June 23, 1969

Many errors in this one

# TS Interrupt System

## Introducti

The Interrupt System, which provides the sole interface between user processes and the outside world, is divided into two parts, the Central portion consisting of the code proper, and the PPU portion comprising the actual communication with the external devices.

The Interrupt system uses several objects which reside in ECS:

1. Files and event channels, which provide the immediate interface between ECS and the Interrupt System. They are seen by user level processes and appear just like ordinary files and event channels, except that they are stationary in ECS.

2. Pseudo-processes, which are used by the Interrupt System to simulate processes for hanging on event channels. (They are also used as a convenient storage place for information not neing used.) Pseudo-processes are transparent to user level processes.

## I Central Memory Portion

There are two sections to the Central code of the Interrupt System. The first, Initialization, sets up ECS at the beginning of time, sets up a few somethings in Central Memory and disappears. The second section consists of a collection of routines which work with inidivudal devices, plus some miscellaneous "stuff".

### 1.1 Initialization of ECS

The routine INTINIT constructs in ECS all objects needed by the Interrupt System. This is done at the beginning of time so that they will be in a position in ECS such that they never have to move. When INTINIT is called, there must be no gaps in ECS.

INTINIT is called at 2 different times:

1. At INTINITA , early in initialization before more than a very few things have been constructed. It is used to construct a file to contain C-list indices into the master C-list of interesting things constructed later.

2. At INTINITB for the actual construction of objects. It first constructs a file to be used for the interrupt queues needed by UNHUNG1. (See 1.2.2) (No user ever sees this file, and in fact, its C-list entry is destroyed.) It then calls the Device Routines to construct the objects for each of the various kinds of devices. Currently there are two separate kinds of devices: MUX and the simple devices.

The Device Routines actually use a collection of three subroutines, specially constructed for INTINIT, to do their work; i.e., to construct the particular objects in ECS used by the Interrupt Routines.

MEC - This subroutine constructs an event channel and leaves the capability in the Master C-list. It is entered with the count of the maximum number of events expected in the event channel at one time.

MPS - This subroutine constructs a pseudo-process. It is entered with the size of the process in words. It makes no permanent C-list entry; the C-list entry is destroyed at the end. It returns with the absolute address of the pseudo-process in X5 and the MOT index and unique name in X4 .

MFILE - This subroutine constructs a one level file with a data block whose size is given in X6. It leaves the entry in the C-list and returns with the absolute ECS address of the data block in X0.

Each type of device is considered as a class of objects, associated with which is an interrupt queue and a file in ECS containing pointers to the pseudo-processes. (See Figure 1.) All of these are set up by the routine NEWCLASS, which expects five parameters:

1. the location of the Interrupt Queue Index within the pseudo-process for this class of objects,

2. the location in Central of a pointer in ECS to the file containing the locations of pseudo-process,

3. the location in Central pointing to the interrupt queue for this class of devices in ECS,

4. the interrupt index for this class of devices,

and 5. the number of devices in the class.

In summary, the basic function of INTINIT is to create a file in ECS available to user processes which contains the first C-list index of an object belonging to each class of interrupt devices (INTINIA), to create a file (never seen by a user process, in fact removed from the master C-list) which

