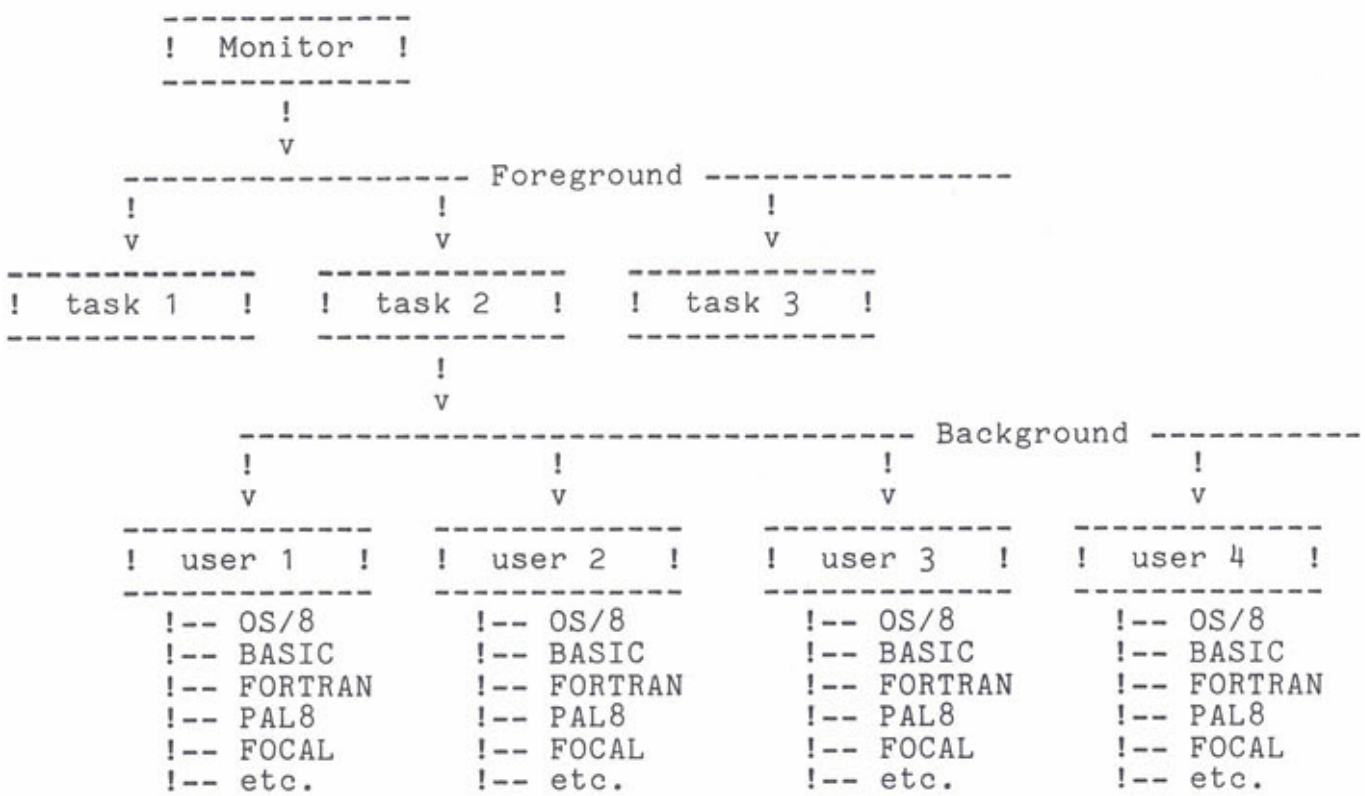
MULTI8 SYSTEM MANUAL



Third printing, july 1979

This manual is intended for the system-manager who needs a thorough understanding of the system. It is the reference document for MULTI8 and will be kept up to date for future changes and extensions. This version describes the MULTI8 V7A monitor and system tasks.

MULTI8 Release Notes is a seperate document primarily intended for users who upgrade from V6C to V7A. It lists the differences between the two versions and guides you in converting your own tasks.

Timesharing users of MULTI8 will find all information pertaining to terminal operation and background programming in the MULTI8 Terminal Manual. Copies of these manuals may be ordered from:

Westvries Computer Consulting B.V.
Rijksstraatweg 19, 1969 LB Heemskerk
The Netherlands

Print History

September 1976	preliminairy and incomplete version
November 1976	first printing, describing V6A
April 1977	second printing, describing V6B
July 1979	third printing, describing V7A

Preface

MULTI8 is the result of a two-year co-operation of the Medical Biological Laboratory of TNO Rijswijk and the Physiology Department of the University of Utrecht, both in Holland. During this period the system was redesigned several times, eventually resulting in a stable piece of software that has proved to be very reliable. From the beginning the designers of MULTI8 have felt that the system should be general and flexible so that it can be used by other PDP8 users also. In that way we are assured of a long and stable life which is of utmost importance to the users of the system.

We think that a complex software product like MULTI8 can not be distributed without good maintenance and support. Therefore MULTI8 is distributed via a commercial organization, giving possibilities for support that can not be realized by the non-profit laboratories where the system was developed.

I hope that MULTI8 will be a contribution to the use of that silly '12-bitter', which time after time has proved to be competitive with much wider machines.

Ernst Lopes Cardozo
Utrecht, november '76.

MULTI8 V7A is the accumulated result of about 3 years of further development done at the Physiology department of the University of Utrecht. An important factor in the realization of V7A is the availability of the Memory Management Unit built by DIGICOS BV. This hardware module, designed to the specifications of the MULTI8 implementers, enhances the protection and addressing structure of the PDP8 to allow a smooth operating virtual memory system. Besides the 'more' and 'faster', emphasize has been on 'better'. Better configuration procedures, better (yes, it was possible to improve) reliability, better maintainability. And still, in this exploding microprocessor world, the old PDP8 seems to hold a place for its own.

Ernst Lopes Cardozo
Ravenswaay, june 1979

TABLE OF CONTENTS

1. Introduction

MULTI8 is a real-time/timesharing system for the family of PDP8° computers. The system consists of two parts, the foreground doing the real-time work, and the background executing the timesharing programs.

The foreground consist of 2K resident code called the Monitor. Monitor implements all necessary primitives to enable a large number of independent 'tasks' to run concurrently and cummunicate with each other in a well defined and secure manner. Most of the functions that are in traditional operating systems burried in the monitor or executive are in MULTI8 built as separate tasks, resulting in an extremely flexible system.

Tasks may CALL each other in much the same way as subroutines are called. They can STOP and START each other or WAIT for SIGNALs from other tasks or from outside the system (interrupts). External tasks reside on the system disk, whereas internal tasks are permanently present in memory. External tasks are brought into memory automatically whenever required. Monitor takes care of dynamic core allocation and task code relocation. Tasks may specify that they have to be CONNECTed to one or more external interrupts and supply their own interrupt handling routines. This mechanism gives direct acces to MULTI8'S fast interrupt response (less than 250 usec worst case, 20 usec normally). Within the foreground a number of system supplied tasks are available, eg. device drivers, a command decoder, ODT-task, password task, etc.

