3. System Tasks

This chapter contains two sections, one on internal system tasks and one on external system tasks. The primary distinction between the two is whether the task resides in memory or on disk. External tasks are assembled separately and are loaded in the disk-resident tasks library by the taskbuilder program (see section 6.2). Internal tasks are assembled as a part of the monitor source and loaded and saved along with Monitor. Because internal tasks have no diskaddress (blocknumber), there is one word spare in their Task Control Block. This word is often used to store the base address for reentrant tasks. Consequently only internal tasks can be reentrant. Note that there is no indication in the TCB whether the task is external or internal. Internal tasks should never issue RELEASE or SWAPOUT requests.

In principle internal tasks can be assembled separately also. However, the programmer has to specify where in memory the task must be loaded and should update the various system tables (memory map, task name table and task control block table).

For the majority of standard computer peripherals driver tasks have been written that adhere to one common calling protocol. These tasks are called 'blockdrivers', as they perform I/O in units of one or more blocks. A block is a sequence of 256 (decimal) 12 bit words. The blockdriver protocol supports direct addressable devices, but can also be used for sequential access devices like papertape, etc.

Here is a sample call to some block driver task 'BD':

```
START,  CDTOIF          /SET CORRECT DATAFIELD
        TAD (PARAMS     /LOAD AC WITH POINTER TO PARAMETERS
        JMS MONITOR
           CALL
TNAME,     "B^100+"D&3777 /CALL THE TASK 'BD' (BLOCK DRIVER)
         JMP BUSY       /ERROR RETURN, TASK WAS BUSY
        DCA EVENT       //TRANSFER IS INITIATED. AC HOLDS EVENT WHERE
                        //COMPLETION WILL BE SIGNALLED
        CDTOIF          /RESET DATAFIELD (!)
        ......          /PERFORM OTHER OPERATIONS WHILE I/O PROCEEDS
        JMS MONITOR
           WAIT
EVENT,     0            /WILL GET EVENT NUMBER GOT FROM 'BD'
        SZA             /ERRORS IN TRANSFER ?
         JMP ERROR      /YES, AC CONTAINS ERROR CODE
        .....           /NO, OPERATION COMPLETE

PARAMS, FUNCTION        /WLL.LLL.FFF.UUU
        BUFFER
        BLOCKNO

BUSY,   CLA             /(!)
        TAD TNAME       /GET REQUEST PARAMETER
        SMA CLA         /WAS THE NAME FOUND ?
         JMP NOTASK     /NOT CHANGED, UNDEFINED TASK NAME
        JMS MONITOR
           STALL
           ...          /WAIT A SUITABLE TIME
        CLA CLL         /CLEAR AC !!! (AC=2 AFTER A STALL)
```

```
        JMP START       /RETRY OPERATION

ERROR,  TAD (-EOMERR
        SZA CLA         /END OF MEDIUM ?
        .....           /NO, TROUBLES !
        .....           /YES, DONE
```

Note that the address of the transfer vector (PARAMS) should be relocated in case this code is part of an external task!

Some blockdrivers (eg. KR, LP, PP, PR, PL) need a 'CLOSE' call after the end of the file transfer in order to allow other tasks to call them. All blockdrivers accept the close call. A close call is a call with AC=0. Many blockdrivers will return immediatly, with the AC set to -1. This value makes the following WAIT request a dummy.

3.1 Internal system tasks.

Internal system tasks are:

SY	the system disk blockdriver task
TI	the timer task
Tn	terminal output handler for terminal n
Kn	terminal input handler for terminal n
En	central emulator for background n
In	input reader for background n
On	output writer for background n
DK	VIRTUAL disk mapper

The central emulator, input reader and output writer tasks are described in chapter 4.

SY

SY is the driver task for the system disk. The source of SY is contained in M2.PA. SY consists of two parts, the task-part and the interrupt service routine. The task part is activated by a CALL from another task, specifying the address of a three-word parameter block. The taskpart will enter this pointer in a queue which is emptied by the interrupt service routine. Both parts have one subroutine in common, SYSDO, which is used to start a disk transfer.

On entry of the task-part the AC+DF is a pointer to the parameters:

```
Word 0: wll.lll.fff.uuu=function word (r/w,length,field,unitnumber)
Word 1: memory address of buffer
Word 2: disk block number
```

All types of disks are addressed by 256 word blocks, starting with block 0. The transfer length is specified by 5 bits (00-37), indicating the number of half-blocks to transfer. A transfer always starts at the beginning of a block.

N O T E

A specified transfer length of 00 will result in a 4K transfer.

The internal queue contains two-word entries:

Word 0: fff.00e.eee.eee f=field of parameter block, e=event number
Word 1: aaa.aaa.aaa.aaa a=address of parameter block

Through conditional assembly the interrupt service routine is adapted to the type of disk device.

TI

The timer task is RUN when the system starts and keeps running. It uses one of the various types of clocks to implement the timeout facility. TI consists of an interrupt service routine (located in M1.PA, immediately behind the skipchain) and a task-part (in M2.PA). The interrupt part receives the primary hardware interrupts and counts down to a rate of DGNTICK (normally 10Hz). At this rate SIGNALs are sent to the TIMER event. The task-part is WAITing for these signals and will scan the list of event flags (HRDLST). Entries in HRDLST are two words each:

Word 0: if negative: - timeout count
Word 1: state (WAITING - FREE - INTERRUPTED - RESERVED).

If a certain entries' word 0 is negative, it is incremented. In case of overflow a SIGNAL is sent to that event number, with the TIMEOUT code (=2) as status. When TC08 DECtape is part of the configuration, TI will scan all tape drives every 5 seconds. Any drive that is found to be off-line is marked in the tape table (TAPETB). This table keeps track of the block number under the tapeheads and is used by the DECtape emulator to determine whether to swap a users' program or do a direct transfer. If an entry in TAPETB is zero, the current tape possition is unknown and the DECtape emulator will declare the background inactive during the possibly long search. TAPETB is maintained by the dectape driver task (DT), that uses TAPETB to determine the optimum start direction for the tape drive.

Static electricity sometimes makes that the interrupt enable flipflop of KL8E type terminal interfaces flops by accident. In that case the terminal falls dead, unless the bit is flipped again. Therefore TI will every 5 seconds set all terminal interrupt enable bits. In the same way the lineprinter interrupt is periodically enabled. If the LE8E interface senses an error-condition in the printer, it raises the error flag that causes an interrupt. This flag however, is in fact a level, dat can only be cleared by operator intervention. Therefore the system has to disbale the lineprinter interrupts when the error flag is set. To restart the printer after the error condition has been removed, the interrupt has to be re-enabled, which is thus done by TI.

At each clock tick TI examines the status of the background BJOB points to. If it is runnable, TI increments its (double precision) account register at UACCNT.

N O T E

This mechanism gives only an approximate measure of the cpu-time used by each background. All foreground activity is charged to the running background.

Tn

Both Tn and Kn come in two flavours, reentrant and non-reentrant. The reentrant versions are about two times larger than the non-reentrant versions, but can service a mixture of KL8E and KL8A interfaced terminals. If more than one KL8E or an KL8A are present, the reentrant version will assemble. Tn (actual task names are T1, T2, etc.) is called with one ASCII character in the AC. This character will be sent to the terminal. TABs are expanded to 1-8 spaces. A number (TnFILL) of filler characters (NULL) may be sent after some special character (TnCHAR), as specified in the configuration file. On entry NULL-characters are ignored. BACKSPACE may be translated to any suitable character as defined by the parameter TnBACK. On entry, the most significant AC bit specifies whether Tn should RETURN or EXIT. Foreground tasks should follow the following protocol when using the Tn/Kn tasks:

Always start with outputing one or more characters. These characters should have bit 0 set, to ensure that Tn will RETURN, so that no other task can interrupt your output. Once you have got control over Tn, you may do input via Kn. You have to supply the echo characters, which should have bit 0 set too. Only the last output character of your task should have bit 0 zero, to release the Tn task.

Kn

For reentrancy see at Tn. Kn (K1, K2, etc.) is called to read a character from the keyboard. On entry a negative AC is interpreted as a timeout value. If no character is received within this time, the value 4000 is returned. Incoming characters get their parity bit forced on. A configuration-specified code (TnESCP) will be converted to ESCAPE (233).

DK

The task DK is only included in systems that use non-standard userdisk allocation. Its function is to convert virtual disk requests to real disk requests. DK does so, based on the contents of a table describing the allocation of user disks. For each 'unit' of DK, this table contains four parameters: the actaul driver task name (eg. SY), the actual unit number, the offset of virtual block 0 and the length of the virtual disk. DK checks that the request falls entirely inside the allocated area and then passes a converted request to the proper driver task.

