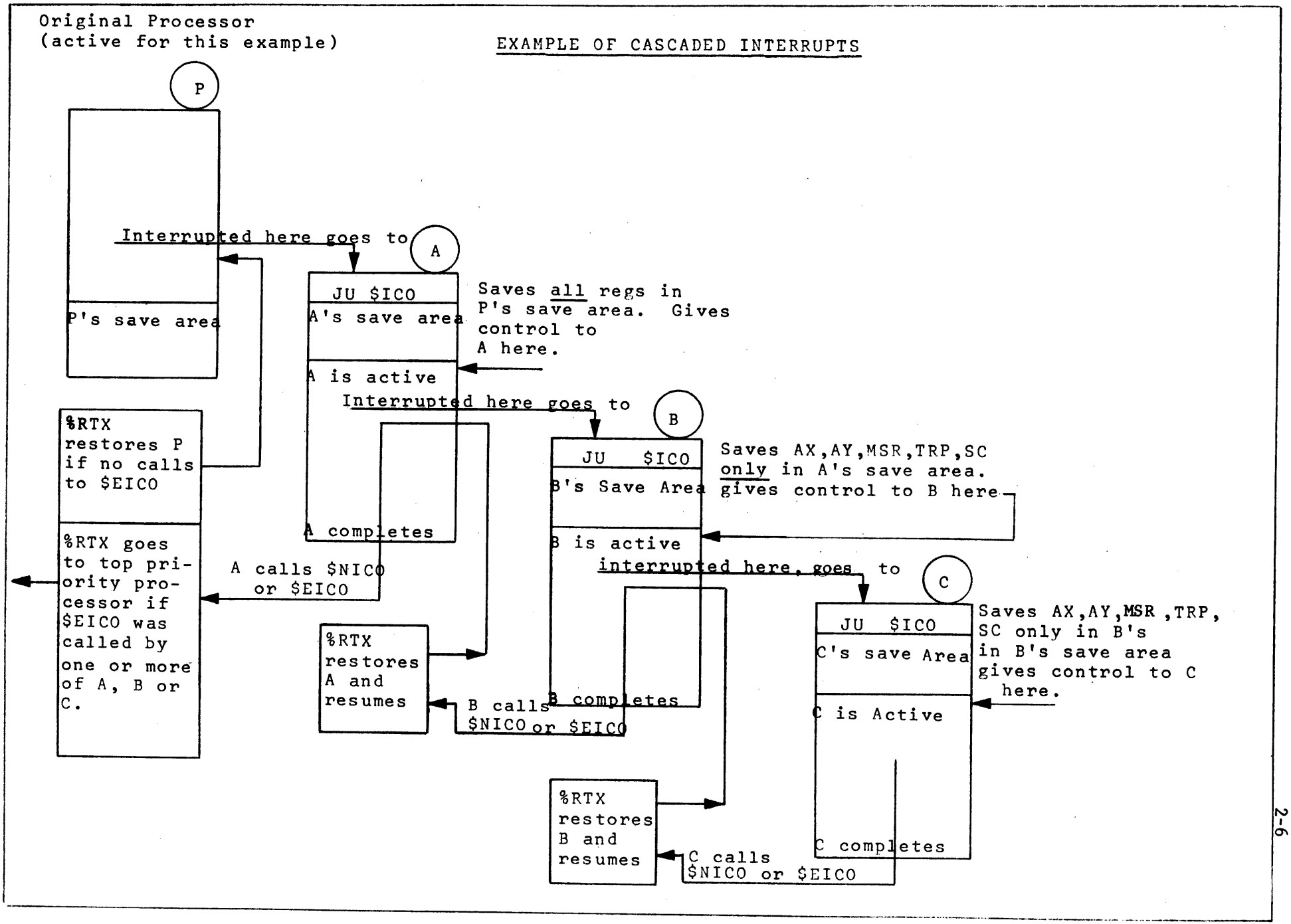

Original Processor
(active for this example)
EXAMPLE OF CASCADED INTERRUPTS
P
P's save area
Interrupted here goes to
A
JU $ICO
A's save area
Saves all regs in P's save area. Gives control to A here.
A is active
Interrupted here goes to
B
JU $ICO
B's Save Area
Saves AX,AY,MSR,TRP,SC only in A's save area. gives control to B here
B is active
interrupted here, goes to
C
JU $ICO
C's save Area
Saves AX,AY,MSR,TRP, SC only in B's in B's save area gives control to C here.
C is Active
C completes
C calls $NICO or $EICO
%RTX restores B and resumes
B completes
B calls $NICO or $EICO
%RTX restores A and resumes
A completes
A calls $NICO or $EICO
%RTX goes to top priority processor if $EICO was called by one or more of A, B or C.
%RTX restores P if no calls to $EICO

FIG. 2

7) $EICO

This may be called by an ICO-type interrupt service routine only. If it is called from anywhere else, %RTX winces. A call to $EICO exits from the ICO-type interrupt service routine and alters the processor scanner so that when all current interrupts have been serviced, %RTX resumes scanning at the top of the list.

8) $NICO

This is the other exit from an ICO-type interrupt service routine. It is the same as $EICO except the processor scanner is not altered.

9) $ISR

This location should reflect what the ISR says when %RTX is in the processor scan. To be safe, whenever the ISR is altered, $ISR should be altered in the same way and the line of code which does these operations should be preceded by an FOI ICF and followed by and FOI ICO.

Note: Do not alter ISR and then store ISR in $ISR. First alter ISR, and then do the same operation on $ISR. Normally (e.g. during processor scanning), ISR and $ISR will be equal, but certain relations between cascaded interrupt routines can cause them to be unequal.

10) $INTL

This location is set up during the entering of an ICF-type interrupt service routine. It should be set to the address -1 of the location in which the SC was stored when the interrupt occurred.

An ICF-type interrupt service routine should save and restore all registers it alters during its operation. Thus, a typical entry to an ICF-type interrupt service routine is as follows:

```
INRUP:   RM    TRP,$TRP          ;SAVE TRAP REGISTER
         MRI   INTLC-1,TRP       ;SET UP ADDR-1
         RM    TRP,$INTL         ;OF INTERRUPT LOCATION
         RMI   AX,Ø              ;SAVE REGS
         RMI   AY,Ø
         RMI   MSR,Ø
         RMI   ISR,Ø             ;THESE 3 INSTRUCTIONS
         ZR    ISR               ;ALLOW POWER FAIL
         FOI   ICO               ;AND ARE NOT VITAL
          :
          :
```

The last three instructions need not be included if allowing a power fail interrupt is not crucial while the ICF-type service routine is operating. Similarly, the saving of one or more of AX,AY or MSR may be omitted if it is not altered -- although typically all of them will be.

11) $EICF and $NICF

These are the exits available to an ICF-type interrupt service routine. They are analogous to the $EICO and $NICO exits available to an ICO-type interrupt service routine.

Since an ICF-type interrupt service routine saved registers upon entry, it must restore them upon exit. To continue the example above, the exit from that routine would be:

```
MR      INRUP+7,AX              ;RESTORE REGISTERS
MR      INRUP+11,AY
MR      INRUP+13,MSR
FOI     ICF                     ;ONLY FOR POWER
MR      INRUP+15,ISR            ;FAIL OPTION
JU      $NICF (or JU $EICF)
```

Where the two instructions beginning at the FOI ICF are not necessary if the power fail was not enabled on entry. Note also that the TRP need not be restored. %RTX will restore it from $TRP.

If $EICF or $NICF are called from anywhere but an ICF-type interrupt service routine, %RTX grumbles.

The less commonly used entry points are $ACTV, $RSME, $SAVA $PROL, $SCAN and $REST. They are present primarily for such routines as $ATCH, $GETM, etc. which need to reference certain locations in %RTX.

12) $ACTV

This entry point merely zeroes the first word of the save area of the current processor or the ISAVA+2 location in the case of an Interrupt Service Routine. ie.,it sets the current processor or Interrupt Service Routine active. Usually this can be accomplished by a ZM instruction. However, in some cases, notably in certain types of shared subroutines, the location to be zeroed may not be known. The user is not likely to encounter this situation.

13) $RSME

If this is called with AX set to the save area address -1, and AY set to the prologue address -1 of a given processor in the system, that processor will be invoked correctly and immediately. The scan pointer remains as it was so that if the invoked processor calls $NEXT or $EXEC, the scan pointer is updated to the next lower priority processor below the processor which called $RSME. The interrupt control flip-flop must be off before $RSME is called (i.e. issue a FOI ICF and then go to $RSME).

The primary purpose of $RSME is to accommodate the $ATCH and $SREL routines. It is also used by $GETM.

14) $SAVA, $PROL, $SCAN

These are adequately described in the list of entry points given on pages 10 and 11.

15) $REST

A call to $REST-1 with AX set to a save area address will restore registers and resume an interrupted, active processor.

Again, this entry point is defined for use by some of the other routines supplied with %RTX and is normally not called by user programs.

Notes:

1) $ACTV and $ICO are the only entries in %RTX which return control to the caller immediately following the call. No registers (except the TRAP) are altered by $ACTV.

2) Whenever %RTX transfers control to an inactive entry, only the following conditions are guaranteed:

a) AX = address -1 of current save area
b) AO is in ADD state
c) BOV is clear
d) LNK is clear
e) ICO is on (i.e. a FOI ICO has been executed)
f) The contents of the save area will not have been altered in any way.

3) Whenever %RTX transfers control to an active processor or Interrupt Service Routine, the following conditions are guaranteed:

a) All registers, except possibly the ISR, will contain the values they had at the time of the interrupt. This includes the SC, meaning that the program resumes as if nothing had happened.

b) The contents of the save area may have been altered in order to save (and subsequently restore) the registers.

c) ICO is on (i.e. a FOI ICO has been executed).

A pictorial diagram of the differences between an inactive and active state is shown in Figure 3.

4) Should the system need to provide for saving registers in addition to AX, AY, MSR, TRP and SC (e.g. 6 GPR) the procedure involves using a version of %RTX with calls to $USAV and $URES and

providing two routines with entries $USAV and $URES. This is described in detail under the specifications for $USAV and $URES.

5) It is possible to dynamically submit a new processor list. For instance, during the operation of the system a condition may occur which requires a different set of processors to handle than those currently operating; all that needs to be done here is to submit a new processor list.

A new list can be sumbitted at any time by anybody except a shared subroutine or within the scope of an attach (see $ATCH). The procedure is to simply store the address-1 of the new processor list into $PRCL. To assure that %RTX begins scanning the new list immediately, do the following:

a) If the new list is being submitted by an Interrupt Service Routine, exit via an "end-mode" return (i.e. $EICO or $EICF).

b) If the new list is being submitted by a Task Processor, start the procedure by issuing a FOI ICF, then store the new list address -1 in $PRCL, and then exit via $NREL.