The timesharing background is created by a number of co-operating tasks, that use an 8K to 32K section of memory to generate a virtual PDP8 for each timesharing user. On these virtual machines the OS/8° single user operating system is run, giving each user acces to an efficient filesystem and a powerful set of system programs. Full device sharing (eg. lineprinter, disk, DECtape, etc.) is provided by the timesharing subsystem in an automatical way. Users do not need to allocate or release devices explicitly. Background users are denied destructive access to both the foreground as well as other user's programs and data. They may however access other user's (or the system's) disk area on a read-only basis. Powerful communication primitives are implemented to allow the background programs to direct foreground tasks to perform any desired realtime action. Thus a timesharing program may take analog data or perform other highly real-time functions by triggering a foreground task to perform the time-critical actions. This results in a well structured and efficient system design.

The generation and maintenance of MULTI8 requires a PDP8E or PDP8A based OS/8 V3D system with sufficient file storage space to contain the MULTI8 source files. The execution of MULTI8 with one or more OS/8 backgrounds requires a minimum of 16K memory and at least 256K disk storage (RF08). Check the System Generation section for details on hardware requirements.

The MULTI8 package consists of PAL8 sourcefiles that should be assembled with a configuration parameter file that specifies the

- - - - - -

° PDP and OS/8 are trademarks of Digital Equipment Corporation.

hardware and software environment. This includes peripherals, number of backgrounds, etc. First the MULTI8 monitor, contained in the files M1.PA - M5.PA is assembled, with a DDT-compatible symboltable output. This symboltable is then processed by the program GLOBL, resulting in a file with symbol definitions (MS.PA). Next the monitor dump program and all external tasks are assembled with these symbol definitions.

This manual starts with the description of Monitor, which implements the interrupt handling scheme, multi-tasking and (foreground) memory allocation. Next a number of system tasks are treated, the system disk driver, terminal handlers and some more. Thereafter the timesharing subsystem is considered, first the resident central emulator, the input reader and output writer, then the non-resident emulator tasks and the background scheduler. A separate chapter is devoted to configuration and installation procedures. The last chapter describes some utility programs that are part of the MULTI8 package.

The reader is assumed to have a fair knowledge of the PDP8 instruction set, the PAL8 assembler and the OS/8 system. For reference he may find the Small Computer Handbook and the OS/8 Handbook useful.

1.1 Coding Conventions

Throughout the code of MULTI8 and its tasks a number of coding conventions have been adhered to. They were selected in order to increase the legibility of the source program:

- Code that is executed with the interrupt system disabled is marked by a triple slash before the comment, eg.

```
        CLA             ///CLEAR ACCUMULATOR
```

- Code that is executed with a 'strange' datafield in effect is marked by double slashes. In the monitor this means that the datafield differs from the instruction field. In some tasks the 'normal' datafield setting is the field where the background data tables reside, field 1, which may be different from the instruction field. In these tasks the double slashes appear when the datafield does not equal the normal setting.

- Instructions that follow a skip-instruction or a call to a subroutine that has multiple exits are marked by a single space preceding the op-code, eg.

```
        SZA             /AC=0 ?
         JMP ELSE       /NO
```

- Parameters following a subroutine call are preceded by three spaces. Thus a call to an OS/8 handler will be coded like:

```
JMS I (HANDLER  /CALL THE HANDLER
   FUNCTION     /FIRST PARAMETER
   BUFFER       /SECOND PARAMETER
   BLOCK        /LAST PARAMETER
 JMP ERROR      /ERROR RETURN, NORMALLY SKIPPED
CLA CLL         /NORMAL RETURN
```

1.2 System Layout

A sample layout of the components of MULTI8 in the PDP8 memory is shown in figure 1-1. Not all components are needed in all configurations. Consult the configuration section for details.

- o - o - o -

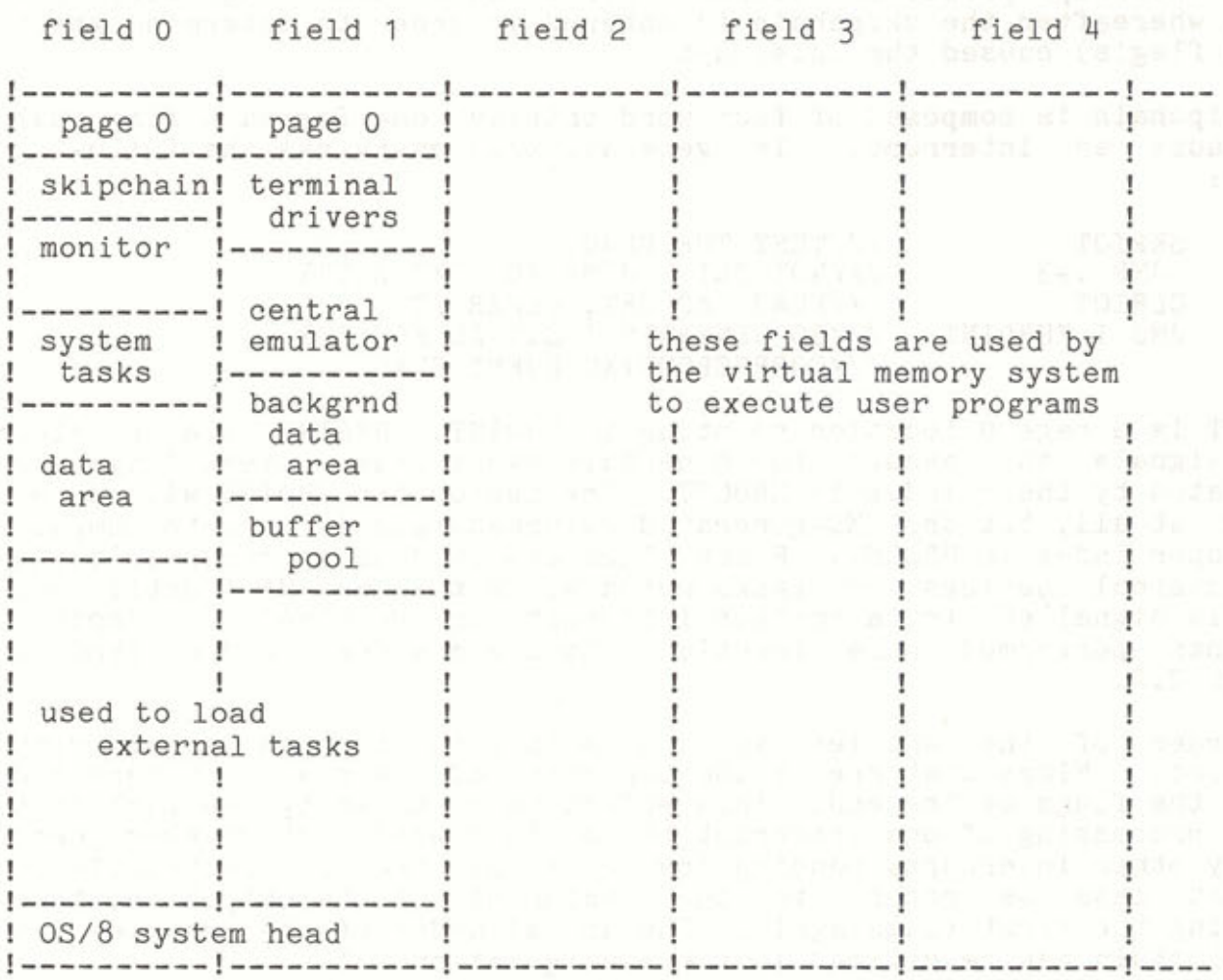


Figure 1-1. Sample memory layout.