3.2 External System Tasks.

MULTI8 is furnished with a set of external system tasks. In this section the purpose and use of these tasks will be described.

BE.TK (Background Errorprinter) is normally called by either the keyboard input task (In) or the CONTROL/B task (CB). The former is triggered by an emulation error (illegal instruction, eg. HLT), the latter results from a Where-command entered by the user when the terminal is in CONTROL/B mode. On entry BE expects the AC to point to the dataarea of one of the backgrounds. BE will print a status message on the terminal belonging to that background. The format of the message is:

```
PC=02727 AC=00000 DF=0 MQ=1563 GT=0 TRAPPED 6031
```

The message is preceded by a 'bell' character (CONTROL/G). BE drops its output characters in the outputbuffer of the relevant terminal by calling the routine FILLQ in the central emulator (see section 4.2).

BS.TK (the Background Scheduler) is described in section 4.7 .

CB.TK (the CONTROL/B command decoder) is part of the background support package. When a background terminal is in CONTROL/B state, the Input Reader task will assemble one line of input from the keyboard. As soon as the CR is received, CB is called, with the pointer to the backgrounds dataarea in the AC. Now CB will read the line from the input buffer (through the MTQ routine in the Central Emulator) and interpret the command contained therein. The first character on the line selects the command, and any octal digits found in the line are assembled to one 12-bit value. Most of the commands involve only modifications of background registers and are done by CB itself, but the command WHERE requires eleborate action and is thus passed to the task BE. The commands recognized by CB are described in the MULTI8 Terminal Manual.

The BOOT and RESTORE commands require some further explanation. BOOT and RESTORE both consist of reading some code, prepared when the system was started, into virtual field 0 of the background. This code is then executed by the background. Only the entrypoints for BOOT en RESTORE differ.

For BOOT, the program reads block 0 of the users' disk and therefrom initializes the resident OS/8. This insures that all the settings that the user might have made (eg. SET TTY SCOPE, OPEN DSK3) remain in effect over the bootstrap operation.

In the case of RESTORE, a fresh copy of OS/8 is read from DSK0: and copied to the users' SYS:. The users' channel table is reset, and then an OPEN DSK0: is effected. Finally the normal operations of BOOT are executed. This is sufficient to

bring a virtual machine back in operation, even if its' memory and disk contain pure garbage. During a RESTORE, the directory on the users' disk is inspected. If it looks like a non-system directory, it is destroyed, to prevent the user from overwriting his monitor. When the system is started, all terminals are RESTOREd.

CR.TK CR.TK is supplied as an alternative to KE/KR. CR is a combined emulator/driver for the cardreader. Although it is slightly slower than KE/KR, it has some advantages: because CR merely emulates the cardreader IOT's, the SET CDR 026/029 commands remain in effect. Further, the foreground memory requirement of CR is one page less than the KE/KR pair. Modifications to CR can easily addapt it for marksensing cards (replace the RCRA (6632) instructions by rcrb (6634)). Note however, that the use of KE/KR avoids a nasty OS/8 problem with Fortran IV and MINBOL programs; These language processors shuffle around with handlers and may thereby lose information from the cardbuffer in CDR:.

To install CR.TK, remove KE/KR, and delete a line from M5.PA. In the table DEVLST: insert a slash before 'DEVICE CDR; 2030; 60' TO PREVENT THE REPLACEMENT OF CDR: BY A FAKEHANDLER ENTRY.

CD.TK (the foreground command decoder) is one of the first tasks developed for the MULTI8 system. It gives the system manager access to the foreground by enabling him to RUN, BREAK, STOP and RESTART tasks. CD itself is RUN by the initialization program. Once active, it will suspend itself with a WAIT SWPOUT request, specifying event 0. Event zero is reserved for this purpose and is signalled as soon as the breakcharacter is entered on any keyboard connected to the system. The breakcharacter is CONTROL/F by default, but can be changed by specifying BRKCHR=XXX. When the breakcharacter is typed, CD will receive a signal with the name of the keyboard task belonging to that terminal in the AC. (Eg. if the breakcharacter is entered on terminal 3, CD will get the value "K^100+"3&3777 in the AC). This enables CD to 'talk' to the terminal it was called from. CD starts with printing ^F,CR,LF,F> and next awaits one of the following commands:

```
F>R IT          (Run the task IT)
F>B ABCDEFG     (Set the break-bit for task AB)
F>S SP          (Stop the task SP)
F>C SP          (Continue (=RESTRT) the task SP)
```

Only the first letter of the command, and the first two letters of the taskname are significant, so

```
F>RUN STATUSDISPLAY
```

is legal too. The task receives the name of the keyboard task in the AC. In that way a task like IT or MA is able to communicate with the terminal it was called from. An illegal command, taskname or other failure will result in

```
F>R QQ
?
```

After executing the command, CD will return to the WAIT SWPOUT request and thus be sitting on the disk again. CD is mostly used to RUN tasks like MAP and ODT in the process of debugging new tasks. At most installations there will be no need for the normal user to use or even know the breakcharacter.

DI.TK (the directory lookup task) can be used to perform a LOOKUP in an OS/8 structured directory. DI should be called with the AC and data field a pointer to a list with the following structure:

```
"S^100+"Y&3777          /NAME OF DEVICE DRIVER
1                       /UNIT NUMBER
FILENAME MYFILE.XY      /FILENAME IN STANDARD FORMAT
```

After return from DI, this list is changed to:

```
"S^100+"Y&3777          /NAME OF DEVICE DRIVER
1                       /UNIT NUMBER
LENGTH                  /-LENGTH OF FILE
0
BLOCK                   /FIRST BLOCKNUMBER OF FILE
```

At this point the datafield is equal to that before the call. If the AC is non-zero, an error has occured (device busy, file not found, etc).

DS.TK Is the emulator for the public disks. At some installations it is required that certain disks (eg. RKA1 and RKB1) can be read and writen by all users. The access to such disks is governed by DS. DS will claim the device for the user that starts using it and holds it as long as that users' program runs. This prevents problems with the directory. If you are sure that the way these disks are used in your installation avoids problems with multiple updates, you can change the task source and define the symbol 'DANGER'. If DANGER is defined, the task always EXITs, so that requests from different users can be interleaved.

DT.TK (the TC01/TC08/TD8E DECtape blockdriver) is a normal blockdriver task, transferring blocks of data to and from the DECtape. The one deviation of the blockdriver protocol is that a transfer length of zero pages will not transfer 4K, but zero words instead. This feature is used by the Tape Emulator to position the tape before locking the background target field in memory. For TC08 the event DTA should be connectable. DT will automatically determine the correct start direction. For TC08, a blocknumer with bit zero set (4000-7777) makes DT to read 256 18-bit word blocks (as used by PDP11, PDP9, PDP15 and PDP10).

FE.TK (floppy emulator) handles IO requests for the floppy disk passed through the fakehandler. FE calls the floppy disk block driver, RX. If the symbol DANGER is defined in the source of FE, users can interleave requests to the floppies. This is the default mode. If DANGER is not defined, the floppy disk system is claimed by the first user that uses it,

just as is the case with the lineprinter.

GE.TK (Graphics Emulator) is the emulator for the XY8E or KLPLOT plotter (see also at PL.TK). For XY8E, GE handles trapped plotter IOT's. It extracts the plotter direction and pen-information and assembles 12-bit plotter words, 6 bits for the step and a 6-bit repetition counter. These words are packed in a 256-word buffer and sent to PL, the plotter driver.

IT.TK (initialize time and date task) will be demonstrated by an example:

```
F> R ITIME
TIME=9:12(CR)          set system time hour:minutes
DATE=4/20/77(CR)       set date month/day/year
```

If you do not want to change the current date, you can answer the second question with Return. IT will automatically run on terminal 1 when the system is started.

KE.TK (cardreader emulator) passes calls from the fakehandler to the cardreader blockdriver (KR).

KR.TK (cardreader blockdriver) reads cards and converts the holerith punch code to ASCII characeters. The characters are packed into the user buffer. If an end-of-file card is read, a ^Z is placed in the buffer and the rest of the buffer is zeroed. If the cardreader times out (eg. no more cards), KR hangs until more cards are loaded. Only an end-of-file card stops KR.