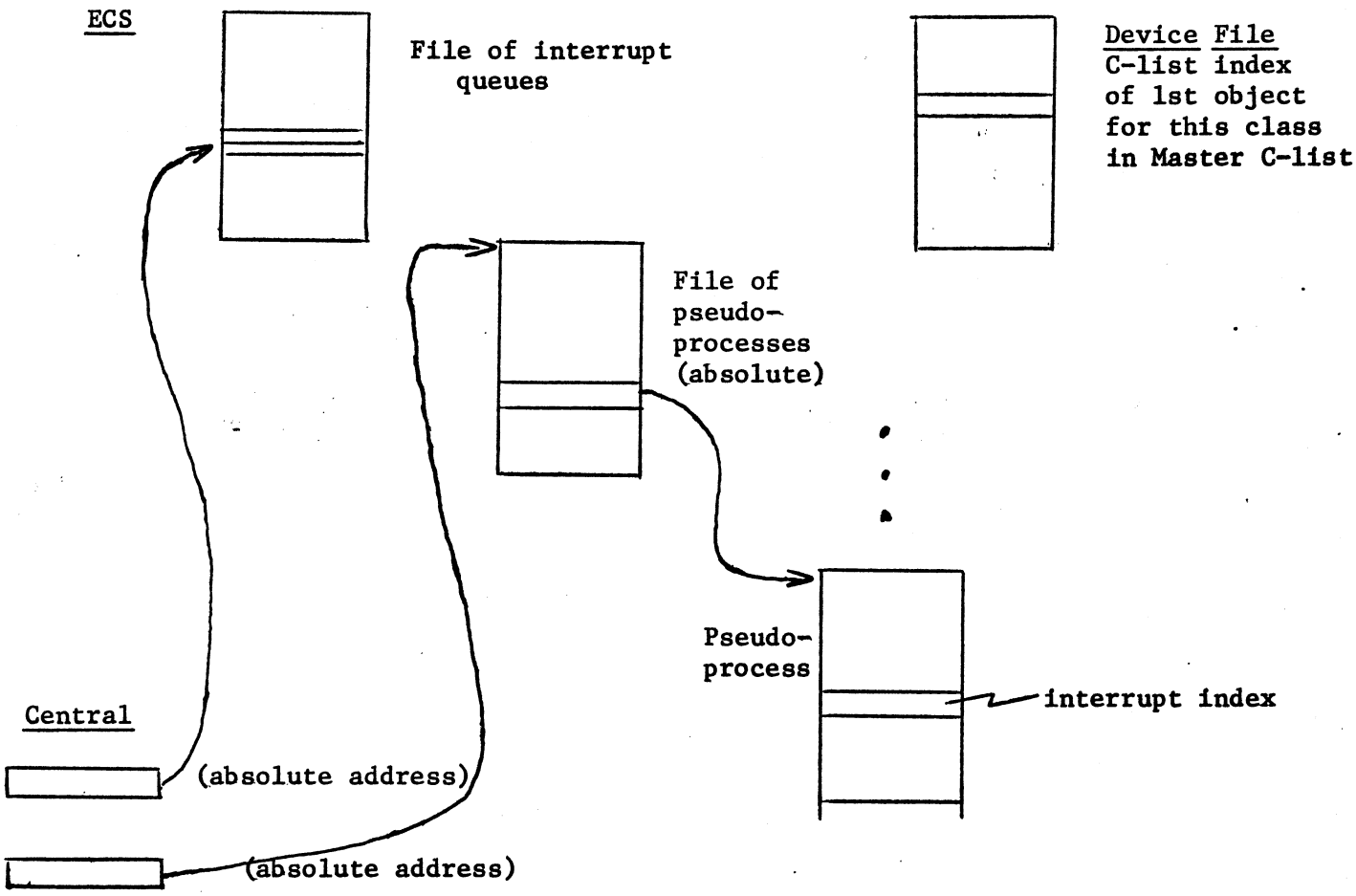



Figure 1

contains the absolute addresses of the pseudo-processes used by those devices, and finally, to create a pair of words in the file of interrupt queues used for the interrupt queue for that class of devices.

## 1.2 Interface between the Interrupt System Central Code and the ECS System Central Code

### 1.2.1 Calls from the Interrupt System to the ECS System

HANG - This routine is called to hang a pseudo-process (or process when called by other routines) on an event channel. It expects the following parameters:

1. the address of a scratch area it can use
2. a queuing word index to use, found in the pseudo-process,
3. identification of the pseudo-process
4. identification of the event channel to be used.

EVENT1 - This routine places an event on the specified event channel. It expects the following parameters:

1. event channel to be used
2. identification of process or pseudo-process sending the event
3. event datum
4. origin of scratch area including an address relative to this origin to which the disposition of the event is returned.

### 1.2.2 Calls from the ECS System to the Interrupt System

UNHUNG1 - This routine signals the arrival of an event to a pseudo-process. It expects the following parameters:

1. return address
2. origin of scratch area
3. the absolute address in ECS of the pseudo-process
4. the event received.