2. Monitor

Monitor is the lowest level in the MULTI8 structure. Its functions are 1) interrupt handling, 2) multi-tasking and 3) memory allocation. These functions will be described in the above order. Section 2.4 will show how an external task is coded.

2.1 Interrupt Handling

As MULTI8 is intended to be a fast real-time system, interrupt handling must be as quick as possible. Interrupt handling code might be executed a few thouthand times per second, which means that every single instruction must be considered with care. The interrupt handling logic has been designed with these facts in mind.

When the cpu honors an interrupt request, a hardware generated JMS 0 instruction is executed. At address 1 a JMP 177 is found, where the actual interrupt processing starts. First the AC and flag word are saved, whereafter the skipchain is entered in order to determine which device flag(s) caused the interrupt.

The skipchain is composed of four-word entries, one for each flag that may cause an interrupt. In general, each entry has the following format:

```
SKPIOT          ///TEST THE FLAG
 JMP .+3        ///NOT SET - JUMP TO NEXT ENTRY
CLRIOT          ///FLAG WAS SET, CLEAR IT
JMS I ZHRDINT   ///GO GENERATE A SIGNAL FOR THE
                ///CORRESPONDING EVENT FLAG
```

ZHRDINT is a page 0 location pointing to HRDINT. HRDINT is a routine that signals an 'event' for a certain event flag. Event flags are designated by their index in HRDLST. The subroutine HRDINT will never return at all, but the JMS-generated returnaddress is used to compute the proper index in HRDLST. Event flags are used to synchronize tasks and external devices or tasks mutualy. A task may WAIT until some event is signalled, ie. a certain interrupt is received or another task has performed some function. Events are further described in section 2.2.

The order of the entries in the skipchain induces a priority structure. Flags that are at the beginning of the chain are serviced before the flags at the end. This effect is enhanced by the fact that after processing of one interrupt the monitor will test whether there are any other interrupts pending (by means of the SRQ-instruction). In that case we return to the beginning of the skipchain, thus favouring the first flags again. The actual order of the entries in the skipchain can be defined in the configuration file.

So interrupts are processed by HRDINT and converted to software signal's. Now there are two possibilities- 1)there was a task WAITing

for this signal; 2)there was no task WAITing for this signal.

In case 1) the interrupt is termed a 'significant event' and the following actions are taken: The WAITing task is schudeled, that means its Task Control Block (see the section on multi-tasking) is queued in the INTQ. Next the status of the system is considered. If the system was executing some other foreground task, then a normal interrupt exit is made, returning to the interrupted task. If the system was at lower priority, ie. executing a background program or in the idle loop, then the processor status is saved (routine RSAVE) and the DISPATCHER is entered in order to select a foreground task for execution. Thus the background is pre-empted by foreground tasks. The system status is determined by examination of CURTSK, a page 0, field 0 variable with the following encoding:

Negative	a foreground task is in execution; CURTSK=Task Control Block Pointer (TCBP)
Positive	a background program is in execution. BJOB points to the dataset of the executing BG.
Zero	the system is executing the idle loop: NULL, JMP NULL

N O T E

The status of the idle loop is never saved.

N O T E

The encoding of CURTSK requires that the address of a Task Control Block is between 4000 and 7777. This limits the total number of tasks in the system to a (theoretical) maximum of 256.

In case 2) the interrupt is not a 'significant event'. The monitor will immediatly return to the interrupted program. Some interrupts require immediate action. Lets take the (TC08) DECtape control for example. When the controler is in 'search' mode it will generate an interrupt each time a blocknumber is read from the tape and deposited in memory. The software should compare this blocknumber with the desired blocknumber. In case these are the same, the controler status should be changed to 'read' or 'write' in order to initiate the actual datatransfer. However, the new controler status must be loaded before the first data word is over the tapeheads. This leaves about 400 usec after receipt of the interrupt.

This is just one example of a situation where immediate action is required in response to an interrupt. Tasks that have to deal with such conditions can use the CONNECT feature of the monitor. This means that the relevant skipchain entrie(s) are patched by the monitor in order to redirect the interrupt to the serviceroutine embodied in the task. A connected skipchain entry has the following format:

```
SKPIOT            ///TEST THE FLAG
 JMP .+3          ///NOT SET - JUMP TO NEXT ENTRY
CDF CIF TSKFLD    ///SETUP FOR TRANSFER TO FIELD OF TASK
JMP I CONTAB+n    ///JUMP VIA POINTER IN PAGE ZERO
```

CONTAB (connect table) is located on page 0 , field 0. When a task is connected to an interrupt (see below) the address of its interrupt service routine is entered in CONTAB.

N O T E

In the case of connected interrupts the flag is NOT cleared in the skipchain.

After doing what has to be done the interrupt routine must return to the monitor. This can be done in one of two ways:

```
        CDF CIF 0       ///PREPARE INTERRUPT EXIT
        JMP I ZFSTEXT   ///BACK TO PROGRAM, NO SIGNIFICANT EVENT
```

or:

```
        TAD (EVENT      ///AC 05-11 IS EVENT NUMBER
        CIF 0           ///LEAVE DATAFIELD=THIS FIELD
        JMS I ZSOFINT   ///GO SIGNAL SIGNIFICANT EVENT
           STATUS       ///BIT 01-11 IS TRANSFERRED TO WAITING TASK
                        ///ZSOFINT DOES NOT RETURN !
```

Both ZFSTEXT and ZSOFINT are on page zero of every foreground field.

When a connected task is finished, it must be disconnected from the interrupt before its memory space can be released. So the original skipchain entry must be restored. The monitor is able to do so because it has saved the CLRIOT at initialization time. All CLRIOT's can be found in CLRLST, which is in the data-area in field 0.

Internal tasks are CONNECTed by direct modification of the skipchain source. Examples of permanently connected interrupts are the system disk and the timer. External tasks are CONNECTed by Monitor when they are brought into memory and disconnected when their memory space is released. Whether an external task is to be connected is determined by its Task Header.

The Task Header ocupies the first words of each external task (locations 200, 201, etc.). The first word contains the task name (eg. "N^100+"A&3777). The second word contains the task length (number of pages, in bits 0-4), the autostart flag (bit 5) and the number of connected interrupts (in bits 6-11). For each connected interrupt two words follow, containing the device number (not the hardware device code, but the event number) and the entry address of the interrupt service routine in the task.

```
        *200
        "A^100+"1&3777  /TASK NAME IS A1
        200             /ONE PAGE, NO CONNECTED INTERRUPTS

        *200
        "X^100+"Y&3777  /TASK NAME IS XY
        601             /3 PAGES, ONE CONNECTED INTERRUPT
        PLOT            /DEVICE NUMBER OF PLOTTER
        INT             /ENTRYPOINT OF SERVICE ROUTINE
        ......
INT,    6502            ///CLEAR PLOTTER FLAG
        ......
```

N O T E

The first character of a taskname must be aphabetic (A-Z), to insure that the coded name is postive, as oposed to TCBPs.