LE.TK (the lineprinter emulator) is one of the most complicated task in the system. There are two different ways to output to the lineprinter, through the OS/8 lineprinter handler (which is replaced by the fakehandler) and through direct lineprinter IOT's. The latter is used eg. by the FORTRAN runtime package. When a program issues its first lineprinter output, the emulator task will determine what kind it is and adapt itself to that mode of operation. Actually, the task contains two pieces of code, one for emulating simple IOT's, and another for transferring whole blocks at a time. When emulating IOTs, the emulator will perform an EXIT and thus release the device when a ^Z is found in the datastream, or when it is called with the link set, signalling that the background program has read the Keyboard Monitor or the Command Decoder into its memory. After initialization, only the first page of the task contains code, the rest (2, 4 or 8 pages) is used to build large buffers (one or two, depending on the specification in the LE source. After filling the buffer, the lineprinter blockdriver is called to transfer the data to the printer. If LE is processing calls through the fakehandler, it treats ^Z as end-of-buffer indicator. This is necessary for certain existing application programs (eg. the MINBOL package) that use this feature to send variable-length records to the printer. Only a regular OS/8 close call (zero length) or a CALL with the Link set do generate an end of file. If LE is in IOT mode and a fakehandler call is made, it closes the current file and returns to the central emulator with the

trapped instruction in the AC. As the instruction (6000) has a negative value, the central emulator will assume that this is a 'replacement' for the original instruction, patch the trapped instruction (which does not change anything in this case) and than decrements the users' PC. Then the background program is started again. It will execute the same instruction again, but now a fresh copy of LE is obtained, which initializes itself to the new mode of operation. The same method is used to switch from fakehandler calls to IOT emulation. All this is to make it possible to run BATCH with both the log and some program output going to the lineprinter. Note that each mode of operation will start on a new page.

LI.TK (the line input task) is used to read a line from one of the terminals (by foreground tasks). LI is called with in the AC and datafield a pointer to a linebuffer. On return, the datafield is still the same. The AC is zero, unless the input was terminated by an illegal control character (eg. ^Z). The linebuffer consists of a two-word header followed by the buffer itself:

```
"K^100+"2&3777  /NAME OF INPUT HANDLER
-40             /MINUS BUFFER LENGTH
ZBLOCK 40       /THE BUFFER
```

LI may also be used to output a line, possibly followed by input. Any characters in the linebuffer will be output by LI before input is solicited. The output string must end with a null. If the last character of the output string is CR (215), then no input is requested. During input RUBOUT (delete a character), CONTROL/U (delete line) and LINEFEED (retype line) are effective. The input line may be terminated by CR or ESCAPE. The input is placed in the linebuffer, starting at the first word after the header, one character per word.

LP.TK (the lineprinter blockdriver) is functionally equivalent to the OS/8 lineprinter handler. The value of the symbol LP8E determines the maximum line width (bit 2-10), whether the printer can handle lower case characters (bit 0), whether it is an LS8E (bit 1), KL8E (bit 3) or DKC8AA (bit 11) control (which have sligthly different instructions). If spooling is selected (bit 2=0, the normal mode), LP transfers all data to the file SPOOL.LP on DSK0: (actually, the device and unit number can be changed by assignments in the source of LP). If the lookup of SPOOL.LP (through DI) fails, LP generates a fatal error, which is passed by LE as either a handler error (through the fakehandler), or an emulation error (with 'TRAPPED' 6666, the lineprinter IOT). Within the body of LP is code for a second task, that reads the data from the spoolfile and transfers it to the printer. TABs are expanded, and ESCAPE is translated to '$'. If a line overflows the printer width, a CRLF is inserted. To conserve some of the earths tree population the ouput of empty pages is prevented.

MA.TK (the memory map printer) may be RUN from any MULTI8 terminal (through CD). It will display the actual foreground memory map. Entries labeled $$ are pages occupied by Monitor, AB is

task AB, -AB is RELEASed by task AB, ((is a buffer page (not the last page of a buffer), () is the last page of a buffer, .. is a free page.

```
F>R MAP
0  $$  $$  $$  $$  $$  $$  BS  BS  BS  BS -TE -DT -DT  MA  ..  $$
1  $$  $$  $$  $$  $$  $$  $$  $$  $$  $$  $$  LE  LE  LE  LP  LP
1  LP  LP -AO -PE -PE -PE  ..  ..  ..  ..  ((  ()  ((  ()  ..  $$
```

This is a map of a system in which 8K is assigned for the foreground. The numbers at the beginning of the line denote the field number. Each line is 2K. Note that all tasks are allocated towards the beginning of the map. Note also that most of the space is free or released.

ME.TK (Magtape Emulator) interfaces the fakehandler with MT, the magtape driver. Magtape is a complicated device, with many uncommon features and commands, eg. rewind, backspace, skip, etc. In OS/8 these commands have been implemented as special function calls to the handler (MTAn:). However, one function has not been implemented this way, the setting of parity and density. The normal mode for OS/8 is so called dump format, 6 bits per frame, both for 7 track and 9 track tape. To change the parity and density setting (as eg. MCPIP does) the OS/8 handler is patched after it is fetched in core. It will be clear that this procedure is impossible with the fakehandler. Unfortunately, MCPIP never considers whether it has got the true handler (as it should do for RTS/8 as well). So we can not support MCPIP.

ME passes the arguments of the OS/8 fake handler call to MT, after slightly repacking them: if it is a special function call (length is 0), the function bits are moved to bit 6-8 of the function word (normally the field bits), and the unit number goes in bits 9-11. This is necessary to conform to the 3-word format of the MULTI8 blockdriver protocol. As a consequence, special function 6 (read/write a record with specified length) is not supported, as there is no place to put the field of the users' buffer. Also ME makes a decision whether to set the background inactive, based on the function to be performed.

MT.TK (the magtape driver) is largely equivalent to the OS/8 magtape handler MTA:. Both the older TC58 (and the compatible DATUM controler) and the TM8E are supported.

OD.TK (the foreground ODT task) is RUN through CD. First it calls the password task (PA), as OD opens the way to any system crash you can imagine. Then you may inspect and alter any memory location:

```
F>R ODT

PASSWORD: OK!                 (password is not echoed)

7F                            (select field 7)
200/ 1324(CR)                 (examine location 200, field 7)
300/ 5473(LF)                 (examine 300, close and open next)
```

```
0301 2314(LF)          (close and open next)
0302 1233 1234(CR)     (deposit  1234 and close)
E                      (exit)
```

Note that the initial field selection is field 0!

PA.TK (the password task) is called with the name of a keyboard driver task in the AC. PA will ask for the password and check it. It returns with the AC unaltered and the Link set if password wrong, cleared if password correct. The password characters are not echoed. As distributed, the password is SESAM(CR).

PE.TK (the punch emulator) is nearly identical to the lineprinter emulator (LE) in character mode. It does however, not test for ^Z.

PL.TK is the (spooled) plotter driver. This task takes blockdriver calls and writes the plotter data into the file SPOOL.PL. PL contains a slave-task that will read the file and send the plotter commands to the XY8E. Only unencoded plotters are supported. The diskfile may by on the system disk or any other random access device (see task source). If the file is not found, PL passes an handler error to GE, which will cause an emulation error.

PL conditionally supports a RICOH GP-10, a small drumplotter that can be interfaced through an KL8E interface. This is a rather neat and cheap plotting device, which is well supported by GE/PL. For instance, each picture will be headed with a date and time stamp, and ended with a message. The input side of the KL8E interface can, after a simple modification, be used to connect a 'picture abort' switch, which will suppress unwanted output, without disturbing further pictures that might be in the spoolfile already. Contact Westvries for more information on the device.

PP.TK (the papertape punch blockdriver)

PR.TK (the papertape reader blockdriver)

RE.TK (the reader emulator task) emulates the normal papertape reader IOT's. To speed up processing, the RSF instruction is patched with a SKP in order to reduce the number of program traps. When the end of tape is reached (detected by a reader timeout) the SKP is replaced by a NOP. Most reader routines have an internal timeout mechanism that loops a large (eg. 4096) number of times along an RSF instruction. Because of the emulation, such loops tend to last rather long. By patching the RSF with NOP, execution of the timeout loop is speeded up considerably. If the same reader routine is to be used to read another tape, the RSF must be reinserted by the program.

RX.TK (flopy disk driver) handles RX01, RX02 and RX04 type drives. It operates in standard 12bit mode, following OS/8 interleave

conventions. The type of operation is dynamically determined by the drive hardware and the formatting of the media in the drive.

ST.TK (timesharing status printer) gives a life display of the activities of the virtual machines and the actual use of the background memory. ST is invoked with:

```
F>R ST
CORE   BG1 BG2 BG3 BG4 BG5 BG6 BG7
153533 F2H OS8 ^B  EDT RUL KB  IOL
```

the second line will be continually rewriten (overprinted). The first six possitions denote the up to six background fields (ie. field 2-7 of the machine). Each digit specifies the number of the background that currently owns the corresponding field. Then the status of each background is given in two or three characters. The following possibilities exist:

BYE	loged out.
OS8	waiting in the keyboard monitor for a new command.
TEC	waiting for input in TECO.
EDT	waiting for input in EDIT.
KB	waiting for terminal input.
IOH	waiting for IO, high priority (LONG is clear).
IOL	waiting for IO, low priority (LONG is set).
^B	terminal is in CONTROL/B mode.
RUH	executing, high priority (LONG is clear).
RUL	executing, low priority (LONG is set).
FnH	waiting for field n, high priority (LONG is clear).
FnL	waiting for field n, low priority (LONG is set).
HLT	processing an emulation or swap error.