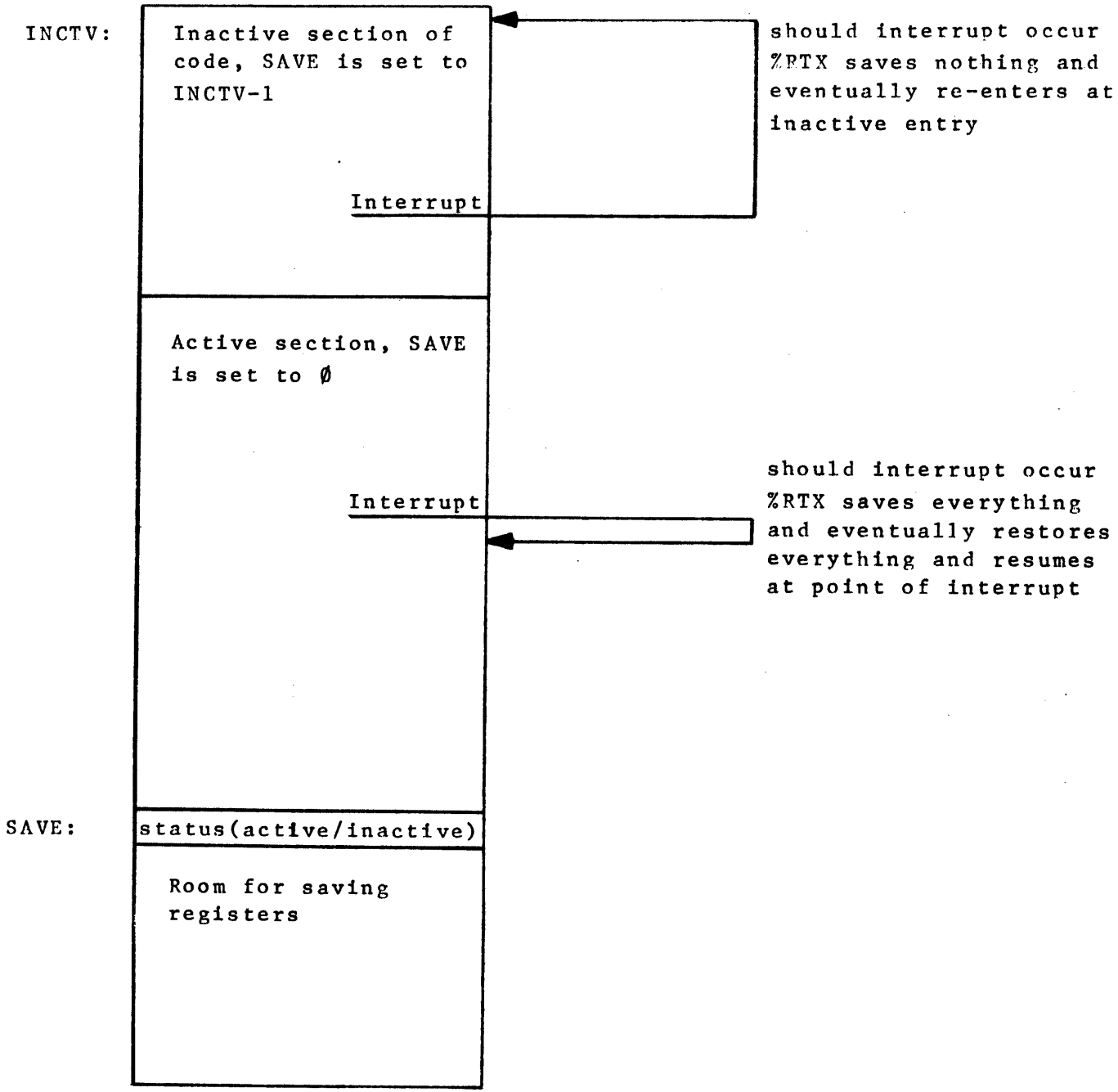

INCTV:
Inactive section of code, SAVE is set to INCTV-1
Interrupt
should interrupt occur %RTX saves nothing and eventually re-enters at inactive entry
Active section, SAVE is set to ∅
Interrupt
should interrupt occur %RTX saves everything and eventually restores everything and resumes at point of interrupt
SAVE:
status(active/inactive)
Room for saving registers

FIG. 3

USER SAVE/RESTORE

These are user written routines to save and restore registers other than the standard. A special version of %RTX with $USAV/$URES calls must be used rather than the normal version of %RTX. The tape of %RTX with $USAV/$URES calls is numbered 70-43-022R-A and is four locations longer than the normal version of %RTX.

$USAV

This will be called during the saving of registers. The AX register will contain the value of the SC at the time of interrupt. %RTX expects $USAV to save additional registers and return without destroying the AX register. An example might be, for the 6 GPR option (or a Model 40):

```
        ENTRY       $USAV
$USAV:  RMD         30, $SAVA;SAVE GPR's
        RMD         31,$SAVA
        RMD         32,$SAVA
        RMD         33,$SAVA
        RMD         34,$SAVA
        RMD         35,$SAVA
        RR          TRP,SC;RETURN
        END
```

$URES

This wil be called during the restoring of registers. Everything except the users registers and the SC will be restored before control is given to $URES. %RTX expects this routine to restore the user's registers without destroying any of AX, AY, MSR or TRP, and then $URES must restore the SC. To continue the example, the $URES that corresponds to the $USAV

above could be:

```
            ENTRY       $URES
$URES:      MRD         $REST, 30       ;RESTORE GPR's
            MRD         $REST, 31
            MRD         $REST, 32
            MRD         $REST, 33
            MRD         $REST, 34
            MRD         $REST, 35
            FOI         ICO             ;THESE RESTORE
            MRD         $REST,SC        ;THE SC
            END
```

Note: If $SREL, $AHGH or $ALOW are to be included in the system, the special versions (which call $USAV,$URES) of these routines must be used. The special version of $SREL is numbered 70-43-023R-A and is two locations longer than the normal. The special version of $AHGH/ $ALOW is numbered 70-43-024R-A and is four locations longer than the normal.

ATTACH

Length: 77_8 (63_{10})

Entry Points: $ATCH

Function:

This routine is used to eliminate conflicts in the usage of shared subroutines in the system. It does this without requiring the subroutine to be written differently from a subroutine written for non real-time use. If $ATCH and $SREL are used, for instance, reentrant code is unnecessary.

Usage:

For each subroutine which is to be shared between processors in the system, the user should assign a single core location initially set to zero. This location will be used as a flag to indicate whether the associated subroutine has been "attached" or not. The flag is examined and set by $ATCH.

Before a processor in the system calls a shared subroutine, it should first "attach" it by calling $ATCH and giving as an argument the address of the flag or flags associated with the subroutine(s) it wishes to attach. The general format of the call is:

```
JU          $ATCH
WRD         FLAG1     ;ADDRESS OF SUBR1 FLAG
WRD         FLAG2     ;ETC
 :
 :
WRD         FLAGn+100000
```

whereas the last or, in most cases, the only argument has Bit 15 set (by adding 100000_8 to the address).

When control is returned to the processor which called $ATCH, each FLAG listed in the call sequence will be set to the address -1 of the processor's prologue -1 entry in the processor list to identify the processor who has attached the subroutine(s). In addition, the processor will be set active (i.e. $ATCH could possibly be called from an inactive section, but upon return the caller will be active).

The processor is now free to call the subroutines whose flags were listed in the call to the $ATCH. To release the subroutine for use by other processors it is only necessary to zero the flag word associated with that subroutine, and the ZM FLAG instruction must be followed by a JU $SREL which informs **%RTX** that any higher priority processor which may have tried to attach the subroutine can now do so.

The scope of an attach consists of all code within a processor from the call to the $ATCH routine to the point at which either the last of the flags listed in the $ATCH call has been zeroed followed by a JU $SREL, or an exit has been made to %RTX ($NREL or $EXEC). (Usually all flags should be zeroed before such an exit, although it is possible to structure things so that a subroutine is attached to the same processor for several scans by %RTX.)

Great care must be taken if $ATCH is called within the scope of an attach. For example, to attach more than one subroutine, the following procedures are okay:

```
SCOPE
        JU      $ATCH
       -WRD     FLAG1
        WRD     FLAG2 + 1ØØØØØ
         .
         .
         .
        JU      SUBR1           ;NEED SUBR1 TWICE
         .
         .
         .
        JU      SUBR2           ;NEED SUBR2 ONCE
        ZM      FLAG2           ;SIGNAL DONE SUBR2
        JU      $SREL
         .
         .
         .
        JU      SUBR1           ;2nd TIME SUBR1
        ZM      FLAG1           ;THEN DONE
        JU      $SREL
         .
         .
         .
```

The scope of the attach extends from the JU $ATCH to the JU $SREL following the ZM FLAG1 and is bracketed above. (Note that the JU $SREL following the ZM FLAG2 does not delimit a scope.)

Another way:

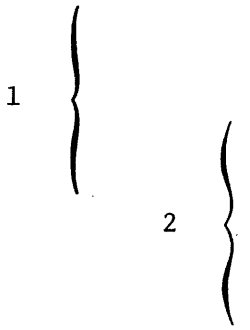
```
SCOPES
        JU      $ATCH
        WRD     FLAG1 + 1ØØØØØ
         .
         .
         .
        JU      SUBR1
        ZM      FLAG1
        JU      $SREL
         .
         .
         .
        JU      $ATCH
        WRD     FLAG2 + 1ØØØØØ
         .
         .
         .
        JU      SUBR2
        ZM      FLAG2
        JU      $SREL
         .
         .
         .
        JU      $ATCH
        WRD     FLAG1 + 1ØØØØØ
         .
         .
         .
        JU      SUBR1
        ZM      FLAG1
        JU      $SREL
```

The scopes are marked.

However, the following could cause a disaster unless the two rules stated below are complied with.

```
1        JU      $ATCH
         WRD     FLAG1+ 1ØØØØØ
          .
          .
          .
         JU      SUBR1
          .
          .
          .
    2    JU      $ATCH
         WRD     FLAG2 + 1ØØØØØ
          .
          .
          .
         JU      SUBR2
          .
          .
          .
         ZM      FLAG2
         JU      $SREL
          .
          .
          .
         ZM      FLAG1
         JU      $SREL
```

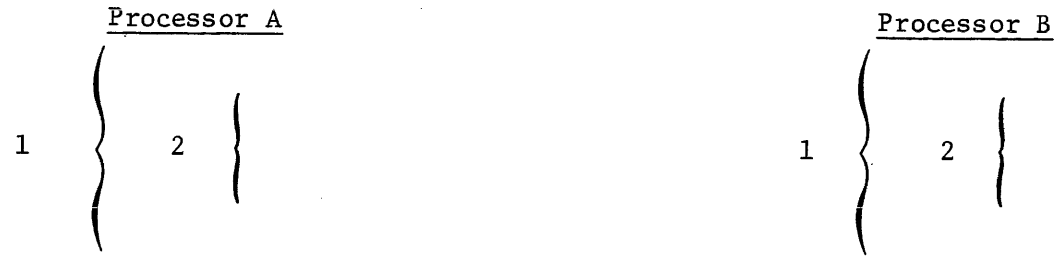
1. Scopes must be fully nested, i.e. the following overlapping of scopes will cause a disaster:



2. There must not exist two processors in the system with nested scopes in reversed sense of each other. ie., the following will cause a disaster if Processors A and B are in the same system:

Processor A		Processor B	
JU	$ATCH	JU	$ATCH
WRD	FLAG1 + 1ØØØØØ	WRD	FLAG2 + 1ØØØØØ
⋮		⋮	
JU	$ATCH	JU	$ATCH
WRD	FLAG2 + 1ØØØØØ	WRD	FLAG1 + 1ØØØØØ
⋮		⋮	
ZM	FLAG2	ZM	FLAG1
JU	$SREL	JU	$SREL
⋮		⋮	
ZM	FLAG1	ZM	FLAG2
JU	$SREL	JU	$SREL

Note that Processor A has the scope of the attach for FLAG2 nested within the scope of the attach for FLAG1 whereas in Processor B it is the other way around. The following would be okay, however:



Description of Algorithm

If $ATCH is called and one of the flags in the argument list is non-zero (attached to somebody else), the processor which called $ATCH is set inactive in such a way that the next time it is invoked by %RTX it will resume inside the $ATCH routine to examine the particular flag again. $ATCH then uses the information in the non-zero flag to call $RSME to force the processor to which the subroutine was attached to continue operating. Should the latter zero the flag and call $SREL, things have been set up so that $ATCH is immediately re-entered. Since the flag is now zero $ATCH attaches the subroutine to the original caller, and, if the last flag in the argument list has been processed, sets the caller active and returns.

Note that the processor which "finished" the subroutine had control only up to the call to $SREL at which point the original caller to $ATCH regained control.

By this means, the subroutine is given to the highest priority processor which is trying to attach it at any particular time.

Notes

1) It is okay to have inactive code in the scope of an attach. (But see note 5.)

2) $ATCH cannot be used by Interrupt Service Routines.

3) Since $ATCH may set the caller inactive, and in this state no registers are saved in case of interrupt, it may be safely said that $ATCH has the possibility of destroying the contents of every register. However, when $ATCH returns, the following machine conditions are guaranteed:

a) AO is in the ADD state.
b) AX = contents of $SCAN.
c) AY = -2
d) LNK is set.
e) BOV is clear.

In addition, the caller will be active and all flags in the argument list will be non-zero (attached).

4) $ATCH can be used to attach anything. For instance, one processor may wish to alter a table and keep other processors from using the table until it is done. Or, on the Model 40, instead of taking the time to save and restore the 6 GPR at every interrupt, the processors which use them can attach them either singly or as a group. $ATCH is perfect for such applications.

5) $ATCH may be used in a system which submits more than one processor list in the course of its operation provided:

a) All elements of the system are permanently in core.
b) The scope of any attach which references a flag which is also attached by a task in another processor list must be free of any inactive code or returns to %RTX.

SUBROUTINE RELEASE

Length: 24_8 (20_{10})

Entry Points: $SREL

Function:

Allows higher priority processor to attach and use subroutine just released.

Usage: (See $ATCH)

Call $SREL immediately after zeroing an attached flag. This allows the released flag to be attached by any processor waiting for it (if any). No information other than the trap register is destroyed by this call. I.e., $SREL acts as if it were an interrupt and saves all registers (and goes to $USAV if incorporated) so that %RTX restores everything, returning just following the call to $SREL.