The Task Header is followed by the Page Header, a list of pointers that have to be relocated, ending in a zero word. Task execution starts at the first word behind the Page Header.

Another example is the papertape reader task, a task that reads papertape with the highspeed reader in much the same way as the PTR: handler of OS/8 does. This handler makes use of the CONNECT feature, thus the interrupts are serviced by the task itself. For one request (call) it has to read several hundreds of characters. These can be stored immediatly in the buffer and there is no reason to start any task execution before the buffer is full. The interrupt that fills the last word of the buffer is declared a 'significant event', and SIGNALs the completion of the request. So only one in a few hundred interrupts gives rise to serious processor action, the others being handled by the interrupt routine only. Normally the execution time of interrupt routines is between 50 and 150 usec, while the activation of a task requires a multiple of this time. So the connect feature may reduce interrupt processing overhead considerably.

N O T E

During interrupt processing the cpu-status is only partially saved. Interrupt service routines may not disturb the MQ and EAE registers (A/B mode, stepcounter). They should not turn the interrupt system on (no ION) and may use only auto-index register 13 (AUTO13). Interrupt service routines may not use the CDTOIF instruction. Interrupt routines may NOT call the Monitor.

2.2 Multi-tasking

The MULTI8 foreground is a multitasking system. That means that all parallel executing programs are organized as 'tasks', well defined pieces of program that communicate with each other through primitives suplied by the Monitor. Inside a task no parallel execution takes place (appart from possible CONNECTed interrupt routines). A task may be viewed as a subroutine that is called in order to perform a specific function. Examples of tasks are the device drivers, the diagnostic timer, etc. A device driver task is called with a formatted set of parameters that specify what transfer has to be performed. The task interprets these parameters and controls its device accordingly. When the transfer is completed, the driver task will inform the caller of that fact.

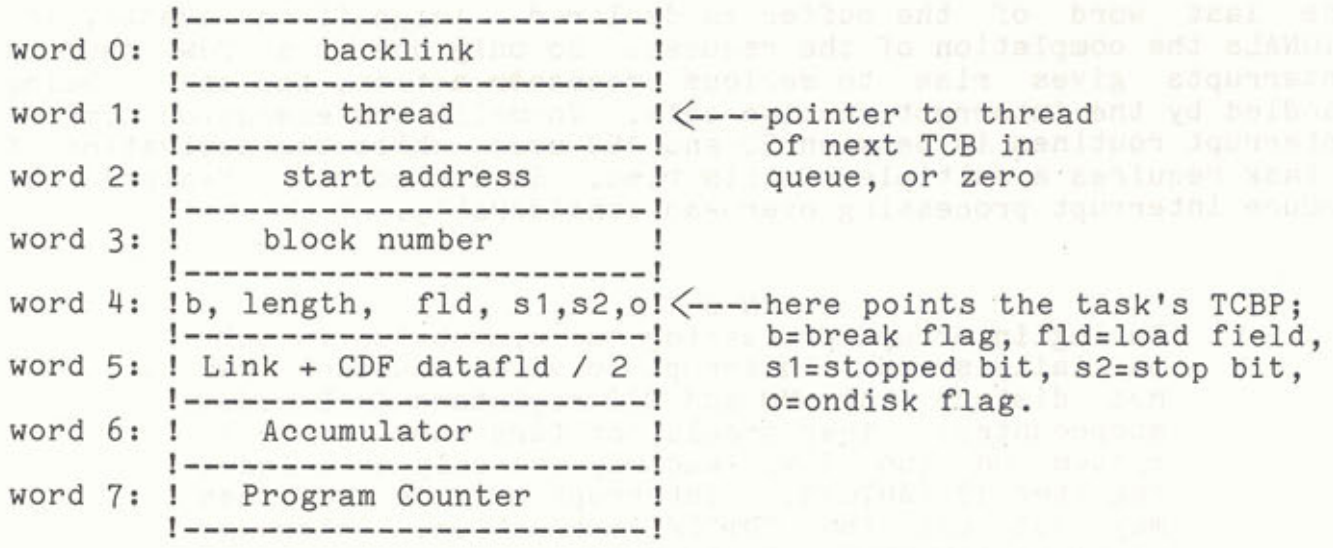
Some tasks are always called like subroutines, eventually returning control back to the calling task. Other tasks are started and execute in parallel with the task that initiated them. The distinction is

made by the way the task is called, not by the way the task is coded.

Frequently tasks have to synchronize their execution with other activities in the system, either peripheral devices or other software tasks. To this purpose the Monitor provides certain functions ('primitives') that enable a task to WAIT until a specified event has taken place.

MULTI8 tasks have to be coded in assembler language. External tasks are assembled by PAL8 and subsequently loaded by the MULTI8 task builder. This results in a tasklibrary on the system device, from which tasks can be readily loaded into memory when required.

Each task is identified by its Task Control Block (TCB), maintained by Monitor:



The general format of a Monitor call is:

```
        JMS MONITOR              /MONITOR ENTRY IN PAGE ZERO
           FUNCTION OPTION       /FUNCTION PLUS OPTIONAL OPTIONS
           PARAMETER             /SEE DESCRIPTION OF REQUESTS
         JMP ERROR               /ERROR RETURN
        .....                    /NORMAL RETURN
```

MONITOR is the Monitor entry point, duplicated in page zero of every foreground field. No CIF or CDF is required. Not all functions have a parameter, errorreturn or normal return. Note that the contents of MQ, stepcounter and A/B mode flipflop (EAE) are lost during a monitor call! AC, Link and datafield are retained, unless otherwise specified.

The following Monitor calls are provided:

RUN is used to start a task. The calling task continues.
CALL is used to call a task as a subroutine. The calling task is suspended until the called task has issued a RETURN.
RETURN is used to transfer control back to the calling task, or to

stop a task that was RUN.

STOP is used to suspend execution of a task. That task may later be continued by a RESTRT request.

SUSPND will stop the issuing task itself.

RESTART continues execution of a task that was previously STOPped. RESTRT has no effect when issued to a task that is not STOPped.

WAIT suspends task execution until a specified event is signalled.

SIGNAL is used to signal the occurence of an event, eg. to signal completion of an I/O request to another task.

RESERV returns an event number in the AC. May be combined with RETURN and CONTINUE. In that case both the issueing task and its caller receive the event number.

HALT is used to stop the task's execution. The difference between HALT and RETURN is that RETURN will enable execution of the calling task in case the task was called. HALT may be combined with SIGNAL, a construction that is often used in handler tasks.

PRECEDE has as only effect that the task is scheduled in MAINQ. It can be used to break very long computations to avoid monopolisation of the cpu.

BREAK is used to set a tasks breakflag, or to test-and-clear a tasks own breakflag.

REQBLK is used to request a buffer area from Monitor. These buffers consist of one or more consecutive memory pages.

RELBLK is used to release a buffer area formerly requested from Monitor.

Before going into detailed description of these requests, we will describe the possible combinations of requests and options. Options are special side-effects that may be enabled with certain requests. The following options do exist:

RELEASE will potentailly free the memory space the task is occupying.

SWPOUT will free the memory space the task is occupying. When the task is later activated again, a new immage will be read from disk. Note that the task image is never written back to disk.