After writing the status line, ST tests its break flag. So you can stop the display with:

```
F>B ST
```

Also, if the output handler is busy, ST loops for approximately 1 second, and then gives up. So if you call CD (by typing CONTROL/F) on the terminal where ST runs, and then wait for 1 second, ST will have terminated.

TA.TK (the talk task) is run with giant IOT 4. It expects a TALK n xxxxxxx command at location 01000 in the virtual memory (the OS/8 keyboard monitor input buffer resides there) and sends the message to the named terminal, or to all terminals if n=0. If a terminal is claimed by another foreground task, TA will timeout and abort the message.

TE.TK (the DECtape emulator) forms the interface between the Central Emulator and the DECtape handler (TC08 or TD8E). Main feature of TE is that it determines whether to deactivate the background or not, dependent on the distance between the target block and the current tape possition. To this end DT maintáns a table with for each dectape unit its approximate

possition (TAPETB in the monitor data area). If the targetblock is more than 50 (octal) blocks from the current possition or the tape possition is unknown (see at TI, section 3.1), the background is set INACTIVE (and possibly swapped out by the Background Scheduler). TE will direct DT to perform a search-only to the target block-4. Next the background program is requested in memory and the actual transfer is performed.

- o - o - o -

4. Timesharing

The utility of the system is greatly extended by the timesharing background facility. This implements a number of virtual machines on which independent copies of the OS/8 system may be run. In this way the system may become a multi-project computer, providing a sufficiently general environment for almost any type of usage.

The timesharing subsystem is based on the DIGICOS Memory Management Unit (MMU). This device can be programmed to trap each IOT, HLT or OSR when the cpu is in User Mode. As CDF, CIF and combinations thereof are in the IOT class, programs can effectively be denied access to parts of the machine reserved for other users or the system. Relocation hardware in the MMU allows user programs to be run in any combination of memory fields. IOT trapping can be disabled selectively, which means that dynamically fieldchange instructions to specific fields can be enabled on disabled. These functions of the MMU are extensively used by the virtual memory system.

4.1 Introduction

In this section emphasis will be on the global structure of the timesharing subsystem. Main components of the timesharing system are:

- the Central Emulator; receives traps and emulates instuctions.
- the Input Reader; listens to the terminal input.
- the Output Writer; writes output to the terminal
- the emulator tasks (DECtape, lineprinter, reader/punch, etc.)
- utility tasks (error printer, CONTROL/B command interpreter).
- the Background Scheduler task.
- the background dataarea; a set of tables, common to all these tasks.

General flow of control:

An IOT instruction is executed in the background (in usermode). This results in a trap interrupt, that starts the Central Emulator task, which determines the type of the trapped instruction and either branches to a local emulator routine or CALLs an external emulator task. In the latter case this task will eventually return control to the Central Emulator, who will continue the program executing in the background.

Characters typed at the terminal are read by the Input Reader (In) and stored in the input buffer. Output characters generated by the users program are put in the output buffer and eventually sent to the terminal by the Output Writer (On).

Independent of these, the Background Scheduler will now and then remove one of the currently running programs and install one of the other backgrounds. The Background Scheduler and the emulator tasks communicate via the status words of the backgrounds and timing SIGNALS sent forth and back. The system dispatcher insures that only backgrounds with a valid status (eg. in memory, not being emulated, etc.) are granted cpu-time.

The parts of the timesharing subsystem communicate mainly through the background data tables. Each background is described by a set of variables with a standard layout. Thus a background may be identified by a pointer to these variables.

```
/BACKGROUND DATA TABLE FOR BACKGROUND 1.

BG1,     RELOC 0            /ALL FOLLOWING SYMBOLS ARE RELATIVE

USTAT,   INACTIVE+ONDISK    /STATUS WORD
UMQ,     0                  /USERS MEMORY QUOTIENT REGISTER
USC,     0                  /MSS.SSS.FFF.XXX: A/B MODE,
                            /STEPCOUNTER AND LOCKED FIELD
UPC,     7201               /USERS CURRENT PROGRAM COUNTER
UFLDS,   100                /LGX.XXU.III.FFF: LINK, GREATER-THAN,
                            /USER MODE, VIRTUAL INSTRUCTION & DATAFIELD
UAC,     0                  /USERS CURRENT ACCUMULATOR
UINST,   0                  /LAST TRAPPED INSTRUCTION
USW,     0                  /USERS SWITCH REGISTER (VIRTUAL)
UTEMP,   0                  /SCRATCH LOCATION
UDTV,    ZBLOCK 3           /TRANSFER VECTOR
UBUFIN,  0                  /COUNTER OF INPUT BUFFER
         BG1IN+1            /READ POINTER
         BG1IN+1            /WRITE POINTER
UBUFOUT,0                   /COUNTER OF OUTPUT BUFFER
         BG1OUT+1           /READ POINTER
         BG1OUT+1           /WRITE POINTER
UWRTR,   KHOBG1+4           /TCBP OF OUTPUT WRITER
UCUR,    EMBG1+4            /TCBP OF EMULATOR
UCHNL0,  "D^100+"K&3777     /DRIVER FOR USER'S SYS:
UNUMB,   4511               /FILE STRUCTURED, DEVICE TYPE 51, UNIT 1
         "S^100+"Y&3777     /ENTRY FOR DSK0:
         6500               /READ ONLY, OS/8 SYSTEM, UNIT 0
         ZBLOCK 4           /CHANNEL 2-3
UECHO,   0                  /ECHO FLAG, 4000=NO ECHO
UFLD0,   ZBLOCK BGCORE      /TABLE OF REAL FIELDS, 0=NOT-RESIDENT
UCHAR,   0                  /THE CURRENT INPUT CHARACTER
UKB,     K1TCBP+0           /TCBP OF INPUT TASK
UTTY,    T1TCBP+0           /TCBP OF OUTPUT HANDLER
UCOUNT,  0                  /COUNTER FOR 'BS'
USLOT,   MAXSLOT+1          /WAIT-FOR-BG-IN-CORE SLOT
UACCNT,  ZBLOCK 2           /LOWORDER, HIGHORDER CPU USAGE
                            /IN UNITS OF DGNTICK
UNEXT,   BG2                /POINTER TO NEXT BG
UEND=.                      /END FOR THIS BG
         RELOC              /END OF RELATIVE DEFINITIONS
```

Note that the UNEXT pointers form a ringstructure; UNEXT of the last bg points to BG1.

Layout of the status word (USTAT):

Bit 0: INACTIVE; If set the background is blocked by an emulator, eg. waiting for I/O.

Bit 1: EMULATE; if set, the background is actively being emulated. Its lock-field (ie. the instruction field, or the last field brought into memory by an 'incore' request) may not be swapped out of memory.

Bit 2: BGSTOP; Set by the input reader when the background enters CONTROL/B mode.

Bit 3: ONDISK; Set when the backgrounds instruction or data field are not in memory. This bit is only changed by the background scheduler.

Bit 4: LONG; Set by the background scheduler when the background has expired a short slice. Cleared by some emulators to raise background priority after I/O completion. In certain cases the Background Scheduler will give precedence to backgrounds with LONG=0.

Bit 5: INCORE; This bit is set by some emulators and requests the background scheduler to send a signal when the requested field is in memory. It is then cleared by the background scheduler.

Bit 6-8 INCFLD; along with the INCORE bit, these bits are set to the virtual field number requested by the emulator.

Bit 9: (reserved)

Bit 10: BGERR; Set by the central emulator when an illegal instruction is encountered. Triggers the input reader to call the background error printer and enter CONTROL/B mode. Cleared by the input reader.