Description of Algorithm:

$SREL saves all registers in the callers save area, decrements $SCAN by two and jumps to $NEXT. This has the net effect of either restoring the caller immediately, or resuming in the $ATCH routine should it be waiting on the flag just zeroed before the JU $SREL.

Notes

1) Do not call $SREL from an inactive processor.

2) See note to write up for $USAV, $URES.

GET MEMORY/PUT MEMORY

Length: 454_8 (300_{10})

Entry Points: $GETM, $PUTM, $PUTP

Function: Dynamic Core Allocation

Subroutine Called: %RTX, $SREL

Usage:

Should the real-time system be such that the demands for memory which need to be assigned to tasks are unpredictable, these routines can be used to assign and release areas of core storage.

$GETM - Get Memory

This is called with one argument stating the number of storage locations desired, e.g.

```
        JU    $GETM          ;GET 300 WORDS
        WRD    300           ;OF FREE CORE
```

Upon return the address of the first location of the requested block of core will be in the AX register. The free area will consist of exactly the number of words requested and no more. If a block of core of the size requested is not available at the time it is requested, the processor which called $GETM will be placed in a "wait" state until that memory is available. In this state the processor is inactive and, essentially, removed from the system. The net result is that a call to $GETM may allow all other proces-

sors to operate before control is returned. Also, since it can become inactive there is the possibility that all registers in the system will be altered. The only conditions guaranteed on return are:

1) AX = address of first location of free core area of size requested.
2) AO in ADD state.
3) LNK is clear.

$PUTM - Put Memory

This is called with the address of the first word of the free area in register AX. This value must be equal to a value returned by some previous call to $GETM. Upon return the core area will no longer be assigned, i.e. it is available to $GETM to assign to the next request. Since this routine can become inactive waiting for $GETM to finish operating, it is possible for all registers in the system to be altered on return. The only condition guaranteed on return is that the AO is in the ADD state.

$PUTP - Put Partial Memory

This is called to make a partial area of core available to $GETM. For instance, a processor might call $GETM to read in a block of data from a device. It is known that this data could occupy up to 200_8 words of core, so the call to $GETM requests 200 words. When the data has been read in, the processor discovers that it consisted of only 50_8 words. If desired, the last 130_8 words can be released for $GETM to assign elsewhere by calling $PUTP. For this call, AX must be set at the address of the first word of the complete block (i.e. the value originally returned by $GETM) and AY is set to the address of the first word of the partial area to be put back.

For example, suppose it is desired to read in a data block which might be 500_8 words long. A processor could proceed as follows:

```
            JU      $GETM           GET 500 WORDS
            WRD     500
            RMI     AX,Ø            ;SAVE START OF
STBFR =     .-1                     ;BUFFER ADDRESS.
            .
            .
            .
```

Suppose after reading, the address of the last data word stored is in 'LAST'. The following will allow the area from the word following the last data to the end of the original buffer to be made available to $GETM:

```
            .
            .
            .
            MR      LAST, P1, AY    ;ONE BEYOND LAST TO AY
            MR      STBFR, AX       ;START OF BUFFER
            JU      $PUTP
            .
            .
            .
```

And finally when the processor is completely done with that memory area it calls $PUTM with the original address to release the rest of the buffer, i.e.

```
            .
            .
            .
            MR      STBFR, AX       ;PUT BACK ASSIGNED
            JU      $PUTM           ;MEMORY
```

Of course, the intermediate steps of putting back the partial area could be omitted if core space is not that much of a premium.

Notes

1) These routines can be called from an inactive area. However, the caller will be active upon return.

2) Free core is originally set by a call to $SETM (see its write-up) during initialization.

SET MEMORY

Length: 30_8 (24_{10})

Entry Points: $SETM

Function: To initialize pointers in memory for $GETM and $PUTM.

Calling Sequence: AX = address of first free core location.
AY = address of last free core location.

```
JU  $SETM
```

Usage:

This routine should be called during system initialization (start-up) to define free core to $GETM and $PUTM if dynamic core allocation is to be used.

Notes: (See also $GETM, $PUTM)

1) If $SETM is loaded last, it can reside in the free core area. In this case it will be destroyed during the operation of the system, but usually this does not matter since $SETM cannot be called at any other time except initialization/start-up.

2) Free core must be one contiguous block (usually from the end of the programs to the beginning of the resident loaders in high core).

3) $SETM does not zero the free core area, it only intializes pointers.

ALLOW HIGH/LOW

Length: 31_8 (25_{10})

Entry Points: $AHGH, $ALOW

Function: Allows all higher priority ($AHGH)or all lower and higher priority ($ALOW) processors to be invoked before resuming.

Usage:

If it is known that a processor in the system is going to take a considerable amount of time to complete its task, it can prevent other processors from being excessively "locked-out" by calling $AHGH or $ALOW. The calling sequence has no arguments. Both routines save and restore all registers with operation resuming at the return point. $AHGH and $ALOW may be thought of as a programmed interrupt. $AHGH forces the scan pointer to the top of the priority list and returns to %RTX, thereby beginning the scan on all higher priority processors. $ALOW bumps the scan pointer to the next lower priority (if any) processor in the list and returns to %RTX, thereby beginning the scan on all lower priority processors, followed by a scan on all higher priority processors. In either case. %RTX eventually re-invokes the processor which called $AHGH or $ALOW, causing registers to be restored and processing to resume following the point of call.

Notes

1) Do not call these from an inactive area. Instead, call $NEXT (analogous to $ALOW) or $STRT (analogous to $AHGH).

2) See notes to write-up of $USAV, $URES.

WAIT

Length: 54_8 (44_{10})

Entry Points: $WAIT

Function: To wait for a dynamic condition without locking up the system in a tight loop.

Calling Sequence:

```
Set     AY
JU      $WAIT
WRD     ARG1        ;CONDITION
WRD     ARG2        ;ADDRESS OF FLAG
```

Usage:

If a processor needs a certain condition fulfilled before it can proceed with its operations, it can use $WAIT to avoid locking up the system in a tight loop. For instance, suppose the condition is the completion of input into a buffer, and that this is signaled by the Interrupt Service Routine setting "IDONE" to a value greater than Ø. The following could be done, but is not advisable:

```
:
:
MR          IDONE, AX
JC          AX, LEZ, .-2
```

because the entire system is locked out while this processor is in the two instruction loop waiting for IDONE to become greater than zero.

The calling sequence to $WAIT consists of setting AY, the jump to $WAIT, followed by two arguments. The first argument must be a data test on the A0. I.e. it must be (in instruction format) of the form 13 XXXØ Ø3; where XXX specifies the test. The second word is the address of the flag whose condition

is being tested. The easiest way to specify the arguments to $WAIT is to follow the JU $WAIT with a JC on the A0. E.g. for the example above:

```
ZR   AY                    ;SET AY = Ø
JU   $WAIT
JC   A0, GTZ, IDONE        ;SET UP BOTH ARGUMENTS
:
:
```

Note that the JC A0, GTZ, IDONE is never executed; it only serves to establish the arguments - word 1 is the data test on A0, word 2 is the address "IDONE".

Description:

What $WAIT does is to set the A0 to the ADD state, load the contents of IDONE into AX, (the caller having set the contents of AY) and test the A0 for the condition specified in the calling sequence. If the condition is true (successful), the caller is set active, and $WAIT returns following the two arguments. If the condition is not true, the calling processor is set inactive in such a way that the next time it is invoked it reenters the $WAIT routine to repeat the test. Thus the caller is temporarily removed from the system.

Notes

1) Since the caller may be set inactive, no registers are saved. Thus, a call to $WAIT could result in all registers being altered. The only conditions guaranteed on return are:

a) AY is same as before call to $WAIT.
b) A0 is in ADD state.
c) LNK is clear.
d) BOV is clear.

(Although it seems that a guaranteed condition should also be: AX is equal to the contents of the flag tested, it is not the case. Most of the time it will be---but in some systems it is possible for the flag to be altered between the time it was tested and the time return is made from $WAIT.)

2) $WAIT can be called from an inactive area. However, the caller will be active upon return.

ENQUEUE, DEQUEUE

Length: 71_8 (57_{10})

Entry Points: $ENQ, $ENQF, $DEQ, $DEQF

Function: First-in, first-out list using linking pointers.

Usage:

The primary use of these routines is to facilitate the passing of large blocks of data from one system component to another without having to physically move the data from one area of core to another. This scheme also allows data to "back-up" without being lost if a real-time system should become temporarily overloaded.

When a part or all of a system uses this approach to handle data, the components must reference so-called "queues" to find out where the data is located. A particular queue consists of two words at a known place in memory. The first word of the queue contains the address of the first word of the first item (e.g. block of data) on the queue. The first word of the first item (link word) contains the address of the first word (link word) of the second item on the queue, the first word of the second item contains the address of the first word of the third item, etc. The first word of the last item on the queue is set to zero to identify it as being last. The second word of the queue contains the address of the first word of the last item on the queue. Figure 4 offers a pictoral representation of the enqueuing process. An empty queue is identified by the first word of the queue being zero -- and in this case the second word must contain the address of the first word. Figure 4 also illustrates an empty queue.

$ENQ (or $ENQF) is provided to enter a new item as the last item on a queue, and $DEQ (or DEQF) is provided to remove the first item from a queue. Both of these routines involve changing link words only, they do not move any data from one place to another.

Calling Sequence - $ENQ, $ENQF

Load AX with address of link word of new item.

```
JU      $ENQ (or JU $ENQF)
WRD     QADDR              ;ADDRESS OF FIRST WORD OF 2 WORD QUEUE
:
```

This adds the item whose link word address is in AX to the end of the queue at QADDR. $ENQ returns with interrupt control on, $ENQF returns with interrupt control off (FOI ICF). Neither AX nor AY is changed by $ENQ or $ENQF.

Calling Sequence - $DEQ, $DEQF

```
JU      $DEQ (or JU $DEQF)
WRD     QADDR              ;ADDRESS OF FIRST WORD OF 2 WORD QUEUE
:
```

$DEQ (or $DEQF) returns after deleting the top (first) item from the queue at QADDR. This brings to the top a new item whose link address is in both QADDR and AY. The address of the deleted item's first (link) word is in AX. This is convenient for an immediate call to $ENQ (or $ENQF) to enqueue the same item on to another queue to pass data to another component in the system. $DEQ returns with interrupt control on, $DEQF returns with interrupt control off. Figure 5 shows the dequeuing process.

Notes

1) Normally $ENQ and $DEQ are the desirable calls. However, in some cases, to assure that information is not lost (e.g. dequeuing within an inactive area) the "return with ICF" versions should be used. An example, which also illustrates "rotating a queue" might be as follows:

Suppose one of the processors in the system is dedicated to handling all teletype input and that there are several teletypes on the system. Each teletype has been assigned an input buffer by some processor (say via a $GETM) and all the buffers are linked onto the queue "TTYIB" (teletype input buffers). Also suppose that the word following the link word in each buffer is a flag which is set to zero when the input is complete (e.g. the Interrupt Service Routine might set this flag when it detects a carriage return). The following prologue (inactive) for this processor will, at each scan by %RTX, examine the flag in the top item on the queue, remove the item from the top of the queue and replace it at the bottom. (The next %RTX scan will thus examine a new flag if there is more than one buffer on the queue).

```
PROL:    MR   TTYIB, AX          ;ADDR. OF TOP BUFFER LINK TO AX.
         JC   AX, ETZ, $NEXT     ;IF NO ITEMS ON QUEUE.
         RM   AX, P1, .+3        ;ADDR. OF FLAG TO NXT INSTR.
         MR   Ø, AY              ;CONTENTS OF FLAG TO AY.
         JC   AY, ETZ, ENTER     ;FLAG IS Ø, SO GO ENTER.
         JU   $DEQF              ;DEQUEUE TOP BUFFER
         WRD  TTYIB              ;BRINGING UP NEXT BUFFER.
         JU   $ENQ               ;PUT TOP BACK ON BOTTOM
         WRD  TTYIB              ;THEREBY ROTATING
         JU   $NEXT
ENTER:   ZM   PSAV               ;SET SELF ACTIVE
         :
         :
```

Note that the call to $DEQF (rather than $DEQ) is absolutely essential. This is because if $DEQ is called an interrupt could occur between the call to dequeue and the call to $ENQ, and since the prologue is inactive %RTX would re-enter at "PROL" so that the item dequeued before the interrupt will never be enqueued back onto TTYIB (or anywhere else) and is lost to the system.