CONTINUE will enable the task execution after a request that normally stops execution (eg. RETURN).

CLEAR will reset the task's backlink so it is no longer reserved for the previous caller.

Request - options:	RELEASE	SWPOUT	CONTIN.	CLEAR
RUN	no	no	no	no
CALL	yes	yes	no	no
RETURN	yes	yes	yes	yes
STOP	no	no	no	no
SUSPND	no	no	no	no
RESTRT	no	no	no	no
BREAK	no	no	no	no
WAIT	yes	yes	no	no
SIGNAL	no	no	no	yes
RESERV	no	no	yes	yes
HALT	yes	yes	no	yes
PRECEDE	no	no	no	no
REQBLK	no	no	no	no
RELBLK	no	no	no	no

SIGNAL and HALT may be combined. RUN is a combination of CALL with CONTINUE. EXIT is the combination RETURN CLEAR. RESERV may be combined with RETURN.

Another way to look at the various requests is how they influence the number of parallel executing tasks. The following requests will increase this number by one: RUN, RESTRT of a task that was STOPped or SUSPNDed, SIGNAL of an event for which a task was waiting. Requests that reduce the multitasking level are: RETURN of a RUN task, STOP, SUSPND, WAIT and HALT.

A task may be in one of the following states; Initialy it is inactive. Its Task Control Block and name are in the system tables, its image is in memory or on the disk, but it is not manipulated by Monitor as long as it is inactive. The TCB is not queued in any system queue. When the task is activated through a RUN or CALL request by another task (the first tasks are activated by the initializing program) it is queued in the main scheduler queue (MAINQ). The task is now active (though not actually possesing the cpu), its backlink is nonzero (=1 if run, =TCBP of caller +1 if CALLed). When elected by the dispatcher, it will receive the cpu, provided its image is in memory. In case it is not, the Monitor will create a temporary task that will bring the image in memory. During its active life the task may for some time be suspended, eg. when it WAITs for an event. Such a situation is characterised by the fact that the task is not threaded in any system queue, but a pointer to its TCB is stored in the event variable for which it is waiting. Its backlink will still be non-zero. Eventually the task may become inactive again as result of a RETURN or HALT request. Then its backlink will contain zero or the TCBP of the task that called it. In the latter case our task may only be called by the same caller again.

So the backlink plays a key role in determining whether a task may be RUN or CALLed by another task or not. For the RUN or CALL request to be succesful, the backlink of the called task must be either zero or equal to the TCBP of the calling task. Thus the backlink may have one

of four values:

=0 Task can be called by any other task.
=1 Task is active; can not be called by any task.
=TCBP task is inactive, can be called by previous caller only.
=TCBP+1 task is active, can not be called by any task.

It will be clear now that there are two separate mechanism operating on the backlink. A task can be reserved for one other task (eg. lineprinter driver task). The identity (TCBP) of the owner is then recorded in the backlink. The backlink is reset when the task uses the CLEAR option. When a task becomes active, its backlink is incremented by one, with the effect that no task in the system can call it because the backlink does not match any TCBP.

RUN

Parameter: name or TCBP of task to be RUN. In case a name is specified, Monitor will overwrite it with the TCBP. AC, Link and datafield are transferred to the called task. Both the calling and the called task are queued in MAINQ, with the calling task first. The callers AC, Link and datafield remain unchanged. The errorreturn is taken if 1) the taskname does not exist or 2) the task is busy or reserved by another task. In the first case the taskname is unchanged, in the latter case it is changed in the proper TCBP. When the errorreturn is taken, an implicit PRECEDE is executed. So it is legal to write a JMP .-3 on the errorreturn location, provided the taskname is known to exist.

CALL

Parameter is taskname or TCBP. If it is the taskname, it will be overwritten with the TCBP to prevent repeated lookup of the taskname. The issuing task is suspended until the called task executes a RETURN request. When the calling task continues, it has the AC, Link and datafield of the RETURNing task (just like a regular subroutine). Errorreturns are the same as with RUN. The options RELEASE and SWPOUT may be used when the calling task has no vital information in its immage. In that case the called task may overlay the calling task, depending on memory availability (see for example OD.TK).

RETURN

Return has no parameter. If the task was previously CALLed (not RUN), the current AC, Link and datafield are transferred to the calling task, which is scheduled in MAINQ. The issuing task stops, unless the CONTINUE option was specified. Both SWPOUT and RELEASE may be used to free memory occupied by the issuing task. The CLEAR option can be used to reset the backlink. If CLEAR is not used, the task can only be called by the same caller next time. It is illegal to use RETURN twice, eg. ... RETURN CONTINUE ... RETURN. For combinations of RETURN with RESERV see under RESERV.

STOP

Parameter is taskname or TCBP. AC, Link and datafield of issuing and target tasks remain unchanged. The STOPbit in the TCB of the target task is set. If it was set already, no action is taken. When the target task passes the dispatcher, its STOPPEDbit will be set and it will not be executed. If the taskname is not present in the system, the errorreturn is taken.

SUSPEND

The STOPbit and STOPPEDbit of the issuing task are set. The issuing task is not scheduled, and thus suspended until another task RESTRTs it. There is no parameter and no errorreturn.

RESTRT

Parameter is taskname or TCBP. The STOP- and STOPPED-bit of the target task are cleared. If the target task had its STOPPEDbit set, it is scheduled in MAINQ. AC, Link and datafield of both issuing and target tasks remain unchanged. If the taskname is not present in the system, the errorreturn is taken.

WAIT

Parameter is number of event to wait for. The task is suspended until a SIGNAL is received for that event. Because event flags may buffer one signal, the return may be immediate. A negative AC specifies a timeout value. It is the number of system ticks (normally .1 sec.) after which a timeout signal will be given by the timer task (TI). A positive or zero AC has no effect. Note that timeouts may only be used for permanently assigned events. If a timeout is received on an RESERVed event flag, and the normal signal is later received, that eventflag can not be used any more, because it will stay in the INTERRUPTED state. The exception is for eventflags specifically reserved to receive a timeout signal only, eg. by the STALL request.

After return from the WAIT request, AC 1-11 contains a code received from the signalling agency. Code 2 is reserved for timeout signals from TI, and a few others are used by the blockdriver protocol. Generally 0 means ok. RELEASE and SWPOUT may be used to free memory during the suspended period. Special feature: a negative eventnumber will result in an immediate return, with the AC set to the complement (!) of the eventnumber.

If the eventflag was already in the WAIT-state, ie. another task was already waiting for that same event, an immediate return is taken with the AC set to 1.

STALL

The STALL request is identical to the sequence RESERV a slot - WAIT with timeout. The parameter is the number of ticks that the task should be suspended. This value may be positive or negative. After

the specified interval the task will continue execution, WITH THE AC SET TO THE TIMEOUT CODE (=2). The original value of the AC is lost, but Link and datafield remain unchanged. The options RELEASE and SWPOUT may be used to free memory during the suspended period.

SIGNAL

Is used to signal the occurence of an event. The parameter is the eventnumber. The AC, bits 1-11 are passed to the WAITing task. AC, Link and datafield of the issuing task remain unchanged. SIGNAL can be combined with the HALT request. For options see at HALT.