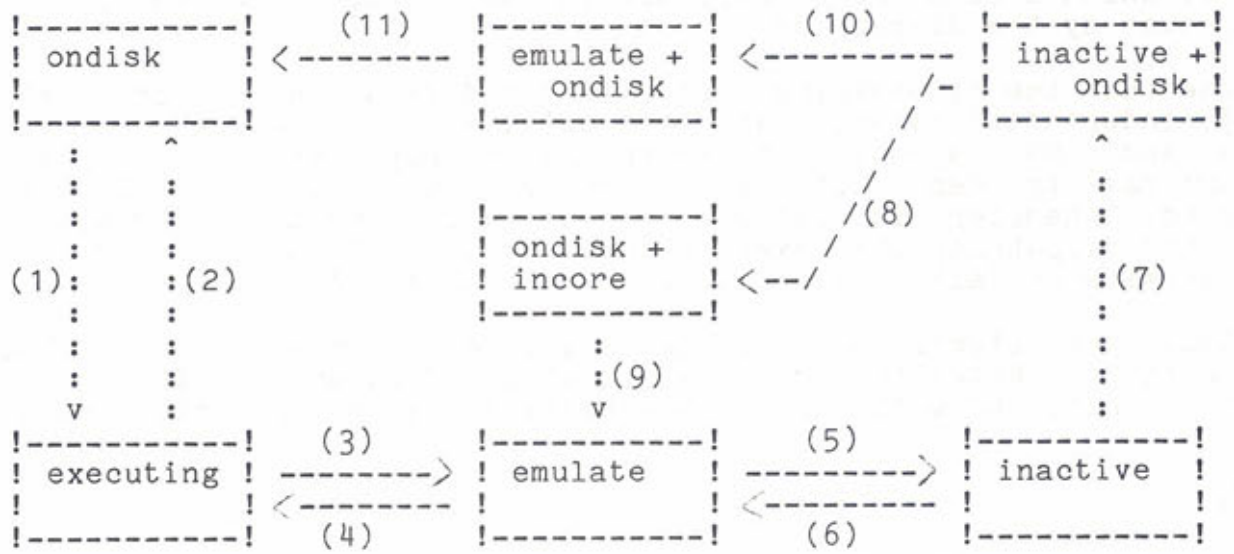
Bit 11: SWPERR; Set by the background scheduler in case of an unrecoverable disk error during swapping. Triggers the input reader to call the background error printer and enter CONTROL/B mode. Cleared by the input reader.

The following is a realistic sequence of states. Suppose a background is idle, waiting in the OS/8 Keyboard Monitor for the user to enter a command. That means the background program has executed a KSF instruction, which is analized by the central emulator. State: EMULATE. If the keyboard input buffer happens to be empty, the emulator decides to make the background inactive. State: INACTIVE, not-EMULATE, not-ONDISK. Next the background scheduler comes along and finds memory occupied by an inactive job. If there are other jobs that can proceed, the background scheduler will swap these for the resident fields of our job. Just before writing a field to disk BS will adjust the ONDISK bit of our bg.

After some time the user enters a command. This is received by the Input Reader, stored in the buffer, and then the central emulator, which had HALTed, is RUN by the Input Reader. The Central Emulator will put the first input character in the users AC, change the bg status to non-INACTIVE, non-EMULATE, and SIGNAL the Background Scheduler that another job is competing for execution. Also, the emulator has cleared the LONG bit, to show that this is an interactive

job and quick response is desired. At some later moment BS will bring our job in memory and reset the ONDISK bit. Now the entire statusword is zero, and the dispatcher will start executing the background as soon as no foreground tasks are available to be run.

Suppose that the command entered by the user takes a very long time to execute. After about half a second the BS will notice that our job has consumed its 'short slice', and set the LONG bit in the status word. If other non-LONG jobs are waiting then, they will receive the cpu now, otherwise our job enters its first 'long slice', which lasts for about 5 seconds. If during this time another job becomes active with its LONGbit reset, it will preempt our job. If there are only LONG jobs in the system, then the BS will switch every 5 seconds. A switch may or may not give rise to swapping, depending on the number of fields needed by the background programs, and the number of fields available in the machine.



(1) swap in
(2) swapout
(3) trap
(4) end of emulation
(5) wait for I/O
(6) I/O completed
(7) swapout
(8) swapin request from emulator
(9) swapin completed
(10) I/O completed
(11) end of emulation

Fig. 4-1. Background State Transition Diagram.

When our job wants to transfer some information to DECtape it executes instructions resulting in a trapped IOT that tells the Central Emulator that a DECtape transfer is to be made. DECtape transfers are handled by the Tape Emulator task, TE. TE is called by the central emulator, with a pointer to the first word of the dataarea of our bg in the AC. At that moment the status of our bg is EMULATE. As the DECtape search is likely to take quite some time, the Tape Emulator will decide to make our job INACTIVE during the time that the DECtape unit is positioning the tape. So it changes our status to INACTIVE, not-EMULATE, and SIGNALs the BS that memory is potentially free. Whether or not our job is swapped out depends on the activity of other jobs in the system, and is entirely under control of the Background Scheduler. Meanwhile the Tape Emulator will position the tape (through the DECtape driver). However, the actual datatransfer should

take place between the tape and the users memory (by databreak), so the buffer field should be in memory. When the tape is properly positioned, the Tape Emulator changes the status of our bg to INCORE (plus the virtual field needed in bit 6-8), resets the LONG bit (the tape is at full speed approaching the desired block, so we want it fast) and SIGNAL the Background Scheduler. This wakes up BS, who will discover the INCORE request bit and, conditions permitting, will bring the requested field in memory. Meanwhile the Tape Emulator is WAITing at the private eventflag of our bg. As soon as the swap is completed, the BS will reset ONDISK and INCORE, set EMULATE and SIGNAL that the field is in memory now. Along with this signal BS passes the number of the field where the buffer field has been loaded. The signal wakes up the Tape Emulator, which directs the DECtape driver to perform the transfer. During the transfer the status of the bg is EMULATE, which will prevent BS to remove the buffer field from core. After completion of the I/O operation the Tape Emulator EXITs to the Central Emulator, which clears the EMULATE bit in the status. Now the bg can be continued by the dispatcher.

So we see that the timesharing system is build from a number of loosly coupled, independent tasks, that synchronize and communicate through SIGNALs and the various bits in the status registers. Thereby the emulators have to deal with only one bg at a time, while the Background Scheduler is resposible for all scheduling decisions. Finally the dispatcher will execute whatever bg is in memory, if its status permits so (all bits except LONG must be zero).

Note that the signals sent to the BS are merely to wake it up. The only information extracted from the signal by BS is whether it was a timeout or not. BS will analize the total situation each time again.

4.2 Central Emulator

From the discussion at the end of the previous section, it will be clear which position the Central Emulator takes among the other components of the timesharing subsystem. In this section we will go into the details of the internal logic of the Central Emuator.

The Central Emulator (En) is an internal task, or in fact a set of internal tasks that share a single reentrant code-image. There is one incarnation for every bg in the system. En receives the primairy trap interrupts and emulates most of the trapped instructions. The less frequently occuring traps are handled by emulator tasks. One reason to have traps handled by separate task is that such a task acts as a resource allocator. After the first use of eg. the lineprinter (ie. after the first lineprinter IOT) the lineprinter emulator task is reserved for the central emulator that called it. From that moment on other users can't reach it, because their En has a different TCBP and will thus get an errorreturn when it tries to call the lineprinter emulator. Only after the complete file has been transferred to the printer the printer emulator will eleviate the reservation by using EXIT instead of RETURN. Thereafter it can be called by any other En again. Emulator tasks using this mechanism are LE, RE, PE and any other emulators that handle a strictly single user device.

When a trap interrupt is detected in the skipchain, control is transferred to TRAPINT in the En. First the interrupt flag is cleared, then the trapped instruction is retrieved from the user's memory. The bg state is saved (routine RSAVE) and CURTSK is set up to show that En (our one !) is running. Next the interrupt system is enabled. The status of our bg is set to EMULATE. The type of instruction is determined by indexing EMTAB with the device code (bit 3-8 of the instruction). EMTAB contains one entry for each device code, with the following encoding:

Zero	no operation, ignore this instruction
Positive	name of task that will handle this instruction
Negative	negative of offset of routine within En

If we find a taskname in the table, that task will be called with the current value of BJOB in the AC. The taskname is first compared to the name of the task that was called the last time; if they are equal, the CALL request will still contain the TCBP of that task. Otherwise, the name is copied into the CALL request (this avoids repeated lookup of the same taskname).

Special action is necessary in case the CALL request returns on its errorreturn. This may imply that the required emulator task is currently reserved for another user, and so we have to wait till it becomes free again. During this waiting time we should release the bg memory area and thus the status is changed to INACTIVE. Now our CE enters a loop that tests whether the emulator task is free every half second. This test can be made quickly and efficiently, as we have the TCBP of the task from the failing CALL request. As soon as our task becomes available, we will request our instruction field in core (set INCORE, signal BS, WAIT at USLOT) and return to the original CALL request. When the emulator task is called, there is no indication left that we had to wait for it.

As you will have noticed, there is no explicit queueing of users for a shared device. As long as one user has control over a device (in fact: over its emulator), the other users are looping until it becomes free. Which of the waiters gets the device first is entirely undefined. Although this is a very simple mechanism, it provides for the user who does not want to wait and breaks his program with the CONTROL/B key. It is expected that direct communication among terminal users will prevent serious problems in devicesharing.