2) On return from $DEQ, AY contains the same new value which is in the first word of the queue so that if AY is Ø, the queue is empty. If AX is also Ø, the queue was empty before $DEQ was called.

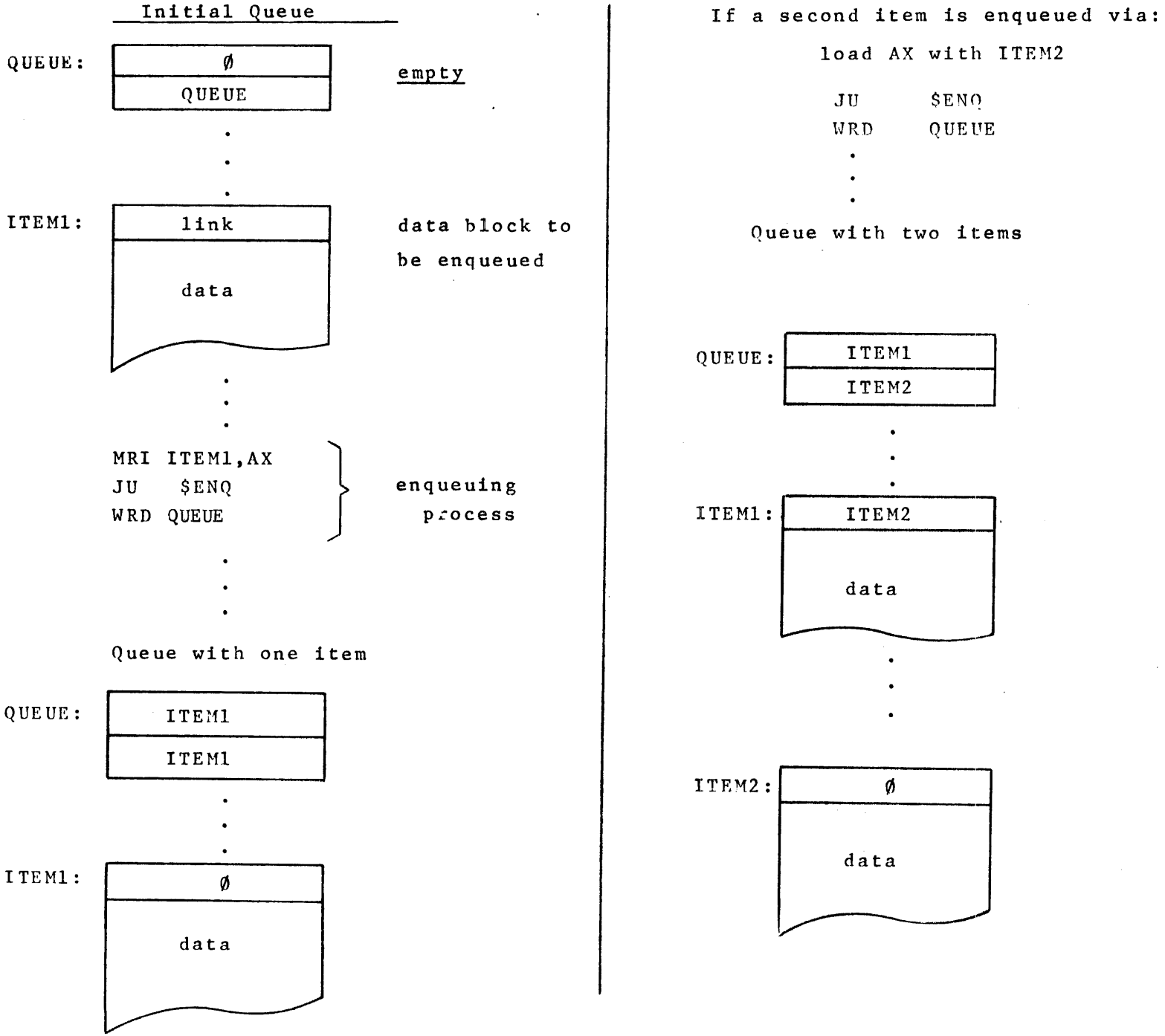


FIG. 4 (Enqueuing)

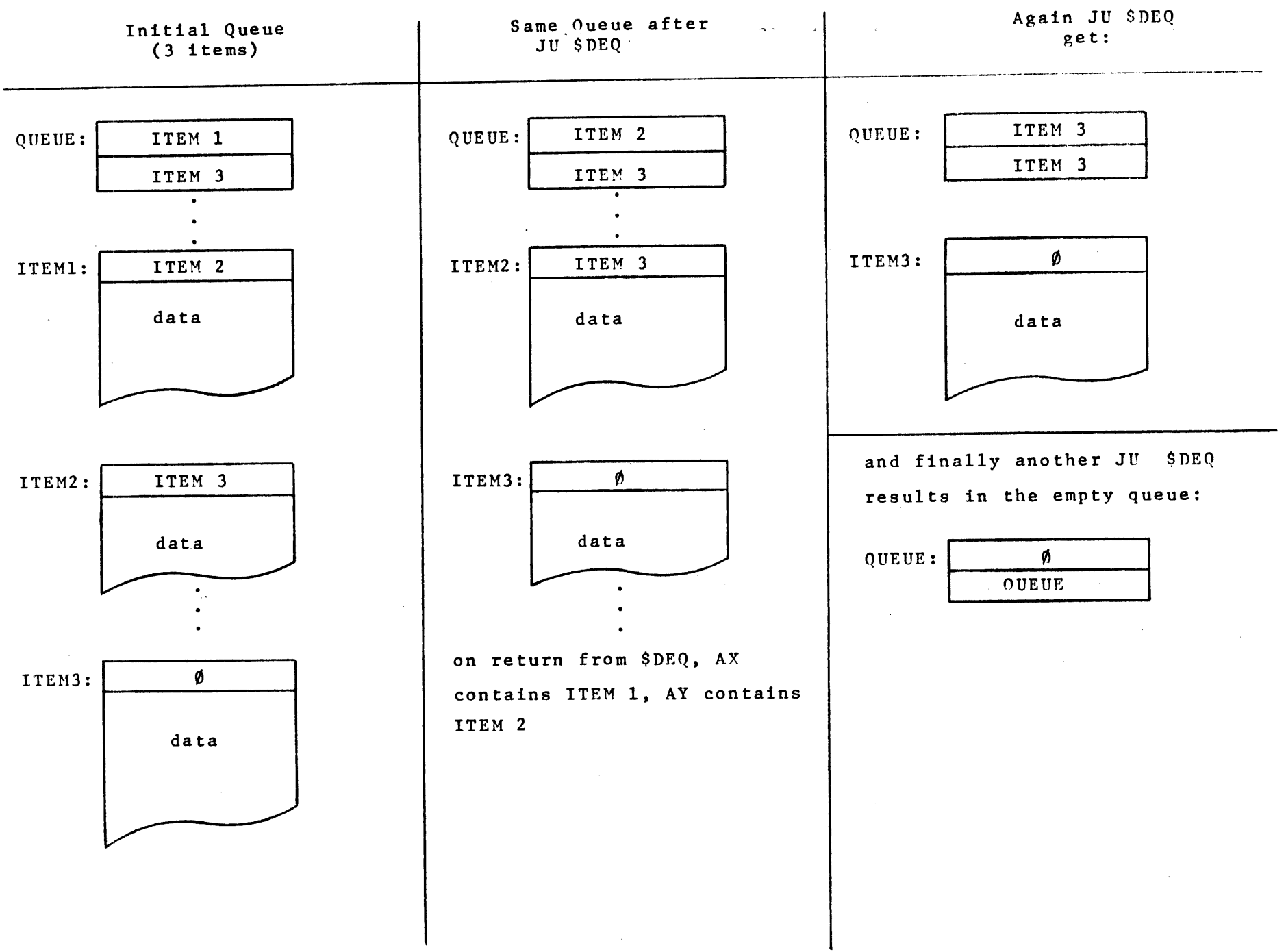


FIG. 5 (Dequeuing)

CHAPTER THREE

STANDARD I/O SERVICE ROUTINES

INTRODUCTION

A set of interrupt service routines are provided to handle interrupts from the teletype and the high speed reader/punch. Before an I/O device can cause an interrupt, the device must be started. Therefore, a set of device starter subroutines are provided in conjunction with the interrupt service routines. The starters and interrupt service routines are linked by information found in a control block. The control block consists primarily of queues containing buffers to be used for input and output and the addresses of the next locations from which data is to be output and into which data is to be stored. Each buffer on an input or output queue must contain a negative count of the number of words in the buffer.

Generally, output works as follows. The desired output device starter is called. The starter uses the top buffer on the output queue to determine where to start the output from. It then outputs the first character, sets a bit in the ISR to allow the output device to interrupt and returns. Then each time the output device causes an interrupt, the interrupt service routine bumps the count in the output buffer. If the count is not yet zero, the next character is output and a non-end-mode return is taken. When the count finally becomes zero, the buffer is removed from the output queue and an end-mode return is taken. If another output buffer is on the output queue, the output starter is called again before the end-mode return is taken.

For input, the appropriate input starter is called. The starter uses the top buffer on the input queue to determine where the first character is to be stored. It then sets a bit in the ISR to allow the input device to interrupt, starts the input device and returns. Each time the input device interrupts , a user subroutine (specified in the control block) is called. This user subroutine allows the user to test for any special end-mode conditions (for example

a carriage return or end of message character). When this subroutine returns the character is stored in the next location of the input buffer, the count is bumped, and if it is non-zero the device is restarted and a non-end-mode return is taken. If the count is zero it is reset to the actual number of characters stored in the buffer, the buffer is removed from the input queue and an end-mode return is taken. If another buffer is on the input queue, the input starter is called again before the end-mode return is taken. Note that the end-mode condition is signaled by a count equal to zero. Thus, if the user routine discovers the special end mode condition to be true, he must set the count to -1 so when it is bumped it will become zero and thereby indicate an end-mode condition.

In the discussion which follows TTY refers to the teletype, TTI refers to the teletype input, TTO to the teletype output, HSR to the high speed reader, and HSP to the high speed punch. The term "multiple devices" means two or more hardware devices of the same type but with different device addresses. For example in a configuration consisting of three teletypes with addresses 77, 67, and 57 respectively the teletypes are referred to as a set of multiple devices. They all interrupt to the same location and use the same bits in the ISR, but the teletypes can be distinguished from each other since each has its own unique address and its own ready flags.

$SRET STARTER RETURN

Length: 13_8 (11_{10}) locations

This routine contains the common return used by all the standard device starters. Therefore, if any of the standard device starters are used, $SRET must be loaded. If the user writes additional device starter subroutines, $SRET could be called to return from the subroutine.

Entry Points:

$SRET Common Return from Device Starter

This routine is called by all the standard device starter subroutines to return to the calling program. It sets up $ISR and ISR to allow the started device to interrupt. When $SRET is called register AY must contain the bit to be OR'd into $ISR and ISR. The MSR is zeroed, interrupt control is turned on and control is returned to the program calling the device starter subroutine.

$SRT1 Return Address for $SRET

This location contains the return address-1 for the program calling the device starter. This location is used by $SRET to return to the program which called the device starter subroutine. Hence a device starter using this routine should begin by storing the TRP register into $SRT1.

$SAVI Save Routine For ICF Interrupts

Length: 36_8 (40_{10}) locations

This subroutine is used to save and restore registers when an interrupt occurs. Its entries are called by the standard ICF-type interrupt acknowledge routines. Therefore, if any ICF-type interrupt routines are being used, $SAVI must be loaded. If additional ICF-type interrupt routines are written by the user, $SAVE may prove useful.

Entry Points:

$SAVI Save Registers

This subroutine is called by all the standard ICF-type interrupt acknowledge routines (after the TRP has been stored in $TRP) to save registers ISR, AX, AY, and MSR when an interrupt occurs. $SAVI then zeroes the ISR and issues a FOI ICO to allow power fail interrupts only, zeroes the MSR to ensure that the AO is in the ADD state and returns. $SAVI has no arguments.

$EMR Restore Registers, End-Mode Return

This routine is called by the standard ICF-type interrupt service routines. It turns interrupt control off, restores the registers saved by $SAVI and takes an ICF-type end-mode return to %RTX. $EMR has no arguments.

$NEMR Restore Register, Non-End-Mode Return

This routine is the same as $EMR except an ICF-type non-end-mode return to %RTX is taken.

$TICF Teletype Interrupt Acknowledge - ICF

Length: Absolute Locations 11_8 - 16_8 plus 30_8 (24_{10}) relocatable locations.