HALT

This request has no parameter and no return. It is used to stop execution of a task that has already issued RETURN CONTINUE. HALT may be combined with SIGNAL to notify some other task of the completion of the issuing task. The RELEASE and SWPOUT options may be used to free memory. The CLEAR option may be used to clear the backlink of the issuing task.

RESERV

RESERV has no parameter. It allocates one of the free event slots in the monitor area and returns the number of this event in the users AC. The allocated event can be used for synchronization between two tasks for one time. To this end the issuing task must pass the number of the event he got to the task he wants to synchronize with. Then one of the tasks may WAIT for that event, and the other task can SIGNAL the event. Note that no timeout may be specified with events allocated through RESERV. Most of the MULTI8 blockdrivers use the combination RESERV RETURN CONTINUE to allocate an event slot and pass its number to the calling task in one operation.

PRECEDE

Because the taskscheduler uses a non-preemptive algorithm, tasks are never interrupted by other tasks between two Monitor calls. So the response time at the tasklevel depends on the length of computations between two successive Monitor calls in tasks. Normally tasks do very litle (often the execution of the Monitor requests takes more time than the taskcode in between), but there may be cases where a task has more than a few milliseconds continious computation to do, eg. a Fast Fourier Transform. If the other system activities request it, the response time at the tasklevel may be improved by splitting the long computation in two or more shorter parts, by inserting a PRECEDE request in an outer loop. PRECEDE has no parameter and no error return. AC, Link and datafield are retained, but the contents of MQ are lost.

BREAK

Has one parameter that may be a task name (or TCBP) or zero. If it is a task name, the break flag of that task is set. If the taskname is

not found in the system tables, the error return is taken. If the parameter is zero, the tasks own break flag is tested and then reset. If the break flag was set, the error return is taken.

REQBLK

Tasks may request pieces of memory to be used eg. as buffer area. Generally the programmer has the choice to include the buffer in his task, where they may be used to hold initialization code, or request them dynamically. The latter has the advantage that the task becomes split in several smaller parts that can easier be fit in memory. However, it requires some extra program logic. The RELBLK request has one parameter, specifying how many pages should be allocated. On return, the AC points to the first word of the buffer, and the datafield register is set to the field where the buffer resides. The contents of the buffer are undefined. If not enough space is available, the error return is taken. Note that the buffer is always one contigious area and will never cross a fieldboundary.

RELBLK

Is used to release a buffer area allocated with REQBLK. The users datafield should be set to the field of the buffer, and the AC should be an address in the first page of the buffer. On return, the users datafield is equal to his instruction field, AC and Link are reset. If the AC and datafield are illegal on entry, the task is aborted. Note that bufferareas are always released unconditionally, eg. like SWPOUT of a task.

2.3 Memory Allocation.

This section describes how the foreground memory area is managed by Monitor, how external tasks are loaded into memory, how they are relocated and how their space is released.

Of the memory fields reserved for the foreground, about 4K is used for the resident Monitor, data areas and the internal tasks. The remaining space is allocated for external tasks and buffers. This space is managed in units of one page (128 words). The memory map (CORMAP) contains one word for each page, with the following encoding:

0	Page is free
positive	page is potentially free, entry is -TCBP of task
negative	page is occupied, entry is TCBP of task, or
-3	page is permanently occupied by resident matter
-2	page is not last page of an allocated buffer
-1	page is last page of an allocated buffer

Requests for space are handled by the monitor routine HOLE. HOLE is called with the number of pages requested in the AC. The allocation algorithm is First Fit, ie. the first sufficiently large section of contiguious free pages encountered is allocated. HOLE will make two

scans over the coremap, if necessary. The first scan looks for free space (zero entries). If no sufficient free space is found, the second scan will look for free or potentially free space. Potentially free space (positive entries) consists of pages that hold an intact image of a task that is logically on the disk. When such a task is activated, the coremap is inspected to see if its image is still intact; In that case no diskread is necessary, only the coremap is updated to reflect the fact that the task is logically loaded now. So HOLE will avoid to load a new task over an image that can still be used, as long as enough free space is available. Whether a tasks image is reusable or not is determined by the task itself, that uses either the RELEASE or SWPOUT option to leave memory. Although the name SWPOUT might sugest the contrary, task images are never written back to disk. A task should only free its corespace if it has no sensitive data in its body. In practice a lot of tasks specify RELEASE with their RETURN or HALT request, although they are sometimes called many times per second. This costs very litle extra time, but ensures that a lot of room is potentially free for peak moments (see the core mape reproduced in section 3.2).

In order to anticipate a possibly large request, the Monitor tries to keep the active memory area as small as possible. This is partly effected by the First Fit algorithm, partly by the so called 'bubbling' stategy. When a task uses the RELEASE option, the map entry immediately preceding the tasks entries is inspected. If that page is free, the RELEASE is executed as SWPOUT.

```
$$  AA  AA  ..  BB  ..  ..      (BB ACTIVE. DOES RELEASE NOW...)
$$  AA  AA  ..  ..  ..  ..      (EXECUTED AS SWPOUT ...)
$$  AA  AA  BB  ..  ..  ..      (BB ACTIVE AGAIN, LOADED CLOSE TO AA)
```

As the example shows, a hole between the active tasks will 'bubble' up to the large empty area at the top of core.

When an external task is activated, the following sequence occurs: First, the TCB is inspected to see if the task is in memory or 'ondisk'. If it is in memory, it can be started right away. If it is on disk, the memory map is inspected at the entry corresponding to the page where the task was previously loaded. The latter is determined by the value in the Start Address entry of the tasks TCB. If the task image is still ok°, his coremap entries are negated and the task can be started on the old image. If the task image is not intact (perhaps the task used SWPOUT), HOLE is entered to allocate an area equal to the length of the task. HOLE maintains a count of the smallest request that could previously not be satisfied. If the new request asks for more, no scan is tried, and the error return is taken. In that case the task has to wait till more space becomes available. Its TCB is queued in COREQ. Each time a section of memory is released the entire COREQ is threaded onto the MAINQ. Thus all waiting tasks will again be considered. Normally HOLE will be able to allocate the requested space. Monitor will now create a temporary (nameless) task that issues the disk request to transfer the task into memory. The code executed by this nameless task is a reentrant part of the monitor. If more than one task has to be loaded at the same time, each one gets its own 'faketask'.

- - - - - -

° The taskimage is ok if the coremap entry corresponding to the tasks first page contains -TCBP.

IF CALLED TASK DOES RELEASE' CALLING TASK SHOULD NOT SWPOUT'

After reading the task image in memory, the relocation and the establishing if CONNECTed interrupts must be done.

MULTI8 tasks are both page-relocatable as well as field-relocatable. The former means that a task may be loaded at any page within a field (excluding page zero), the latter implies that a task may be loaded in any (foreground) field.

Page-relocation is attained by collecting all off-page intra-task references in a special section of each page, the 'page header'. This page header occupies the first locations of the page and ends with a zero word. When the task is loaded, Monitor will update all words in the page header to reflect the difference between the actual load address and the address where the task was assembled (200). After this relocation pass, there is no additional overhead due to page-relocation.