UNHUNG1 looks into the pseudo-process for data: first it uses the address specified in word 4 of the pseudo-process to chain it on an interrupt queue designed for each particular device. The interrupt queue is maintained by two words in a file in ECS. (See Figure 1.) The first word points to the absolute address of the first pseudo-process in the queue, and the

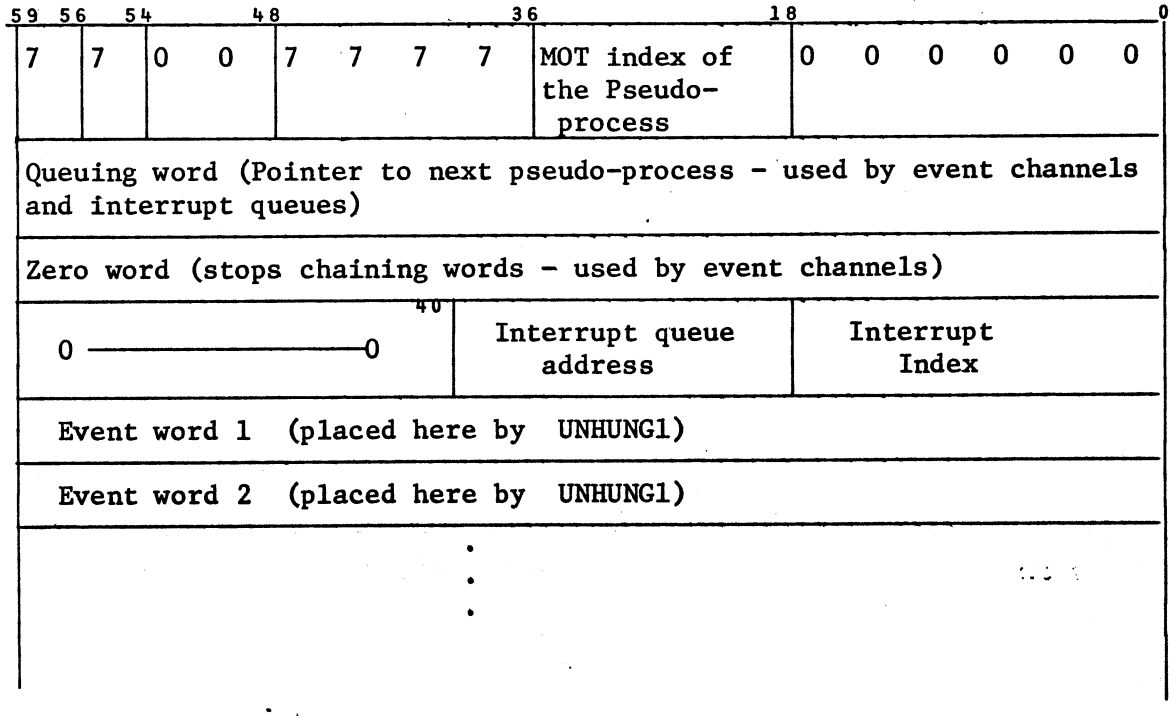
second word points to the last one in the queue. Pseudo-processes are chained on the queuing word (word 2) in the pseudo-process.

Next UNHUNG1 takes the Interrupt Index (also found in word 4 of the pseudo-process), which points to a particular device, and stores it in I.WAKE when I.WAKE has gone zero.

The Interrupt System calls this routine when it has tried to hang a pseudo-process on an event channel (using HAND) and gets an event back immediately.

Figure 2

1st Part of Pseudo-Process




## 1.2.3. Interlock Facility

The Interlock Facility is used to prevent interrupt code from referencing event channels while ECS is doing so. The cell I.LOCK is set non-zero by the ECS system whenever the ECS system is about to work on event channels, and is set zero when the work is completed.

Currently, the interrupt system always checks the cell upon entry to any of its code, and if it is non-zero, quits immediately. (Eventually the interrupt system could be more discriminating and only check I.LOCK when it was about to work on event channels. This could be used by interrupt routines desiring immediate access to a file and only a file.)

## II The Peripheral Processing Unit Portion

There are two areas to the PPU portion of the Interrupt System. The first, the Master PPU, serves to synchronize the Interrupt System. The second area consists of the indiiidual PPUs which handle the individual devices. In some instances, several devices are handled by one PPU, and in at least one instance (the disk) one device is handled by 2 PPUs.