This tape contains the ICF-type interrupt acknowledge routines for the teletype. Absolute locations 11_8 - 13_8 are used to save the SC when TTI causes an interrupt and to transfer control to the TTI interrupt acknowledge routine. Absolute locations $14\text{-}16_8$ are used to save the SC when TTO interrupts and to transfer control to the TTO interrupt acknowledge routine.

Both the TTI and TTO interrupt acknowledge routines store the TRP in $TRP, call $SAVI to save additional registers, store the address-1 where the SC was stored in $INTL and then check the ready flag for TTY77. If TTY77 caused the interrupt, control is transferred to the ICF-type TTY77 interrupt service routine. Otherwise, $TICF halts. If the user adds additional teletypes to the system, $TICF must be edited and reassembled to check the ready flags for the additional teletypes. For this reason $TICF is provided as both a source tape and relocatable object tape.

Entry Points:

$TICF ICF-type TTY77 Interrupt Acknowledge

This is a dummy entry entry point since no other standard interrupt routine references any symbol defined in $TICF. It is declared as an entry point simply so the user will know after loading where the interrupt acknowledge routines have been loaded.

TTY77 Interrupt Handler - ICF ($CB77)

This tape contains the control block, device starter subroutines and interrupt service routines for ICF-type handling of TTY77. If another Teletype is added to the system it must have a unique address other than 77. An interrupt handler similar to $CB77 must also be created to handle the additional Teletype. To do this, the source tape for $CB77 should be copied using %STE and the exchange command used to change every occurence of 77 to the address of the additional Teletype. This new source tape should then be assembled and the resulting object tape loaded to handle the additional Teletype.

For example, suppose another Teletype with address 67 were added to the system. Then $CB77 would be copied and all occurences of 77 changed to 67. The name of the control block for TTY67 would thus become $CB67, the name of the reader starter for TTY67 would become $RS67 and so on.

In the discussion which follows all references to the Teletype are to TTY77.

Length: 427_8 locations

Entry Points Detailed descriptions follow

Name	Function
$CB77	address of control block associated with Teletype 77.
$SF77	reader stop flag, can be set up by the user to stop reader input.
$RN77	for internal use by Teletype handler, contains address-1 to store next reader character.

Name	Function
$RQ77	reader queue, contains pointers to reader input buffers, the user must enqueue reader buffers onto $RQ77.
$RU77	contains address of user subroutine to modify reader data and/or check for special reader end-mode conditions.
$KN77	for internal use by the Teletype handler, contains address-1 to store next keyboard character.
$KQ77	keyboard queue, contains pointers to keyboard input buffers, the user must enqueue keyboard buffers onto $KQ77.
$KU77	contains address of user subroutine to modify keyboard data and/or check for special keyboard end-mode conditions.
$PN77	for internal use by Teletype handler, contains address-1 of next character to output in priority mode.
$PQ77	priority output queue, contains pointers to priority output buffers, the user must enqueue priority output buffers onto $PQ77.
$NN77	for internal use by Teletype handler, contains address-1 of next character to output in normal mode.
$NQ77	normal output queue, contains pointers to normal output buffers, the user must enqueue normal output buffers on $NQ77.
$EC77	echo buffer, used by the standard user subroutine $ECHO to echo Teletype input.
$RS77	reader starter, subroutine called by the user to start reader input.
$KS77	keyboard starter, subroutine called by the user to start keyboard input.

Name	Function
$PS77	priority output starter, subroutine called by the user to start priority output to TTO.
$NS77	normal output starter, subroutine called by the user to start normal output to TTO.
$IP77	interrupt service routine to process TTI interupts.
$OP77	interrupt service routine to process TTO interrupts.
$IL77	return address for user subroutine specified in $RU77 or $KU77, this return adds one to the count in the input buffer, stores the character in the input buffer, then checks the count for equal to zero.
$ID77	return address for user subroutine specified in $RU77 or $KU77, this return stores the character and checks (but does not add one to) the count.
$IE77	return address for user subroutine specified in $RU77 or $KU77, this return checks the count only, it neither adds one to the count nor stores the character.

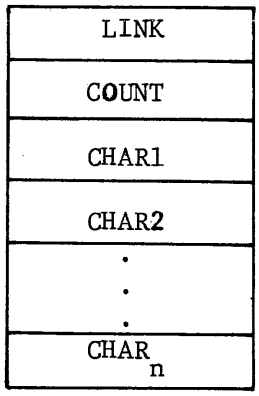
The Teletype routines provided are buffer oriented. That is, input buffers are filled with characters from TTI and characters from the output buffers are sent to TTO. These input buffers and output buffers must be enqueued by the user onto the input queues and output queues found in the Teletype control block. The control block consists of the following information, each label in the control block is an entry point.

```
$SF77:      WRD Ø                ;READER STOP FLAG
$RN77:      WRD Ø                ;READER NEXT LOC
$RQ77:      WRD Ø,.-1            ;READER QUEUE
$RU77:      WRD $IL77            ;USER'S READER SUBROUTINE
$KN77:      WRD Ø                ;KEYBOARD NEXT LOC
$KQ77:      WRD Ø,.-1            ;KEYBOARD QUEUE
$KU77:      WRD $IL77            ;USER'S KEYBOARD SUBROUTINE
$PN77:      WRD Ø                ;PRIORITY OUTPUT NEXT LOC
$PQ77:      WRD Ø,.-1            ;PRIORITY OUTPUT QUEUE
$NN77:      WRD Ø                ;NORMAL OUTPUT NEXT LOC
$NQ77:      WRD Ø,.-1            ;NORMAL OUTPUT QUEUE
$EC77:      WRD Ø,Ø,Ø,Ø          ;ECHO BUFFER
```

Teletype Input - ICF

Teletype input may be from either the reader or the keyboard. Therefore the Teletype control block has two input queues, one called $RQ77 for reader input buffers and the other called $KQ77 for keyboard input buffers. Input buffers (both reader and keyboard) must conform to the following format:

Input Buffer



The first word is reserved for enqueing the buffer onto the input queue. The second word must contain a negative count of the maximum number of char-

acters to be stored. Characters will be stored starting in the third word of the input buffer.

There are two Teletype input starter subroutines, one called $RS77 to start reader input and the other called $KS77 to start keyboard input. Both $RS77 and $KS77 are entry points.

$RS77

To start input from the Teletype reader, the user should first enqueue one or more reader buffers onto the reader queue, then call the reader starter by a JU $RS77. When $RS77 returns, interrupt control will be on, reader input will have been intitiated and the ISR set up to allow TTI interrupts. Then each time TTI interrupts, a character is stored in top buffer on the reader queue. Note that $RS77 simply initiates reader input, the TTI interrupt service routine actually fills the reader buffer.

$KS77

To start keyboard input the user should enqueue one or more keyboard buffers onto the keyboard queue and then call the keyboard starter by a JU $KS77. When $KS77 returns, interrupt control will be on, keyboard input will have been initiated and the ISR set up to allow TTI interrupts. Then each time TTI interrupts, a character is stored in the top buffer on the keyboard queue.

Since there is no way for the interrupt service routine to know whether a TTI interrupt was caused by the reader or by the keyboard, once a Teletype input starter is called all TTI interrupts are assumed to be of the source implicit in the starter. That is, if $RS77 was called, all TTI interrupts are assumed to be from the reader and if $KS77 was called, TTI interrupts are assumed to be from the keyboard.

$IP77

When TTI causes an interrupt, control is transferred to $IP77, the TTI interrupt service routine, for processing. If the reader starter was called to initiate input $IP77 considers the character to be from the reader, the input queue to be the reader queue and the input buffer to be the top buffer on the reader queue. If the keyboard starter was called to initiate input, $IP77 considers the character to be from the keyboard, the input queue to be the keyboard queue and the input buffer to be the top buffer on the keyboard queue.

Each time $IP77 is entered, it stores a character in the input buffer and adds one to the count in the input buffer. When the buffer becomes full it indicates an end-mode condition. The count is set to plus the number of characters stored, the buffer is dequeued from the input queue and an end-mode return is taken. The new buffer now on top of the input queue becomes the new input buffer. Thus all the buffers on the input queue are filled and then dequeued until finally the input queue becomes empty. When this happens, input is through and the user must call an input starter before input will begin again. An exception is during reader input, when the reader queue finally becomes empty $IP77 will call the keyboard starter if the keyboard queue is not empty.

$SF77

$SF77 is a flag in the Teletype control block which can be set by the user to stop reader input before the reader queue becomes empty. If the user sets $SF77 to one, reader input is stopped after the next TTI interrupt and $SF77 is reset to zero. In addition, if the keyboard queue is not empty keyboard input is automatically started. Once reader input has been stopped

in such a manner it can only be restarted by the user jumping to $RS77, at which time reader input will continue from where it left off.

User Input Subroutines

Before storing a character in the input buffer, the interrupt service routine $IP77 calls a user subroutine specified in the control block. If the character is from the reader, the user subroutine specified in $RU77 is called. If the character is from the keyboard, the user subroutine specified in $KU77 is called.

The user subroutine can modify the data before storing it and also check for special end-mode conditions. Three general user subroutines ($ASCI, $ECHO, and $LINE) are provided. $ASCI ignores (that is, does not allow the service routine to store) zero characters and OR's in bit 7 of non-zero characters thus ensuring 8-bit ASCII. Thus, if the user wishes to input ASCII characters from the reader, $RU77 should contain $ASCI. $ECHO performs the same function as $ASCI but in addition echoes the input character. Thus, if the user wishes keyboard input to be echoed, $KU77 should contain $ECHO. $LINE is used to store a line of ASCII text in the input buffer. It recognizes the special characters back arrow and rubout to mean respectively, ignore previous character and ignore the entire line. Also, if a carriage return is encountered an end-mode condition is forced even if the input buffer is not yet full.

$IL77, $ID77, $IE77

If the user wishes to modify data or check for special end-mode conditions not covered in $ASCI, $ECHO or $LINE, he may write his own special user subroutines. When the user subroutine is called AY will contain the TTI character and AX will contain the address of the input queue. When the user

subroutine has completed its task it should return to one of three places in the TTI interrupt service routine. A return to $IL77 will add one to the count in the input buffer, store the character (assumed to still be in AY) then check the count to see if it is zero indicating the input buffer is full. A return to $ID77 stores the data and checks, but does not bump, the count. A return to $IE77 neither bumps the count nor stores the data, it simply checks the count to see if it is zero. The three returns $IL77, $ID77 and $IE77 are all entry points. When the user subroutine is called, $IL77 = TRP+1, $ID77 = TRP+3 and $IE77 = TRP+5.

Therefore, to ignore a character the user subroutine need only return to $IE77. To modify the character before storing, the user subroutine should put the modified character in AY before returning to either $IL77 or $ID77. If the user subroutine discovers a special end-mode condition it should set the count in the input buffer so that when it is checked by the interrupt service routine it will be zero indicating an end-mode condition. Specifically, the count should be set to minus one if returning to $IL77 and zero if returning to $ID77 or $IE77.

If the user does not wish to modify the data or check for any special end-mode conditions the user subroutine should be specified as $IL77. For example if the user were filling reader buffers with binary data, $RU77 should contain $IL77.

TTI Example 1

Suppose it is desired to input a maximum of 18 characters from the keyboard into a buffer called KBUF. The following sequence of code will start input from the Teletype keyboard.

```
MRI     -22,AX              ;SET -COUNT IN
RM      AX,KBUF+1           ;INPUT BUFFER
MRI     KBUF,AX             ;ENQUEUE INPUT
JU      $ENQ                ;BUFFER ONTO
WRD     $KQ77               ;INPUT QUEUE &
JU      $KS77               ;START KEYBOARD INPUT
 .
 .                          ;CONTINUE PROCESSING
 .
```

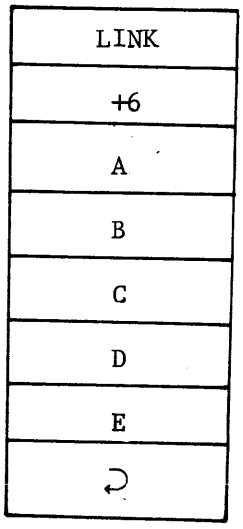
When $KS77 returns, keyboard input will have been started but the input buffer will not yet have been filled. Suppose no further processing can be done until KBUF is filled. Then $WAIT could be called to wait until the count location in KBUF becomes greater than zero, indicating the buffer has been filled and dequeued from the input queue.

```
ZR      AY                  ;WAIT UNTIL COUNT
JU      $WAIT               ;IS SET GREATER
JC      AO,GTZ,KBUF+1       ;THAN ZERO, BEFORE PROCESSING
```

If the keyboard user subroutine specified in $KU77 were $LINE and the following were typed on the keyboard,

ABD←CDE↵

KBUF would contain



Note that $LINE ignored the first D because it was followed by a back arrow and that the end-mode condition occurred on the sixth character because it was a carriage return. The interrupt service routine set the count to plus the actual number of characters stored.