Field-relocatability must be ensured by the programmer, with some help of Monitor. The instruction CDTOIF can be used to Change the Datafield TO the Instruction Field. MYCDF, available on page-zero of every foreground field, contains a CDF to that field. ZMYCDF is a pointer to MYCDF, also available on page zero. So a methode to get a CDF to the current datafield is (if the datafield is one of the foreground fields):

```
        TAD I ZMYCDF    //LOAD AC WITH CDF TO CURRENT DF.
```

Rather than:

```
        RDF
        TAD (CDF
```

MYCIF and MYCDIF are available too, with their pointers ZMYCIF and ZMYCDIF. Another useful routine is DEFER, also on page zero:

```
DEFER,  0
        DCA X
        TAD I X
        JMP I DEFER
```

X is a temporary on page zero (in all foreground fields) that may freely be used by tasks. The restriction is, that X looses its value at every Monitor call.

CDTOAC is a page zero JMS that executes a CDF to the datafield conained in AC 6-8, eg.:

```
CDTOAC=JMS .
        0
        AND C70
        TAD C6201
        DCA XACCDF
XACCDF, CDF
        JMP I .-5
```

In addition, page zero contains 7 scratchpad locations (ZTEM1, ZTEM2, ... ZTEM7), the auto-index registers (AUTO10, -11, -12, -14, -15, -16, -17) and 27 useful constants: C2, C3, C4, C7, C17, C37, C70, C77, C100, C177, C200, C212, C215, C240, C260, C3700, C6201, C7000 (=M1000), C7400 (=M400), M215, C7600 (=M200), C7700 (=M100), C7770 (=M10), C7771 (=M7), C7774 (=M4), C7775 (=M3), C7776 (=M2), C7777 (=M1).

N O T E

AUTO13 is reserved for use by interrupt service routines and should never be used by tasklevel code.

2.4 Writing a Task

The first stage in writing a task is a careful planning of the setup for the problem at hand. Of course it is possible that your problem can be solved by one single foreground task, but this is seldom the case. Most problems can be separated in a real-time part, that involves some time-critical actions and one or more control or compute parts that are less time bound. The general philosophy is to do as much as is feasable at the highest level of the system, eg. in the background. There time and memory space are cheapest. One of the following reasons may exist to program part of the problem in the foreground: 1) Time-critical interactions with peripheral apparatus, 2) problem has to run for long times and can be satisfied by just a few pages in the foreground.

Most likely your program involves user-interaction, realtime data-aquisition, computation and output of results. This kind of situations can best be tackled by a combination of background and foreground programs. Initially the background program performs the interaction with the user, requesting parameters, etc. Next, the background program activates a foreground task, passing whatever parameters are necessary. The foreground task performs the realtime functions, relaying the raw results to the background program (either via a diskfile or directly into memory). Finally the background program performs computational tasks and presents the results to the user.

The advantages of this approach are that the larger part of the programming can be done in a protected environment, during normal timesharing operation of the system. Also, the background program can use all the features of the OS/8 environment. In fact, it can be programmed in one of the highlevel languages available. Just add a small PAL8, MACREL, SABR or RALF routine to interface with the foreground. Only a small part of the problem, usually just one or a few pages, have to be programmed in assembler and debugged with the system shut off for normal operation. In this way the functions of the foreground and background are clearly separated; modification of the computational, input or output part of the program can easilly be accomplished, without disrupting timesharing service.

As an example we will investigate the following situation. Some special device has been coupled to the machine with a digital

input/output interface. When the operator presses a button on the device, it starts some measurements and delivers 16 words, one word every 30 seconds. For each word an interrupt is generated by the interface. Our program should read these 16 values, perform some computation on it and typeout the results on the terminal. Also, it should write the results in a diskfile for later analysis. Our main program will be written in Fortran II. It starts with requesting the name of the diskfile to be produced, the run number, etc.

```
      DIMENSION IBUF(16)
      ........
      READ(1,100) INUMBER
100   FORMAT(' RUN NUMBER= ',I4)
      ........
```

Next the foreground task will be activated to take the 16 data words. The operator is requested to press the button that starts the measuring device.

```
      ........
      WRITE(1,110)
110   FORMAT(' PRESS BUTTON !')
```

Next comes a section of SABR code. It activates the foreground task AQ that takes 16 readings and stores the values in the array IBUF.

```
S       TAD (CODE       /GET CODE OF MEASUREMENT TASK IN AC
S       CPAGE 3         /NEXT 3 INSTRUCTIONS MUST BE CONSECUTIVE !
S       6770            /THE TRAP !
S       SKP             /JUMP OVER THE PARAMETER
S          \IBUF        /PARAMETER FOR AQ (ADDRESS OF ARRAY)
```

CODE is a small integer (assign from 20 upward) that selects the proper function in the foreground. The 6770 is the so called 'giant IOT'. It is trapped and the Central Emulator (see section 4.2) will call a task, dependent on the value that is in the users AC at that time. After completion of that task, the background program is continued at the location following the 6770.

At this point the array IBUF contains the 16 readings from the device. The rest of the Fortran program is straightforeward.

Now we come to writing the foreground task. Each (external) task starts with a preamble, followed by a task header, followed by a page header. Every following page of the task starts with a page header only. Tasks always assemble from *200 upward. The preamble (which can be empty) is used to instruct the taskbuilder to store certain values in certain system tables in MULTI8. The preamble is only used at task-build time and is not part of the task image (cf. the build-format for OS/8 handlers). In this particular case the name of our task should be inserted in GIGATB, the table that is used by the interpretation of trapped 6770 instructions:

```
      *0                /MANDATORY FOR EACH ENTRY IN THE PREAMBLE
      CDF 10            /ALL BG-RELATED TABLES ARE IN FIELD 1
      GIGATB+CODE       /INDEX GIGATB WITH 'CODE'
      "A^100+"Q&3777    /DROP THE NAME OF THIS TASK THERE
```

The first word of the task header is the task name (2 characters):

```
/DATA AQUISITION TASK. NAME=AQ. SERVES TO INPUT 16
/VALUES FROM ***** DEVICE. IS CALLED VIA A GIANT IOT
/FROM THE BACKGROUND PROGRAM.  PARAMETER: ADDRESS OF ARRAY
/IN USER PROGRAM WHERE THE VALUES HAVE TO BE STORED.

        *200

        "A^100+"Q&3777  /TASK NAME IS 'AQ'
        200             /ONE PAGE, NO CONNECTED INTERRUPTS
```

The second word of the task header specifies the length of the task, 200 for 1 page, 400 for two pages, etc. Also, the loworder bits specify the number of connected interrupts (zero in our case). We assume that an entry in the skipchain has been created that fields the interrupt of our device. The event signalled by these interrupts will be called INT (see section 2.1 for the skipchain).