### 2.1 The Master Peripheral Processing Unit (MPPU)

The master PPU handles the synchronization of the Interrupt System with a large loop, starting at MLOOP, which performs the following actions by means of a succession of return jumps:

1. Calls a routine which checks for the status of the user; e.g. arith errors, or RA+1 = 0 (indicating a simulated SCOPE call). If either of these two conditions holds, the PPU calls the ECS system via a monitor exchange jump (XJ).

2. Checks I.WAKE to see if there are any calls on the interrupt system from the ECS system.

3. Checks a channel, INTCHAN (as spelled in listing for MPPU), for calls on the Interrupt System from the other PPUs.

4. Calls a routine to update the master clock in Central (S.MASTR) which is run in steps of one millisecond and contains the true time in milliseconds since the system was started. This routine must be entered at least once every 4 milliseconds.

5. Calls a routine to update the clock S.QUANT which signals the end of a quantum for a user program by going positive (over-flowing). In this case, a monitor exchange jump is signalled to Central.

6. Calls a routine to update a charge clock, S.CHARG, which is updated whenever user code or interrupt code, but not system code, is running in Central.

7. Calls the routine DOINT to check a table for pending interrupts. (MPPU maintains in the table a list of those interrupt routines which have been signaled via either the ECS system or INTCHAN and have not yet been called.) If they are sending interrupts, it scans the table for the first one pending and having found it, find the P-counter in a table in Central, copies it into an exchange jump package located in Central (at I.BOX in the routine CENLINT), and then performs an exchange jump to that package. Since the table is ordered by interrupt number, those with low interrupt numbers are called first.

8. Enters a short loop of 12-24 milliseconds and checks the P-counter. If it is zero, MPPU assumes that the Interrupt was unsuccessful due to I.LOCK being non-zero when checked by the Interrupt Routine. It then goes away, and will make this interrupt call later. If the P-counter is non-zero, it assumes that the interrupt routine is running. It then continues cycling through this short loop, watching for the P-counter to go to zero, checking now and then for new interrupt requests coming in on INTCHAN or in I.WAKE, and recording them. It also maintains the master clock (but no other clocks). When the P-counter coes to zero, it restarts Central with an Exchange jump, and updates the master clock (S.MASTR) and charge clock (S.CHARG) to compensate for sloppiness at the beginning and end of the routine.

## 2.2 The PPU Interrupt Routines

The basic operation of an ordinary interrupt routine involves the following actions:

> If working with event channels (or if the coder was lazy), an Interrupt Routine first checks I.LOCK. If I.LOCK is non-zero, the routine must promptly jump to location zero within a few milliseconds (like 4 or 5). If I.LOCK is zero, the routine proceeds to do whatever it was planning to do. When finished, it jumps to zero, signalling the end of the interrupt.

An interrupt routine gets a pseudo-process off of the appropriate interrupt queue by calling DINTQ, with the absolute address in ECS of the queue. DINTQ either returns with an indication that there were no pseudo-processes on the queue, or it unchains the first pseudo-process and returns its absolute address. (The event can be found in the pseudo-process.)

## 2.3 General overview of How a User Event is Transmitted into Action by an Interrupt Routine

1. The user sends the event to the event channel.

2. If the event channel routines detect the fact that there is a pseudo-process hung on that event channel,

3. they unchain that pseudo-process from the event channel and transfer control to UNHUNG1.

4. UNHUNG1 looks into the pseudo-process (word 4) and finds that the interrupt index is - which it stores into I.WAKE.

5. UNHUNG1 finds (word 4) what the absolute address in ECS of the interrupt queue is and chains the pseudo-process into that queue as described.

6. UNHUNG1 also places the events in event word 1 and 2 in the pseudo-process.

7. The master PPU then discovers I.WAKE ≠ 0, records this fact in its own tables and sets I.WAKE back to zero.

8. When a suitable time occurs (hopefully before too long), it does an XJ to the appropriate interrupt routine as determined by its own tables.

9. The interrupt routine then does various things, including calling the general routine, DINTQ , which takes the pseudo-process off the interrupt queue and passes it back to the interrupt routine.

10. Finally, if when the event was received by the event channel, there were no pseudo-processes hanging, the event is stored on the event channel queue, and later, when the interrupt routine desires to hang a pseudo-process on an event channel, the event channel routines return with the event, and UNHUNG1 is called by the code associated with the Interrupt routines themselves.