TTI Example 2

Suppose it is desired to fill two reader buffers, RB1 and RB2, with a maximum of 100_{10} binary characters each. The following code would start reader input. Since binary data is being input $RU77 should contain $IL77.

```
MRI     -144,AX          ;SET -COUNT
RM      AX,RB1+1         ;IN READER BUFFER 1
RM      AX,RB2+1         ;AND READER BUFFER 2
MRI     RBI,AX           ;ENQUE BUFFER 1
JU      $ENQ             ;ONTO INPUT QUEUE
WRD     $RQ77
MRI     RB2,AX           ;AND BUFFER 2
JU      $ENQ             ;ONTO INPUT QUEUE
WRD     $RQ77
JU      $RS77            ;START READER INPUT
 .
 .
 .
```

As in the case of TTI example 1, when $RS77 returns, reader input will have been started but the input buffers will not yet be filled.

TTI Example 3

This example shows one way to accomplish double buffering of reader input. When a reader buffer is full of (or being filled with) data it appears on the "process data queue" called PBQ. When the data in a full reader buffer has been processed and is no longer needed, the buffer can be considered empty and

will appear on the "reader buffer empty" queue.

The two reader buffers RB1 and RB2 conform to the format for an input buffer (i.e. the first word is reserved for enquing the buffer onto $RQ77, the second word contains a negative count, and characters will be stored starting in the third word). Since the reader buffers must also be enqueued onto the "process data" queue (PBQ) or the "reader buffer empty" queue (RBEQ) each of the input buffers is immediately preceded by another link word. Thus RB1 is preceded by PB1 which is used to enqueue the first reader buffer onto PBQ or RBEQ and RB2 is preceded by PB2 which is used to enqueue the second reader buffer onto PBQ or RBEQ.

The first routine START defines the queues and input buffers, initializes the processor list and starts the scan.

The function of the processor READ is to initiate reader input into empty reader buffers. To do this, READ checks to see if there is a buffer on RBEQ. If there is it must be an empty buffer which needs to be filled so READ dequeues it from RBEQ, enqueues it onto PRQ (queue for buffers being filled) and $RQ77 (input queue) and calls the reader starter to begin reader input. Note that when the starter is called the reader may still be going because of a previous call to the starter. In such a case the starter simply returns.

The function of the processor PROC is to process data in a full reader buffer and then release the buffer for further input. To do this, PROC checks to see if there is a buffer on PRQ. If there is the buffer is either being filled (indicated by a count less than zero) or is full (indicated by a count greater than zero). If the buffer is full PROC processes the data then dequeues the buffer from PRQ and enqueues it onto RBEQ so that processor READ can initiate reader input into the now empty buffer.

```
 *031                                ;EXAMPLE 3
  032                                        ENTRY  PBQ,RBEQ
  033  000000  0  06 0010  11  START: MRI    PRCL-1,AX    ;INITIALIZE
       000001  1  000005
U 034  000002  0  11 0000  06         RM     AX,$PRCL     ;PROCESSOR LIST
       000003  0  000000
U 035  000004  0  00 0100  03         JU     .$STRT       ;& START SCAN
       000005  0  000000
U 036  000006  0  177777       PRCL:  WRD    READ-1
U 037  000007  0  177777              WRD    RSAV-1
U 038  000010  0  177777              WRD    PROC-1
U 039  000011  0  177777              WRD    PSAV-1
  040  000012  0  000000              WRD    0
  041                                ;FIRST INPUT BUFFER
  042  000013  1  000024       PB1:   WRD    PB2          ;LINK TO PRQ OR RBEQ
  043  000014  0  000000       RB1:   WRD    0            ;LINK TO $RQ77
  044  000015  0  177772              WRD    -6           ;-MAX. NUM. CHARS.
  045  000016  0  000000              WRD    0
  046  000017  0  000000              WRD    0
  047  000020  0  000000              WRD    0
  048  000021  0  000000              WRD    0
  049  000022  0  000000              WRD    0
  050  000023  0  000000              WRD    0
  051                                ;SECOND INPUT BUFFER
  052  000024  0  000000       PB2:   WRD    0            ;LINK TO PRQ OR RBEQ
  053  000025  0  000000       RB2:   WRD    0            ;LINK TO $RQ77
  054  000026  0  177772              WRD    -6           ;-MAX. NUM. CHARS.
  055  000027  0  000000              WRD    0
  056  000030  0  000000              WRD    0
  057  000031  0  000000              WRD    0
  058  000032  0  000000              WRD    0
  059  000033  0  000000              WRD    0
  060  000034  0  000000              WRD    0
  061  000035  0  000000       PBQ:   WRD    0,.-1        ;PROCESS QUEUE
       000036  1  000035
  062  000037  1  000013       RBEQ:  WRD    PB1,PB2      ;READER BUFRS EMPTY QUEUE
       000040  1  000024
  063          1      000041         END
```

```
 *0001                          ;ROUTINE TO PROCESS DATA
  0002                                ENTRY PROC,PSAV
  0003                          ;PROLOGUE
U 0004  000000 0 06 0000 11 PROC:  MR    PBQ,AX       ;ANY BUFFERS
        000001 0 000000
U 0005  000002 0 11 0100 03        JC    AX,ETZ,$NEXT;BEING FILLED?
        000003 0 000000
  0006  000004 0 11 0110 06        RMI   AX,P1,0      ;YES,CHECK COUNT TO
        000005 0 000000
  0007  000006 0 06 0001 12        MRD   .-1,AY       ;SEE IF THERE IS A
        000007 1 000005
U 0008  000010 0 12 1000 03        JC    AY,LTZ,$NEXT;FULL BUFR TO PROCESS
        000011 0 000000
  0009                          ;TASK
  0010  000012 0 00 0000 06        ZM    PSAV         ;YES-SET ACTIVE
        000013 1 000026
                                   .
                                   .                  ;PROCESS DATA
                                   .
U 0012  000016 0 00 0120 03        JU    $DEQ         ;REMOVE FULL BUFR
        000017 0 000000
U 0013  000020 0 000000            WRD   PBQ          ;FROM PROCESS QUEUE
U 0014  000021 0 00 0100 03        JU    $ENQ         ;AND ADD TO
        000022 0 000000
U 0015  000023 0 000000            WRD   RBEQ         ;'READER BUFFER EMPTY'
U 0016  000024 0 00 0100 03        JU    $EXEC
        000025 0 000000
  0017                          ;PROC SAVE AREA
  0018  000026 1 177777      PSAV:  WRD   PROC-1,0,0,0,0,0
        000027 0 000000
        000030 0 000000
        000031 0 000000
        000032 0 000000
        000033 0 000000
  0019         1      000034        END
```

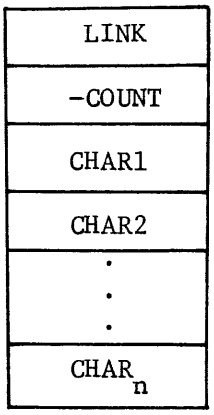
```
 *001                          ;ROUTINE TO BEGIN INPUT
  002                                 ENTRY RSAV,READ
  003                          ;PROLOGUE
U 004 00000 0 06 0000 11      READ:  MR    RBEQ,AX      ;EMPTY BUFR TO
      00001 0 000000
U 005 00002 0 11 0100 03             JC    AX,ETZ,$NEXT;BE FILLED?
      00003 0 000000
  006                          ;TASK
  007 00004 0 00 0000 06             ZM    RSAV         ;YES-SET ACTIVE
      00005 1 000032
U 008 00006 0 00 0100 03             JU    $DEQ         ;REMOVE EMPTY BUFR
      00007 0 000000
U 009 00010 0 000000                 WRD   RBEQ         ;FROM 'BUFR EMPTY QUEUE'
  010 00011 0 06 0010 12             MRI   -6,AY
      00012 0 177772
  011 00013 0 11 0110 06             RMI   AX,P1,0      ;SET -COUNT IN NEW
      00014 0 000000
  012 00015 0 12 0001 06             RMD   AY,.-1       ;INPUT BUFR
      00016 1 000014
U 013 00017 0 00 0100 03             JU    $ENQ         ;ADD NEW BUFR TO
      00020 0 000000
U 014 00021 0 000000                 WRD   PBQ          ;PROCESS QUEUE
  015 00022 0 11 0100 11             RS    AX,P1        ;AND TO
U 016 00023 0 00 0100 03             JU    $ENQ         ;READER QUEUE
      00024 0 000000
U 017 00025 0 000000                 WRD   $R077
U 018 00026 0 00 0100 03             JU    $RS77        ;START READER INPUT
      00027 0 000000
U 019 00030 0 00 0100 03             JU    $EXEC
      00031 0 000000
  020                          ;READ SAVE AREA
  021 00032 1 177777           RSAV:  WRD   READ-1,0,0,0,0,0
      00033 0 000000
      00034 0 000000
      00035 0 000000
      00036 0 000000
      00037 0 000000
  022       1     000040             END
```

Teletype Output - ICF

$CB77 allows for two types of output to TTO, priority output and normal output. Normal output should be used for outputting standard messages and data while priority output should be used for outputting important messages such as error conditions. The main difference between priority and normal output is that while priority output is in effect, no normal output can occur. Whereas, when normal output is in effect, it can be stopped temporarily to allow priority output.

The Teletype control block has two output queues, one called $PQ77 for priority output buffers and the other called $NQ77 for normal output buffers. Output buffers (both priority and normal) must conform to the following format:

Output Buffer



The first word of the output buffer is reserved for enqueuing the buffer onto the output queue. The second word must contain a negative count of the number of characters in the buffer which are to be output to TTO. The first character output will be in the third word of the output buffer.

There are two Teletype output starter subroutines, one called $PS77 to start priority output and the other called $NS77 to start normal output. Both $PS77 and $NS77 are entry points.

$PS77

To start priority output to TTO the user should first enqueue one or more priority output buffers onto the priority output queue, then call the priority output starter by a JU $PS77. When $PS77 returns interrupt control will be on, the ISR will be set up to allow TTO interrupts and priority output will have been initiated (i.e. the first character from the top buffer on $PQ77 will have been output). Then each time TTO interrupts the next character from the priority buffer is output. Note that the priority output starter simply initiated priority output by outputting the first character. The TTO interrupt service routine actually outputs the rest of the buffer.

If $PS77 is called while normal output is in effect, the actual initiation of priority output will be deferred until the next normal output end-mode condition occurs (see $OP77).

$NS77

To start normal output to TTO the user should first enqueue one or more normal output buffers onto the normal output queue and then call the normal output starter by a JU $NS77. When $NS77 returns, interrupt control will be on, the ISR will be set to allow TTO to interrupt and normal output will have been initiated (i.e. the first character from the top buffer on $NQ77 will have been output). If the normal output starter is called while priority output is in effect, the call to the normal output starter is ignored.

$OP77

When TTO causes an interrupt, control is transferred to $OP77, the TTO interrupt service routine, for processing. If output was started by the priority output starter then the output queue is considered to be $PQ77 and the output

buffer to be the top buffer on $PQ77. If output was started by the normal output starter then the output queue is considered to be $NQ77 and the output buffer to be the top buffer on $NQ77.

Each time $OP77 is entered, it adds one to the count in the output buffer. If the count is not yet zero, the next character in the output buffer is output to TTO and a non-end-mode return taken.

If the count has become zero it indicates that the entire buffer has been output which is considered to be an end-mode condition. The buffer is dequeued from the output queue and the new buffer now on top of the output queue becomes the new output buffer. The first character from the new output buffer is output to TTO and an end-mode return is taken. Thus, each of the buffers on the output queue is output to TTO and then dequeued until the output queue becomes empty. When this happens, output is through and the user must call an output starter before output to TTO will begin again.

Each time an end-mode condition occurs during normal output $OP77 checks to see if the user has attempted to start priority output. If the user has attempted to do so, $OP77 will then start up priority output rather than continue on to output the next normal buffer. When normal output is stopped after an end-mode condition by $OP77 in order to begin priority output, the user must call $NS77 to resume normal output.