The Memory Management Unit (MMU).

Version V6B and later support the Memory Management Unit, a device developed by DIGICOS b.v to specifications of the MULTI8 development team. In this section we will outline the function of the MMU and detail its integration in the MULTI8 software.

The MMU consists of one quad Omnibus module. It replaces the normal memory extension control (KM8E/KM8A), implementing two new functions, selective trapping of IOT's and field relocation.

Selective trapping (or 'untrapping') provides the capability to select by software which IOT's should be trapped if executed in User Mode and which not. So software may (by loading certain registers in the MMU) specify that eg. the CDF/CIF 30 instructions are not to be trapped. This makes it possible to dynamically control the access of user programs to various memory fields. CDF/CIF instructions that are untrapped, and RDF and RIF are executed by the hardware, without the (large) emulation overhead. This means that the larger part of the emulation overhead normally experienced in foreground/background operation is eliminated.

The selective trapping function would be of little use without the second function of the MMU; field relocation. The MMU contains a 8*3 bit relocation memory (RAM) that can be loaded by suitable IOT's. When the processor is in User Mode each memory reference by the processor is relocated by the MMU. Recall that the extended memory address for processor cycles is developed by the Memory Extension Control (either form the Intruction Field Register or from the Data Field Register). During relocation this (virtual or processor) extended memory address is used to select a (3-bit) word in the Relocation Register, giving the real or bus extended memory address. Example: the cpu performs a Fetch cycle to read the next instruction from memory. Suppose the Instruction field register contains 001 (base 2), so our program is in virtual field 1. The contents of the Instruction field register are used to address the Relocation register. This register may contain any value eg. 101. This value is gated to the Omnibus EMA lines, and the instruction is actually fetched from field 5. In this way a program running in usermode may be run in any set of fields just by loading the Relocation Register with the corresponding values.

Paging

When a user program is ready to run, the background scheduler will first assure that both its instruction and datafield are in memory. Then the program is started, with only the CDF/CIF to the loaded fields untrapped. Thus the program can run full speed, until it executes a CDF/CIF to a non-resident field. This instruction will be trapped. Such an event is customarily called a 'page-fault'. The central emulator, receiving the trap interrupt, will request the target field INCORE. BS will determine if and where the new field can be loaded. After reading it into memory, BS will adjust the field table of our background to show that the new field is present. When our bg is later elected for execution by the dispatcher, the MMU is instructed to untrap the CDF/CIF to the now resident field. The relocation map is adjusted. Also, the central emulator has decremented our program counter, with the effect that the trapped CDF/CIF is executed again. However, this time the instruction is not trapped and our program can proceed.

Instruction set

The MMU executes all instructions of the standard Memory Extension Control plus the following set:

6205 RTM Read Trap Register; Jam-loads the AC with the contents of the Trap Register. The Trap Register contains the

instruction that was last trapped by the MMU.

6235 LTM LOAD Trap Memory; Loads bit 11 of the AC in the trap/untrap memory addressed by AC bits 3-8, e.g if AC=230 then the IOT's with device code 23 (CDF 30, CIF 30 and CDF CIF 30) will be trapped, if AC=231 these IOT's will NOT be trapped. The AC is cleared.

6245 LRM Load Relocation Memory. AC 6-8 contains the real address (to become the contents of one word of the Relocation Memory), AC 9-11 contains the virtual address (used to address to Relocation Memory). The AC is cleared.

6265 SMME Set Memory Management Enable.

6275 CMME Clear Memory Management Emable. If disabled the MMU is fully compatibel with the normal Memory Extension Control. The MMU is also disabled by pressing the Load Extended Address key/button on the front panel.

6215 SKME Skip if Memory Management enabled.

6225 SKMM Skip if Memory Management available.

Terminal output

Terminal output instructions are quite easily handled. TLS and TPC will transfer the character contained in the users AC to the output buffer of the user's terminal. In addition the users Output Writer task is RUN to ensure that the characters are output indeed. The TSF instruction is always replaced by a skip; when the output buffer is full, the emulator will make the bg inactive at the first TLS or TPC that would overflow the buffer. In that case the central emulator enters another loop where it tests the counter of the output buffer. As soon as there are less than 24 characters in the buffer, the bg is made active again. This avoids a delay in the typing at the end of the buffer, while retaining the profit of executing other programs during slow terminal output processing.

Each output character comming from the background is compared to UCHAR, the last input character accepted by the user's program. If they match, the output character is dropped as it probably is the echo for the input. Special care is taken if UCHAR is TAB (211): any sequence of SPACE characters comming from the background is suppressed. Also CR gets special attention: if CR is identified as echo, UCHAR is set to LF because the Input Reader has already supplied a LF to the terminal.

The buffers used to hold terminal input and output characters have a standard layout and are always handled through a set of general routines: FILLQ, MTQ, GETQ, CLEARQ. All these routines accept one parameter, which is the offset from BASE of the buffer descriptor.

```
JMS I (CLEARQ   /RESET THE USER'S INPUT BUFFER
   UBUFIN       /
```

FILLQ and MTQ are crossfield callable. They should be called with datafield=instructionfield. The parameter is added to the value of BASE in the callers field.

The buffers are build in a shared pool, consisting of 16-word blocks. Each buffer is rooted in a 3-word descriptor of the following format:

```
COUNT,  0       /NUMBER OF CHARACTERS ACTUALLY IN THE BUFFER
HEAD,   BUF+1   /POINTER TO FIRST CHARACTER IN THE BUFFER
TAIL,   BUF+1   /POINTER TO LAST CHARACTER IN THE BUFFER
```

The first word of each 16-word block is used to link the blocks together. Thus in a filled buffer each block contains a pointer plus up to 15 characters. At any moment each buffer has at least one block allocated. More blocks can be hooked to a buffer from the chain of free blocks. When all characters have been read out of a block, it is returned to the free chain, unless it was the last block in the buffer. A counter is maintained of the number of free places in the free chain. When a new block has to be allocated, this count is compared with the counter of the requesting buffer. No block is allocated if the buffer already contains more characters then are still available. This prevents that the pool is monopolized by one or a few buffers.

Terminal input

The processing of terminal input instructions is rather complicated. Most of it is described in the section on the Input Reader. Notice that a strategy has to be determined to decide when the user's program expects to see a new character, and when it expects to see the same input character again. MULTI8 assumes that a character is completely read as soon as the program executes a KCC instruction. This is in correspondence with the fact that KCC will normally step the terminal reader in case of papertape input. KSF signals the emulator that the program wants to know if there is an input character. If there are characters in the inputbuffer, KSF is made to skip (by incrementing the user's stored programcounter). If the inputbuffer is empty, the instruction right after the KSF is considered. If it is a JMP .-1, the background is set INACTIVE. The Background Scheduler is SIGNALed and the Central Emulator HALTs. When new input has arived, it will be RUN by the Input Reader.

Because the Input Reader has already echoed most of the input characters, the emulator has to strip the echo supplied by the background program. This is accomplished by comparing each output character with the last character read. So with each emulated KRB, KCF or KCC the inputcharacter is copied in UCHAR, where it is later compared with output characters.

SM8 (Skip on MULTI8)

In certain cases it is desirable to test whether the program runs in the MULTI8 background or not. Therefore the SINT instruction (6254) is emulated as a SKP.

PEEK

Mainly for debugging purpose a special PEEK instruction is implemented (6264, normally a CUF). The PEEK instruction makes it possible to examin any location in the machine. Example of usage:

```
TAD (ADDRESS    /ADRESS TO PEEK
6264            /
   CDF 30       /FIELD TO PEEK
....            /AC CONTAINS CONTENTS OF ADDRESS
```

GIANT IOT.

To facilitate communication between background programs and foreground tasks, a special IOT has been assigned that can be used to perform all kinds of special functions. When a 6770 instruction is trapped, the contents of the user's AC is used as an index in GIGATB, a table with the same encoding as EMTAB. Any task that is named there will be called as a normal emulator task (eg. with a pointer to the user's dataarea in AC and datafield). Standard functions are:

0	Read time of day into AC; hhh.hhh.mmm.mmm
1	get terminal number in AC; 000n
2	disable keyboard echo
3	enable keyboard echo
4	invoke TALK task
5	used for OPEN/CLOSE
6	stall program for n seconds.
7	Reset user's account register.
10	Read user's account registers in AC and MQ.
11-17	Reserved for system
20+	user assignable

The Handler Call.