The page header contains pointers that must be relocated by the monitor. The page header ends with a zero.

```
AQBUFA, BUFFER          /ADDRESS OF BUFFER. WILL BE RELOCATED.
        0               /ZERO IS END OF PAGE HEADER
```

This task is called by the central emulator with datafield=1, AC=pointer to the data-area of the current background and the link is cleared. The status of the background is EMULATE, and its instruction field is known to be in memory.

```
AQ,     DCA AQBG        //POINTER TO BG AREA, DF=1

/INITIALIZE SOME POINTERS ...
        TAD (-20        //20 (OCTAL) VALUES TO COME
        DCA AQCNT

        TAD AQBUFA      //ADDRESS OF INTERNAL BUFFER
        DCA AQPNT       //TO POINTER
```

Now we can start taking data from the device. This will last approximately 8 minutes. During this time the background program can be swapped to disk to free background memory for other users. This is possible because this task will buffer the 16 data words in the foreground. The background program can only be swapped completely when its status is INACTIVE. Also, we have to signal the background scheduler that its status has been changed to ensure prompt reaction.

```
        TAD I AQBG      //GET USERS STATUS REGISTER
        TAD (INACTIVE-EMULATE //SET HIM INACTIVE, CLEAR EMULATE
        DCA I AQBG      //SET BG INACTIVE
        JMS MONITOR     //SEND SIGNAL TO BACKGROUND SCHEDULER SO HE
           SIGNAL       //WILL NOTICE THAT THIS USER IS INACTIVE NOW.
           BSSLOT       //(THE PRIVATE EVENTNUMBER OF THE B.SCHED.)

/WAIT FOR DATA NOW ...
        CDTOIF          /SET DF TO THIS FIELD (WAS FIELD 1)
```

```
AQLOOP, CLA CLL CML RAR /AC=4000, LONGEST TIMEOUT POSSIBLE (204.8 SEC)
        JMS MONITOR     /WAIT FOR READING NOW
           WAIT
           INT
        SZA CLA         /TIMEOUT ?
         ISZ AQERR      /YES, DEVICE IS HANGING.
        SKP
         JMP AQOUT      /TAKE ERROR EXIT
        6146            /READ DATA
        DCA I AQPNT     /STORE VALUE IN INTERNAL BUFFER
        ISZ AQPNT       /BUMP BUFFER POINTER
        ISZ AQCNT       /16 VALUES DONE ?
         JMP AQLOOP     /NO, MORE TO COME
```

Now the internal buffer is filled with 16 data words. So it's time to get the background program in memory and move the data. At this point, we do not know if any of the background program's fields is in memory. Moreover, the background scheduler can at any moment decided to remove a field of our program. We should therefore request the background scheduler to bring the field that we need, the field were the user's buffer is located (ie. his current instruction field) in memory if it is not already there, and then it should stay there until we finish copying the data. The procedure is quite simple. We change the status of our bg from INACTIVE to INCORE, and set the number of the virtual field we need in bits 6-8 of the state word. Then we send a SIGNAL to the background scheduler.

```
AQOUT,  CDF 10          //MUST ACCESS TABLE IN FIELD 1
        TAD (UFLDS      //GET VIRTUAL INSTRUCTION FIELD
        TAD AQBG        //
        JMS DEFER       //
        AND C70         //IN BITS 6-8
        TAD I AQBG      //ADD INTO BACKGOUND STATUS
        TAD (-INACTIVE+INCORE //CLEAR INACTIVE, SET INCORE
        AND (-LONG-1     //CLEAR LONG TO GET SOME PRIORITY
        DCA I AQBG      //
        JMS MONITOR     //SIGNAL BACKGROUND SCHEDULER THAT OUR STATUS
           SIGNAL       //IS CHANGED.
           BSSLOT       //
```

As soon as possible the background scheduler will transfer the requested field into memory. Then he will send a signal to the private event of this background to let us know that it's present now. Of course we should wait for this signal before accessing the background memory. The number of the real memory field were the requested virtual field has been loaded is passed as status with the SIGNAL from the background scheduler, in bits 6-8 again. The background scheduler has changed the state of our background to EMULATE, which insures that the field we just obtained will not be removed from memory.

```
        TAD (USLOT      //COMPUTE ADDRESS OF USERS EVENT NUMBER
        TAD AQBG
        JMS DEFER       //GET EVENT NUMBER
        DCA AQEVENT     //STORE IN WAIT-REQUEST

        JMS MONITOR     //WAIT FOR SIGNAL FROM BACKGROUND SCHEDULER
           WAIT         //THAT THE FIELD IS IN MEMORY NOW
```

```
AQEVENT,   0            //SET TO NUMBER OF BACKGROUND'S EVENT
        TAD C6201       //FIELD IS IN CORE NOW
        DCA AQCDF       //CDF TO USER'S FIELD
```

The background scheduler has cleared the INCORE bit, and set EMULATE. Next we copy the data words from our internal buffer to the buffer in the background program. The address of the target buffer is at AQARG in the background instruction field.

```
        ISZ AQERR       //DID WE HAVE ERRORS ?
         JMP AQEXIT     //YES, QUIT

        TAD (UPC        //COMPUTE ADDRESS OF BG'S PROGRAM COUNTER
        TAD AQBG
        JMS DEFER       //GET USERS PROGRAM COUNTER
        IAC             //NOW POINTS TO FIRST PARAMETER
        JMS AQUCDF      //ACCESS BACKGROUND FIELD
        JMS DEFER       //GET ADDRESS OF BACKGROUND ARRAY
        TAD M1          //LESS ONE FOR AUTO-INDEX
        DCA AUTO10      //SET UP AUTOINDEX POINTER TO USERS BUFFER

        CLA CMA         //AC=-1
        TAD AQBUFA      //RELOCATED POINTER TO INTERNAL BUFFER
        DCA AUTO11      //SETUP AUTO INDEX POINTER IN INTERNAL BUFFER

        TAD (-20        //
        DCA AQCNT       //SET UP COUNTER FOR 16

AQMORE, CDTOIF          /ACCESS THIS FIELD
        TAD I AUTO11    /GET A VALUE
        JMS AQUCDF      //ACCESS USERS FIELD
        DCA I AUTO10    //STORE VALUE IN USERS BUFFER
        ISZ AQCNT       //16 WORDS DONE ?
         JMP AQMORE     //NO, MORE TO DO
```

Done! Return to the central emulator, that will continue the background program at the instruction behind the 6770. Note that the AC must be zero here, otherwise the central emulator will assume that some emulation error has occurred and stop the background program.

```
AQEXIT, JMS MONITOR     //FINISHED, RETURN TO CENTRAL EMULATOR.
           EXIT SWPOUT  //NO REASON TO STAY IN CORE.

AQUCDF, 0               /SUBR. TO SET DF TO USER'S
AQCDF,  CDF             //CDF TO BACKGROUND'S INSTRUCTION FIELD
        JMP I AQUCDF    //

AQBG,   0               /POINTER TO USERS BACKGROUND DATA
AQCNT,  0               /COUNTER
AQERR,  -1              /ERROR FLAG, -1 IF NO ERROR

AQPNT,  0               /POINTER IN INTERNAL BUFFER
BUFFER, ZBLOCK 20       /16 LOCATION INTERNAL BUFFER.

$
```

This task should be assembled with the monitor definitions file, MS.PA (see 5.2) and then loaded in the system:

```
.PAL AQ<MS,AQ.TK
ERRORS DETECTED: 0
LINKS GENERATED: 0

.R MULTI8
*AQ$
```

A more formal description of the interactions between emulator tasks, the central emulator and the background scheduler is given in section 4.5. For examples of more complex interaction between a foreground task and a background program you should study the emulator tasks, eg. TE, FE, PE, LE and RE.

- o - o - o -