TTO Example 1

Suppose it is desired to output two buffers, each containing nine characters, to TTO under normal mode. The following sequence of code would start the normal output.

```
MRI     -11,AX          ;SET -COUNT IN
RM      AX,NB1+1        ;NORMAL OUTPUT
RM      AX,NB2+1        ;BUFFERS
MRI     NB1,AX
JU      $ENQ            ;ADD 1ST BUFFER
WRD     $NQ77           ;TO NORMAL QUEUE
MRI     NB2,AX
JU      $ENQ            ;THEN 2ND BUFFER
WRD     $NQ77           ;TO NORMAL QUEUE
JU      $NS77           ;START NORMAL OUTPUT
          .
          .
          .           ;CONTINUE PROCESSING
```

When $NS77 returns the normal output will have been started. That is, the first character from NB1 will have been output. If the user now wished to stop normal output and start priority output he need only call the priority starter. For example, suppose an error condition was discovered and the user wished to print an alarm message stored in a priority buffer called ALRM.

```
MRI     -7,AX           ;SET -COUNT IN
RM      AX, ALRM+1      ;PRIORITY BUFFER
MRI     ALRM,AX
JU      $ENQ            ;ADD PRIORITY BUFFER
WRD     $PQ77           ;TO PRIORITY QUEUE
JU      $PS77           ;THEN START PRIORITY OUTPUT
```

When $PS77 returns, the actual priority message may not yet have started to be output since normal output might be in effect. However, the call to $PS77 will have been noted and when a normal end-mode condition occurs (i.e. when

all of NB1 has been output and NB1 has been dequeued) the output of the priority message will begin. When priority output is through the user may wish to continue the normal output of NB2 in which case he need only say JU $NS77 since NB2 is already enqueued onto $NQ77.

$HICF High Speed Reader/Punch Interrupt Acknowledge - ICF

Length: Absolute locations 17_8 - 24_8 plus 30_8 (24_{10}) relocatable locations.

This tape contains the ICF-type interrupt acknowledge routines for high speed reader/punch 76. Absolute locations 17_8 - 21_8 are used to save the SC when HSR causes an interrupt and to transfer control to the HSR interrupt acknowledge routine. Absolute locations 22_8 - 24_8 are used to save the SC when HSP causes an interrupt and to transfer control to the HSP interrupt acknowledge routine.

Both the HSR and HSP interrupt acknowledge routines store the TRP in $TRP, call $SAVI to save additional registers, and store the address-1 where the SC was stored in $INTL. The interrupt acknowledge routines then check the ready flag for high speed reader/punch 76 (input ready flag if HSR caused the interrupt or output ready flag if HSP caused the interrupt). If high speed reader/punch 76 caused the interrupt then control is transferred to the ICF-type interrupt service routine for high speed reader/punch 76. ($IP76 if HSR caused the interrupt or $OP76 if HSP caused it.)

If the interrupt was not caused by high speed reader/punch 76 then $HICF halts since an unknown reader/punch caused the interrupt. Therefore, if the user adds additional high speed reader punches to the system, $HICF must be edited and reassembled to check the ready flags for the additional high speed reader punches. For this reason $HICF is provided as both a source tape and as a relocatable object tape.

Entry Points:

$HICF ICF-Type High Speed Reader/Punch Interrupt Acknowledge

This is a dummy entry point since no other standard interrupt routine references any symbol in $HICF. Its purpose is simply to let the user know into what locations $HICF has been loaded.

HSR/HSP76 Interrupt Handler - ICF ($CB76)

This tape contains the control block, device starter subroutines and interrupt service routines for ICF-type handling of high speed reader/punch 76. If another high speed reader/punch is added to the system it must have a unique address other than 76. An interrupt handler like $CB76 must also be created to handle the additional high speed reader/punch. To do this, the source tape for $CB76 should be copied using %STE and the exchange command used to change every occurence of 76 to the address of the additional high speed reader/punch. This new source tape should then be assembled and the resulting object tape loaded to handle the additional high speed reader punch.

Length: 220_8 locations

Entry Points: Detailed descriptions follow

Name	Function
$CB76	address of control block associated with high speed reader/punch 76.
$SF76	HSR stop flag, can be set by the user to stop HSR input.
$RN76	for internal use by high speed reader/punch handler, contains address-1 to store next character from HSR.
$RQ76	HSR queue, contains pointers to HSR input buffers, the user must enqueue HSR input buffers onto $RQ76.
$RU76	contains address of user subroutine to modify HSR data and/or check for special HSR end-mode conditions.
$PN76	for internal use by high speed reader/punch handler, contains address -1 of next character to be output to HSP.

Name	Function
$PQ76	HSP queue, contains pointers to HSP output buffers, the user must enqueue HSP output buffers onto $PQ76
$RS76	HSR starter, subroutine called by the user to start input from HSR.
$PS76	HSP starter, subroutine called by the user to start output to HSP.
$IP76	interrupt service routine to process HSR interrupts.
$OP76	interrupt service routine to process HSP interrupts.
$IL76	return address for user subroutine specified in $RU76, this return adds one to the count in the HSR input buffer, stores the character in the HSR input buffer, then checks the count for equal to zero.
$ID76	return address for user subroutine specified in $RU76, this return stores the character then checks (but does not add one to) the count.
$IE76	return address for user subroutine specified in $RU76, this return checks the count only, it neither stores the character nor adds one to the count.

The control block for high speed reader/punch 76 is similar to the Teletype control block ($CB77) except that keyboard input and priority output information is not included. The control block is as follows, each label in the control block is an entry point.

```
$CB76=.
$RS76:      WRD Ø              ;HSR STOP FLAG
$RN76:      WRD Ø              ;HSR NEXT LOC
$RQ76:      WRD Ø,.-1          ;HSR QUEUE
$RU76:      WRD $IL76          ;USER'S HSR SUBROUTINE
$PN76:      WRD Ø              ;HSP NEXT LOC
$PQ76:      WRD Ø,.-1          ;HSP QUEUE
```

HSR Input - ICF

Input from HSR is buffer oriented and the format for an HSR buffer is the same as the format for a Teletype input buffer. That is, the first word is reserved for enqueuing the buffer onto the HSR queue ($RQ76) found in the control block. The second word contains a negative count of the maximum number of characters to be stored and characters from HSR are stored starting in the third word.

Input from HSR is handled in the same manner as reader input from TTI. Since there is no keyboard capability with HSR, there is only one HSR starter and one HSR queue. As with TTI reader buffers, the user must enqueue HSR buffers onto the input queue $RQ76.

$RS76

To start input from HSR the user should first enqueue one or more HSR buffers onto the HSR queue and then call the HSR starter by a JU $RS76. When $RS76 returns, interrupt control will be on, HSR input will have been initiated and the ISR set up to allow HSR interrupts. As with the Teletype reader starter, $RS76 simply initiates HSR input, the interrupt service fills the HSR buffer.

$IP76

When HSR causes an interrupt, control is transferred to $IP76, the HSR interrupt service routine, for processing. $IP76 performs the same functions as the TTI interrupt service routine $IP77. That is, characters are stored in the top buffer on the reader queue. When the buffer becomes full the count is set to plus the number of characters stored and the buffer is dequeued from the reader queue. The new buffer now on top of $RQ76 will then be filled and so on until all buffers on $RQ76 have been filled and dequeued. HSR input is then through and $RS76 must be called to begin more HSR input.

$SF76

$SF76 is a flag in the control block which can be set by the user to stop HSR input before the HSR queue becomes empty. If the user sets $SF76 to one, HSR input is stopped after the next HSR interrupt and $SF76 is reset to zero. Once this has been done the user must call $RS76 to restart HSR input from where it left off.

User Input Subroutines

$IP76 allows for user subroutines to alter data and check for special end-mode conditions in the same way that $IP77 allows such subroutines. The address of the user subroutine should be stored in $RU76 in the control block. When the subroutine is called, AY contains the HSR character and AX contains the address of the HSR queue.

$IL76, $KD76, $IL76

The user subroutine must return to one of three locations in the HSR interrupt service routine. These three returns are similar to the three returns for Teletype user subroutine returns. A return to $RL76 adds one to the count in the input buffer,stores the data (assumed to still be in AY) then checks

the count for equal to zero. A return to $ID76 stores the data and checks, but does not bump the count. A return to $IE76 simply checks the count, it neither adds one to the count nor stores the data. When the user subroutine is called $IL76 = TRP+1, $ID76 = TRP+3, and $IE76 = TRP+5. The standard user subroutine $ASCI may be used as a user subroutine for either HSR input or TTI input, and only one copy of $ASCI is needed if both HSR and TTI are running at the same time.

HSR Example 1

Suppose it is desired to fill an HSR buffer with a maximum of 48 binary characters. The following code would start reader input. Since the data is binary $RU76 should contain $IL76.

```
MRI     -60 , AX        ;SET -COUNT
RM      AX,HSRB+1       ;IN INPUT BUFR.
MRI     HSRB,AX         ;ENQUEUE INPUT
JU      $ENQ            ;BUFFER ONTO HSR
WRD     $RQ76           ;QUEUE AND
JU      $RS76           ;START HSR INPUT
  .
  .                     ;continue processing
  .
```

As in TTI example 1, when $RS76 returns HSR input will have been initiated but the input buffer will not yet be full.

HSP Output - ICF

The output to HSP is buffer oriented. The format for an HSP output buffer is the same as the format for a Teletype output buffer. That is, the first word is reserved for enqueuing the buffer onto the HSP output queue ($PQ76) found in the control block. The second word contains a negative count of the total number of characters to be output from the buffer. The first character output is

is in the third word.

Output to HSP is handled in the same manner as output to TTO. However, there is no priority output feature in HSP so there is only one HSP starter and one HSP queue.

$PS76

To start output to HSP the user should first enqueue an HSP buffer on the HSP queue and then call the HSP starter by a JU $PS76. When $PS76 returns, interrupt control will be on, output to HSP will have been initiated and the ISR set up to allow HSP interrupts. $PS76 simply initiates output by outputting the first character to HSP, the HSP interrupt service routine outputs the remaining characters.

$OP76

Control is transferred to the HSP interrupt service routine ($OP76) when HSP causes an interrupt. $OP76 performs the same functions as the Teletype interrupt service routine $OP77. That is, each time an HSP interrupt occurs a character from the output buffer is sent to HSP and the count location bumped. When all characters in the buffer have been output the buffer is dequeued from $PQ76. The new buffer on top of $PQ76 is output and so on until all the buffers have been output and dequeued. HSP output is then through and $PS76 must be called to begin more HSP output.

HSP Example 1

The following sequence of code will start output of an 18 character buffer called HSPB to HSP.

```
MRI     -22,AX          ;SET -COUNT
RM      AX,HSPB+1       ;IN OUTPUT BUFR
MRI     HSPB,AX         ;ENQUEUE OUTPUT
JU      $ENQ            ;BUFR ONTO HSP
WRD     $PQ76           ;QUEUE AND
JU      $PS76           ;START HSP OUTPUT
 .
 .                      ;continue processing
 .
```

As with TTO example 1, when $PS76 returns output to HSP will have been started but the entire buffer will not yet have been output.

$TTYQ TTY ICO Service

Length: Absolute locations 11-16 plus 631_8 (409_{10}) relocatable locations.

This tape contains the ICO-type teletype interrupt service routines and the associated teletype starter subroutines. Note that if ICO-type teletype interrupt service routines are used, the associated ICO-type teletype starter subroutines must also be used. The starter routines are called ICO-type even though interrupt control is off throughout them because these starters are associated with ICO-type interrupt service routines.

The ICO-type teletype interrupt service routines and starters are designed to handle multiple teletypes (one or more teletypes, each with a unique address). Associated with each teletype must be a control block and these control blocks must be enqueued by the user onto a queue called $TTYQ. The format for an ICO-type teletype control block is the same as the format for an ICF-type teletype control block ($CB77) except that the ICO-type control block has three additional words at the beginning of the control block. The first of these three words is reserved for linking to the next ICO-type teletype control block. The second and third words contain the instructions FO Ø,XX and SF XX,Ø respectively, where XX is the octal address of the teletype associated with that particular control block. The control block must be defined by the user.

The control blocks queued onto $TTYQ enables all teletypes to use the same ICO-type TTI and TTO interrupt service routines and starters. When an ICO-type starter is called it has as its argu-

ment the address of the control block associated with the particular teletype to be started. When an ICO-type interrupt service routine processes an interrupt, a subroutine ($FND, discussed later) is called to find which teletype caused the interrupt. $FND returns the address of the control block associated with the teletype which caused the interrupt. The information in this control block is then used by the interrupt service routine to process the interrupt.