The 6000 instruction has been reserved for the fakehandler to pass request parameters to the foreground. The following protocol is used:

```
TAD (000.000.DDD.UUU /GET DEVICE AND UNITNUMBER
6000                 /TRAP !
JMP .+4              /JUMP OVER THE PARAMETERS
   FUNCTION          /FUNCTION WORD (COPY OF OS/8 HANDLER CALL)
   BUFFER            /BUFFER ADDRESS(COPY OF OS/8 HANDLER CALL)
   BLOCK             /BLOCKNUMBER   (COPY OF OS/8 HANDLER CALL)
SZA                  /ERROR ?
 JMP ERROR           /HARD DEVICE ERROR OCCURED
```

The device number (bit 6-8 of the user's AC) is an index in HNDTAB which - again - has the same encoding as EMTAB.

Entry 0 is for the so called 'channels'. Each user has available 4 channels, numbered 0-3. Channels give access to disk-like device. Channel 0 is the user's SYS:. All requests passing through this channel are checked to see if the background is loading its Keyboard Monitor or Command Decoder. In that case the routine EMREL(ease) is executed, which RUNs all emulators named in ASEMTB (ASsignable EMulators TaBle) with Link=1. If this background had any of these emulators, they will finish their tasks and EXIT. This mechanism insures that a background releases all its devices at the end of each program or program step. An exception is made for backgrounds that execute in BATCH mode (as indicated by the contents of virtual 07777); a batch is considered one (large) program. Further the routine EMREL copies the system date (which is updated automatically at 24:00) in the OS/8 date word. Next the virtual disk unitnumber belonging to this terminal is extracted from the dataarea and the request is passed to SY or DK, depending on user disk allocation.

Each background has an 8-word table that describes the actual device open on each of its four channels. Thus each channel is described by a two-word entry, one word for the name of the handler task (eg. SY), and one word that contains a read-only bit (bit 0), the OS/8 device type (bits 3-8) and the unit number (bits 9-11). As said, channel 0 is the background's SYS:. Channel 1 is generally opened to DSK0:, the system disk where the CUSPS reside. Channels 2 and 3 can be assigned by OPEN statements from the background, and deassigned with CLOSE.

Further with the encoding of HNDTAB. Entry 1 is for the DECtapes, 2 for the lineprinter, 3 for magtapes (MTA0-7), 4 for floppy disks (RXA0-7), 5 for public disks (RKB0-RKB3) and 6 is for the cardreader (CDR:). Entry 7 is reserved for future expansion.

4.3 Terminal Input Reader.

The Input Reader (In) has a number of important functions. It is the direct representative of the user and defines the extend to which he can control the system. Unlike emulator tasks, the Input Reader is synchronized with the user, rather than with the user's program.

Basically the Input Reader accepts characters from the keyboard handler and stores them in the user's input buffer. If the inputcharater is CONTROL/S (XOFF), the Input Reader will STOP the Output Writer; If the input character is CONTROL/Q (XON)), the Output Writer is RESTRTed°. If the inputcharacter is CONTROL/O or CONTROL/C, both the input and output buffers are cleared before the character is stored. This insures that the program will see the break character and that the user gets immediate response.

If the inputcharacter is CONTROL/B, the status of the background is set to BGSTOP, which implies that the terminal is in CONTROL/B mode now. The inputbuffer is cleared and further input is collected until the next CR. Then the CONTROL/B task (CB) is called that interprets the command line. CB may return with AC=-1, 0 or +1. -1 Means

- - - - -

°) To avoid confusion when users inadvertently type ^S, the Output Writer is also RESTRTed by ^B, ^C and ^O.

command

Error, 0 means continue background execution, +1 means stay in CONTROL/B mode and read another command.

Another function of the Input Reader is to RUN the Central Emulator when it has HALTed for lack of input characters. Of course, the Central Emulator could be RUN each time a character was stored in the input buffer, but that would imply that the background is awaked for each input character and a very high swapping rate would result. Therefore a rather peculiar algorithm has been devised that assembles input characters until a complete command has been entered. Because the Input Reader has no idea of the command syntax that the current background program uses, it makes a guess according to the following rules:

1) input characters are classified as non-printing or printing characters. ESC and TAB are considered printing characters.
2) printing characters are stored in the input buffer and the Central Emulator is not activated.
3) non-printing characters are stored in the buffer and the Central Emulator is RUN (eg. after CR, LF, RUBOUT, etc.)
4) when no new input character is received within a short time while there are some characters in the input buffer, the Central Emulator is RUN (eg. when the user types a "/" to ODT). This insures that the system will respond by the time the user starts waiting for it.

The length of "a short time" is a function of the number of characters already in the inputbuffer; the more characters have been entered, the longer the system waits before starting the background program. In this way a fairly good distinction is made between short interactive commands (ODT, EDIT) and normal line-oriented commands.

Another function of the Input Reader is to echo printing input characters to give the user the impression that his program is actually running. ESC is echoed as "$", and CR gets a LF appended. TAB is echoed as it is (and is expanded by the output handler). Other control characters are not echoed. By setting bit 0 of these characters in the input buffer, the suppression of the users echo is prevented for these characters (UCHAR will not match with the program's echo).

For some programs it is desirable to inhibit terminal echo, eg. a display-oriented editor. Keyboard echo is disabled by executing the GIANT IOT with AC=2. Echo can be enabled with AC=3. The echo is automatically restored at the end of the program (by EMREL) and when the terminal enters CONTROL/B mode. In the latter case the original echo mode is restored when the program is resumed. When the echo is suppressed by this mechanism, all characters are flagged as not-echoed in the input buffer (bit 0=1) so that any echo from the background program will get through. Also, all characters are treated as control characters and thus immediatly activate the background program.

4.4 Terminal Output Writer.

The function of the Terminal Output Writer task (On) is very simple. It reads characters from the bg output buffer and sends them to the user's terminal output handler (Tn). Thereby bit 0 of the AC is always cleared to allow foreground tasks to interrupt background output. When the output buffer is empty, the output writer HALTs. It should be RUN again when a new character is put in the output buffer.

To support the CONTROL/S and CONTROL/Q functions that temporarily stop/start terminal output, the output writer is STOPped and RESTRTed by the input reader.

4.5 Emulator Tasks.

This section will not give details on all external emulator tasks. Instead it will describe the protocol defined between the emulator tasks, the central emulators and the Background Scheduler. It may be helpful to refer to the bg statediagram in fig. 4-1. Individual emulator tasks are described in the section on external system tasks (3.2).

When an IOT instruction is trapped, the device code is used to index EMTAB. EMTAB may contain a positive value, in which case it is the name of an emulator task that is subsequently called by the central emulator. At that point the AC contains the address of the bg dataarea of the bg involved, the Link=0 and the datafield is 1. The bg is in the state EMULATE and its instruction field is locked in core. The trapped instruction can be found in UINST, the users AC in UAC, his PC in UPC, etc. Note that the symbols UAC, UPC, etc are defined as offsets relative to the begin of the bg's dataarea, eg. UPC=3.

If the action to be performed by the emulator task can be done quick (eg. less than 1 or 2 seconds), the task need not alter the state of the bg. If the function takes more time, it is worthwile to set the bg INACTIVE so that our bg may be removed from memory. If an emulation error (eg. illegal instruction) has occured, the task should RETURN or EXIT with a positive, non-zero AC; This will generate an error display (PC=...(EMULATION ERROR)) and bring the terminal in CONTROL/B mode.

In most cases an emulator task manages a resource, eg. a device. The resource scheduling is also implemented in the emulator task. As long as an emulator task is executing for one user, no other program can call it. Also, if an emulator, after a CALL from one user, returns with RETURN (instead of EXIT) the emulator can only be called later by the same user. This makes it possible to handle a series of requests from one user without intervening calls from other backgrounds.