Absolute locations $11\text{-}13_8$ are used to store the SC when a TTI interrupt occurs and to transfer control to the ICO-type TTI interrupt service routine. Absolute locations $14\text{-}16_8$ are used to save SC when TTO causes an interrupt and to transfer control to the ICO-type TTO interrupt service routine. The ICO-type TTI and TTO interrupt service routines perform the same functions as their ICF-type counterparts, $IP77 and $OP77 respectively. However, the ICO-type interrupt service routines turn interrupt control on while the interrupt is being serviced. The ICO-type teletype interrupt service routines have two undefined parameters, $LRM and $LPM, which are used in the calls to $ICO (see entry points in %RTX). $LRM and $LPM are the ISR masks for TTI and TTO respectively. The user must define $LRM and $LPM as parameters in one of his processors and declare $LRM and $LPM as entry points. All teletypes in a particular system must be handled in the same manner; that is they must be all ICO-types or all ICF-types.

Entry Points:

$TTYQ Queue for TTY Control Blocks (ICO-type)

Contains pointers to the ICO-type teletype control blocks. Associated with each teletype is an ICO-type control block whose format is the same as an ICF-type teletype control block (see $CB77). However, the ICO-type control blocks has three additional words at the beginning.

word 1 reserved for linking to next control block

word 2 FO Ø,XX

word 3 SF XX,Ø

The XX appearing in words 2 and 3 of the control block are the address of the teletype with which this particular control block is associated.

The control blocks must be enqueued onto $TTYQ by the user.

$TKS TTI Keyboard Starter - ICO

This subroutine is called to start keyboard input from any teletype in a system where teletype interrupts are handled in an ICO-type manner. The calling sequence is:

```
JU      $TKS
WRD     x
```

where x is the address of the control block associated with the teletype which is to be started. $TKS performs the same functions as the ICF-type keyboard starter ($KS77).

$TRS TTI Reader Starter - ICO

When teletypes are being handled in an ICO-type manner, $TRS should be called to start reader input from any teletype. The calling sequence is:

```
JU      $TRS
WRD     x
```

where x is the address of the control block associated with the particular teletype which is to be started. $TRS performs the same functions as the ICF-type reader starter ($RS77).

$TNS TTO Normal Output Starter - ICO

This subroutine should be called to start normal output to any teletype where teletypes are being handled in an ICO-type manner. The calling sequence is:

```
JU      $TNS
WRD     x
```

where x is the address of the control block associated with the particular teletype which is to be started. $TNS performs the same function as the ICF-type normal output starter ($NS77).

$TPS TTO Priority Output Starter -ICO

This subroutine should be called to start priority output to any teletype in a system where teletypes are being handled in an ICO-type manner. The calling sequence is:

```
JU      $TPS
WRD     x
```

where x is the address of the control block associated with the particular teletype being started. $TPS performs the same functions as the ICF-type priority output starter ($PS77).

$FND FIND INTERRUPTING DEVICE

Length: $72_8(58_{10})$ locations

This tape contains the subroutine $FND which is called by the ICO-type teletype interrupt service routines to find which particular teletype (in a set of multiple teletypes) caused an interrupt.

Entry Points

$FND Find Interrupting Device

This subroutine is used to find which device in a set of multiple devices caused an interrupt. It can be called by any ICO-type interrupt service routine which has a control block queue in the same format as $TTYQ. The calling sequence is:

```
JU      $FND
WRD     X
WRD     Y
```

where X defines the status bits to sense if the device is ready and where Y is the address of the control block queue associated with the set of multiple devices being tested. When $FND returns, interrupt control is off and the top buffer on the control block queue is the control block associated with the particular device which caused the interrupt. For example, suppose there were three teletype control blocks named CB1, CB2 and CB3 enqueued onto $TTYQ in the order CB2, CB1, CB3. If TTI caused an interrupt the ICO-type interrupt service routine would call $FND by saying:

```
JU      $FND
WRD     1000 ;BIT 9 = STATUS TEST FOR IRDY
WRD     $TTYQ;TEST TELETYPES
```

If the teletype associated with CB3 caused the interrupt, $FND would return with CB3 as the top buffer on $TTYQ.

$ASCI

Length: 31_{10} (37_8) Locations

Function: ASCII input user exit

Calling Sequence: AY contains character from input device.

AX contains address of input queue in the control block.

JU $ASCI (from within the input interrupt routine)

If the input character in AY is equal to zero, $ASCI returns to TRP + 5 which ignores the character. Otherwise, bit 8 is OR'd into AY to ensure an 8-bit ASCII character, the character is checked for being a carriage return, and a return is made to TRP + 1. If the character is a carriage return, the count location in the input buffer is set to -1 so that when the count is bumped at TRP + 1, an end mode condition appears.

This routine is called by $ECHO. It can also be used as the "user end mode check subroutine" specified in the control block if the special end mode condition is a carriage return.

$ECHO

Length: 57_{10} (71_8) Locations (ICF) 58_{10} (72_8) Locations (ICO)

Function: To echo TTI keyboard input

Calling Sequence: AY contains character from TTI keyboard

AX contains address of keyboard queue in the control block.

JU $ECHO (from within TTI interrupt service)

The purpose of $ECHO is to output the character in AY by putting it in the echo buffer, enqueuing the echo buffer onto the priority output queue and calling the priority output starter. The addresses of the echo buffer, priority output queue and priority starter are calculated from the address of the keyboard input queue. Therefore $ECHO can echo the input from keyboard only.

$ECHO first checks to see if AY is zero. If is is zero, a return to TRP + 5 is taken. Otherwise, $ASCI is called to OR in bit 8 and check for end mode. When $ASCI returns, $ECHO checks the count in the echo buffer. If the count is not yet zero then the previous character has not been echoed and $ECHO return to TRP + 5 as if the character were Ø (i.e. the character is ignored). Otherwise, the character is stored in the echo buffer. The echo buffer is then enqueued onto the priority output queue, the priority output starter is called and

control returns to TRP + 1. If the character was a carriage return, a line feed is also echoed. $ECHO is called by $LINE. There are two versions of $ECHO, one for use with ICF teletype routines and one for use with ICO teletype routines. Both versions perform exactly the same functions but have different ways of calling the priority output starter.

$LINE

Length: 72_{10} ($11Ø_8$) Locations

Function: To echo and input a line from TTI keyboard

Calling Sequence: AY contains character from TTI keyboard.

AX contains address of keyboard input queue in the control block

JU $LINE (from within Teletype interrupt service routine)

This subroutine first checks to see if AY is zero. If AY is zero control returns to TRP + 5. Otherwise, $ECHO is called. When $ECHO returns, the character is checked to see if it is a backarrow or rubout. If the character is a backarrow, keyboard next loc ($KN77) and the count in the keyboard buffer are each decremented by one (unless doing so would step these pointers back beyond the beginning of the buffer) and control returns to TRP + 5. This procedure deletes the previous character.

If the character is a rubout then the keyboard buffer count and next loc are reset to their initial values, thus deleting all previous characters in the line. Control then passes to TRP + 5.

If the character is neither a backarrow nor a rubout, then control returns to TRP + 1.

This routine can be used as the "user end mode check subroutine" specified in the control block if the user

wishes to echo keyboard input where the end mode condition is a carriage return, and allowing the backarrow and rubout editing features.

$TKP - TIME KEEPER INTERRUPT SERVICE

Length: absolute locations 100_8-103_8, plus 152_8 relocatable locations

Function:

This ICF-type interrupt service routine is used with the Real Time Clock to keep track of time-of day and/or time intervals. The routine has two constants which are set to the 60 cycle version of the Real Time Clock. These constants can be changed to accomodate any setting for the Clock. These constants are in $TPIN and in $IPSC (see below).

Entry Points: Detailed description follows

$DAY	-	day counter
$HOUR	-	hour counter
$MIN	-	minutes counter
$SEC	-	seconds counter
$FSEC	-	fraction of second counter
$CLOC	-	rollover counter (same units as $FSEC)
$TPIN	-	ticks per interrupt (value counted down in loc 103)
$IPSC	-	interrupts per sec (defines $FSEC and $CLOC units)
$TKP	-	start of time-keeper routine (for identification on load-map only)
$TMST	-	routine to start up the Real Time Clock
$GTTM	-	routine to obtain time of day

Detailed Discussion:

The two constants $TPIN and $IPSC define the units for the operation of the program. Both quantities must be set to a negative count. The clock hardware increments location 103 at a regular interval determined by staples wired on the clock itself. When location 103 overflows, the clock interrupts.

$TPIN is the quantity which is placed in location 103 and determines how much time elapses before an interrupt occurs. At each interrupt, location 103 is restored to the constant in $TPIN, and both $CLOC and $FSEC are incremented. The value in $CLOC is allowed to overflow (wrap around). The value in $FSEC, however, is compared to the value in $IPSC and when their sum is equal to Ø ($IPSC is negative, $FSEC is positive), the value in $SEC is incremented. Thus $IPSC determines the number of interrupts per second. $SEC and $MIN are incremented modulo 60. $HOUR is incremented modulo 24, the overflow being counted in $DAY.

To start the timer, the entry $TMST should be called by the system start up. The clock will be running with location 103 initialized upon return. If any of the quantities $DAY, $HOUR, $MIN, $SEC or $FSEC are to assume initial values they should be set before $TMST is called. Also, if the Real Time Clock being used is not the 60 cycle version, or if it is the 60 cycle version but interrupts are desired more than once a second, the quantities $TPIN and $IPSC should be set before calling $TMST. The calling sequence itself is simply JU $TMST. Typically this call will be just before calling $STRT (in %RTX) to get the system going.

To get instantaneous time-of-day, the entry point $GTTM can be called. This routine turns interrupt control off to prevent the time variables from being changed while they are fetched. It then stores $DAY, $HOUR, $MIN, $SEC and $FSEC sequentially in the five locations immediately following the JU $GTTM, turns interrupt control back on and returns. Thus the call to $GTTM should look like

```
JU      $GTTM
LOC     .+5
```

where the LOC statement reserves the five locations needed to store the time-of-day.

Examples

1. If $TKP is used as loaded, the Real Time Clock must be the 60 cycle version. $TPIN is set to -74_8 (-60_{10}) and $IPSC is -1. Thus the clock will interrupt once per second after counting 60_{10} in location 103 (at the rate of one count per 1/60 second).
2. If the real time system requires interrupts every half second and the 60 cycle clock is being used, set $TPIN to -36_8 (-30_{10}) and $IPSC to -2.
3. In general, if the rate at which the clock increments location 103 is n per second, and the unit time interval desired for $CLOC and $FSEC is $\frac{1}{m}$ seconds, set $TPIN to $-\frac{n}{m}$ and $IPSC to -m. The quantity m must be ≥ 1 and to keep accurate time, the quantity $\frac{n}{m}$ should be an integer.

Notes:

1. The timer algorithm is self-correcting for instances when the clock cannot interrupt due to the interrupt control being off. The interrupt control can be off for a period of time not exceeding the maximum of

   a) one minute or

   b) 32767 clock increments

   without loss of timing accuracy in the long run (i.e. $TKP catches up).
2. The quantity $CLOC can be used to time out specified intervals up to a maximum interval of 65535 units of $\frac{1}{m}$ seconds (see Examples) using

the $WAIT routine as follows:

Suppose it is desired to delay 100_8 $CLOC counts before continuing in some processor. The following code accomplishes this:

```
MR      $CLOC,AX        ;ADD INTERVAL
MRI     100,AY          ;TO CURRENT $CLOC
RRC     AO,P1,AY
JU      $WAIT
JC      AO,GEZ,$CLOC    ;(SEE $WAIT WRITE-UP)
```

Since $CLOC is a rollover counter, the comparison supplied to the $WAIT routine above will fail until $CLOC has been incremented <u>at least</u> 100_8 times, even if $CLOC overflowed somewhere along the line.

3. $TKP always takes an end-mode return after servicing an interrupt.

%RTX DIRECTORY TAPE

A directory tape is provided for the Real Time Executive and the standard interrupt service routines. It has the following routines in this order.

1. $TKP
2. $LINE
3. $ASCI
4. $TTYQ
5. $FND
6. $CB77
7. $TICF
8. $CB76
9. $HICF
10. $SAVI
11. $SRET
12. $SETM
13. $GETM/$PUTM
14. $SREL
15. $WAIT
16. $AHGH/$ALOW
17. $ATCH
18. $ENQ/$DEQ
19. %RTX

Notes:

1. If an executive with $USAV/$URES calls is desired, then the tapes for %RTX, $SREL and $AHGH/$ALOW which have calls to $USAV/$URES should be force loaded before the directory tape is loaded.
2. If the user has any references to $ECHO (or $LINE which calls $ECHO) then either the ICO version or the ICF version of $ECHO must be force loaded either before or after the directory tape has been loaded.

GRI Computer Corporation

320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

TEL: (617) 969-0800