Of course, each emulator will eventually EXIT. This can be done after some suitable signal from the background that is served, eg. end-of-file in an output file, or from the device driver called by the emulator, eg. end-of-file from an input device. In any case care must be taken that the emulator eventually EXITs, even if the background

program is aborted without sending an end-of-file or reading to the end of the input medium. To this end each emulator can be called (RUN actually) when a background enters the OS/8 keyboard monitor or command decoder. To be RUN, the emulator task should enter its name or task control block pointer in the table ASEMTB, located in field 1. ASEMTB is a table, initially filled with zeros, long enough to contain the worstcase combination of active emulator tasks. After the first call, an emulator task should search a free (=0) entry in ASEMTB and enter its own name. If a background later enters OS/8, all tasks listed in ASEMTB are RUN. Of course, only the tasks actually 'owned' by this background receive the RUN request. The distinction between this RUN and a regular CALL is in the Link, which is set for the RUN. Note that the emulator task has to remember in which entry of ASEMTB it did store its name, and that, before EXITing, it must zero this entry again. Failure to do so will result in slowly filling ASEMTB and leads to unpredictable results. In the emulator tasks supplied with the system many coding examples can be found. The following code could be used to setup ASEMTB:

```
        TAD (ASEMTB-1   /SETUP ADDRESS OF BEGIN OF ASEMTB
        DCA AUTO10      /FOR A SCAN TO FIND AN EMPTY ENTRY
        CDF 10          //ASEMTB, LIKE ALL BG STUFF, IS IN FIELD 1
LOOP,   TAD I AUTO10    //FETCH AN ENTRY
        SZA CLA         //IS IT EMPTY (ZERO) ?
         JMP LOOP       //NO, TRY NEXT ONE
        TAD AUTO10      //YES, GET IT'S ADDRESS
        DCA ENTRY       //AND STORE IT IN THE TASK
        CDF 0           //CURTSK IS IN FIELD 0
        TAD I (CURTSK   //GET TCBP OF RUNNING TASK (THAT'S ME!)
        CDF 10          //BACK TO FIELD 1
        DCA I ENTRY     //AND STORE IT IN ASEMTB
```

ASEMTB has been dimensioned so large (16 locations) that a free entry will always exist (16 emulator tasks must be active to fill it). Note that the task may not RELEASE or SWPOUT without first clearing it's ASEMTB entry. So when the emulator is finnished (eg. at end-of-file), it should perform

```
        CDF 10          //ASEMTB IS IN FIELD 1, YOU KNOW
        CLA             //IF NECESSARY
        DCA I ENTRY     //ZERO MY ASEMTB ENTRY
```

On return from an emulator task, the AC signals one of three different things. As always, a zero AC signals 'no problem', and the background program will be continued. A positive AC signals that some emulation error occured, and will bring the background in CONTROL/B mode, with its current state displayed. A negative AC is interpreted as being an instruction that should replace (be patched over) the trapped instruction in the background program. The Central Emulator will apply the patch, backup the user's program counter and continue the background.

The virtual memory system has several consequences for emulator tasks that deal with the background memory, eg. to obtain parameters, return values, or perform I/O to or from the user's memory. When an emulator task is entered (with Link=0, eg. not the release RUN), the BG instruction field is known to be in memory. The field where it is loaded is obtained by the following code:

```
        TAD XXBASE      /ASUME WE HAD STORED THE ENTRY AC HERE
        TAD (UFLDS      /ADDRESS USER'S FIELDS WORD
        CDF 10          //ALL USER DATA IS IN FIELD 1
        JMS DEFER       //GET USER'S FIELDS WORD
        AND C70         //EXTRACT HIS (VIRTUAL) INSTRUCTION FIELD
        CLL RTR         //MOVE IT TO BITS 9-11
        RAR             //
        TAD (UFLD0      //START OF FIELD TABLE IN USER DATA
        TAD XXBASE      //
        JMS DEFER       //THIS GETS THE REAL FIELD NUMBER IN 6-8
        TAD C6201       //MAKE A CDF
        .....           / USE IT TO ADDRESS USER'S INSTRUCTION FIELD
```

As long as the user's state is EMULATE, his instruction field will remain in place. But after the BG has been INACTIVE, you must request it into memory again before using it. This is also true for any other field that you may require. The logic of emulator tasks can thus be simplified by insuring that all parameters they need are always in the instruction field, ie. the field where the trapped instruction resides.

To request a background field into memory, use the following code:

```
        CDF 10          //ACCESS BACKGROUND DATA TABLES
        TAD I XXBASE    //GET USER STATUS
        AND (-INACTIVE-EMULATE-1 //CLEAR INACTIVE AND EMULATE
                        //(ADD '-LONG' TO GET IT FASTER)
        TAD (INCORE     //SET INCORE REQUEST
        TAD XFLD        //ADD VIRTUAL FIELD NEEDED IN BITS 6-8
        DCA I XXBASE    //THAT'S HIS NEW STATE
        JMS MONITOR     //SIGNAL BACKGROUND SCHEDULER TO LOOK AT IT
           SIGNAL       //
           BSSLOT       //
        TAD XXBASE      //NOW GET PRIVATE EVENT OF THIS USER
        TAD (USLOT      //
        JMS DEFER       //THIS GIVES US THE EVENT NUMBER
        DCA .+3         //STORE IT IN THE WAIT REQUEST
        JMS MONITOR     //WAIT TILL BS TELLS US THAT THE FIELD
           WAIT         //IS IN MEMORY, AND WHERE IT IS
           0            //(GETS USLOT)
        TAD C6201       //AH! AC CONTAINS REAL FIELD NUMBER !
        ....            //NOW WE GOT A CDF TO THE REQUESTED FIELD
```

At this point the state of your BG is EMULATE again, which insures that the field just brought into memory will stay there.

4.6 Special Functions.

The special functions of the background support system are performed by two external tasks, in co-operation with the Input Reader. When a terminal enters CONTROL/B mode, the Input Reader will assemble one command line. After receipt of Carriage Return, it CALLs the CONTROL/B task (CB), that will analyze the command. Most commands are

performed by CB itself (eg. setting the user's switch register), but the WHERE command is executed by BE (see section 3.2).

4.7 The Background Scheduler.

It is the function of the Background Scheduler (BS) to decided which background job will be in memory at each moment. BS implements a two-level round-robin system, with level 0 having preemptive priority over level 1. The timeslice used for level 0 (short slice) is about .5 seconds, the long slice (used for level 1) is 5 seconds. New jobs enter at level 0. After their short slice they are moved to level 1 where they are inserted in front of the queue (to avoid extra swapping). In fact no real queues are maintained, but use is made of the fixed ring-structure formed by the UNEXT pointers. Whether a job is at level 0 or at level 1 is defined by the LONG bit in its status word.

Normally BS sits in a WAIT for its own dedicated event BSSLOT. It may be awaked either by a timeout (BS issues a timeout value of 1 system tick) or by a SIGNAL from one of the emulator tasks.

If it is a timeout (AC=2), BS will update the running job's counter, that was setup for the current timeslice. If the counter overflows, BS has to analize the current situation.

If the bg who's slice has just expired was a LONG (level 1) job, BS will search for a new job to run, first a non-LONG job or else a LONG bg. The last candidate considered is the same job again. If it was a short slice that has now expired, BS will check if there are other non-LONG jobs waiting. If not, the current bg will immediately get its first long slice.

If BS is awaked by a SIGNAL from an emulator task, it will first look for an INCORE request of the current level 0 job. If there is such a request, it will be serviced and the slice is continued. If no INCORE request is pending, BS will only check whether the current bg is still able to proceed. If not, BS will scan the backgrounds for a new candidate, either a non-LONG job or a LONG job.

Each time a new bg is elected, BS checks if its instruction and datafield are in memory. If not, BS will will attempt to load the missing fields. If none of the resident fields can be swapped out, BS sets a flag (BSFLAG) for the Central Emulator, who will send a SIGNAL as soon as the current emulation is finished.

The crucial element in the background scheduler is the algorithm that decides which of the currently resident fields should be replaced by a requested field. The implemented algorithm is very complex, to be honest, I'm not sure that I understand it myself. The structure of the algorithm is that each of the available fields is tested to see if it meets a certain condition. If one does, that one will be swapped. If none of the resident fields meets the first criterium, a new scan is started with a new, weaker, criterium. This process is repeated for 6 successively weaker tests. The policy decission is in the choice of the different tests.

Appart from the trivial test 0 (field not yet occupied), the first test will select a field who's present owner is blocked (INACTIVE, BGSTOP, BGERR or SWPERR set but EMULATE clear; But the field may not be the owner's current instruction- or data-field. The second test is equal to the first test, but now the field may be its instruction or data field. The prefference to non-instruction- or data-fields seeks to minimize the chance the the presently blocked bg will pagefault immediatly after activation.

Test3 and test4 again differ only in the I- or D-field criterium. Both accept fields owned by backgrounds not currently in emulation. However, the field may no be owned by the present requestor. Test5 can be passed by any field that is not locked and not owned by the requestor. Finally test6 looks for any field that is not locked or the owners I- or D-field. Note that when we come to test6, all non-locked fields must be my own fields, so test6 lets a background exchange its own fields. Clearly it is useless to swap ones own instruction- or data-field out because that blocks any futher execution.

The scanning of fields always starts with the field past the last selected one, so that several fields falling in the same calss (ie. passing the same test) will be selected in round-robin fassion. A further refinement is that, after the initial swap-in of I-, D-field and one other field in case of a pending INCORE request, only the job with the highest priority in core gets its pagefaults serviced. So if both an interactive and a computebound job are resident, the intercative job can request fields at the expense of the computebound job, but not vice-versa. Note however, that as soon as the intercative jobs becomes inactive, the computebound job can regain its space.

The actual swapping of background fields is performed in chunks of 4K, first writing all the old contents to disk, then reading the new field into memory.

- o - o - o -