MULTI8 RELEASE NOTES

July 1979

MULTI8 V7A release notes.

This document describes specific features of MULTI8 V7A. It is primairily intended for V6 users and explains many features in terms of the difference between the two versions. Related documents are the MULTI8 Terminal Manual and the MULTI8 System Manual. These manuals may be ordered from:

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

MULTI8 V7A incorporates about two years of further development that started from V6. The single feature that has a main impact on system performance is the virtual memory. Other changes are intended to simplify the system generation procedure, increase the systems reliability and functionality.

Virtual memory.

In V6 at any time there was just one background program resident in memory. A swap always involved the complete memory of a virtual machine, ie. 8K or 16K. If the system was configured for 16K backgrounds, it would always swap 16K, even if the user was running a 4K program. In contrast, V7A swaps only those fields of a background that are realy used by the program. This greatly reduces the number of field swaps. Also, V7A can swap a field that is needed into any available field of the real machine. Thus portions of more than one program can reside in memory together. This allows the system to work (process) on one program while it is swapping another program. The result is a considerable increase in cpu usage, thus inproved throughput, and at the same time reduced response times. Especially users that had 16K backgrounds will notice the faster response, as most primairy responses require only one or two fields of a program to be resident. A further benefit of the virtual memory implementation is that the restriction to 8K or 16K backgrounds is removed. The system may be configured with anything from 8K to 32K backgrounds, with the only consequence that larger virtual memories require a larger swapfile. An occasional requirement to run a large program now has little or no effect on the general system performance.

Terminal support.

A number of configuration parameters have been introduced to allow you to addapt the system to specific types of terminals. We had already the possibility to insert a number of filler characters after a specific character. New is the possibility to specify for each terminal what code it uses for 'escape'. Some older terminals, notably ASR33, send either 375 or 376, which for newer terminals are legal characters in the lower case column. Now you can for each terminal define its proper escape code. If the system recieves that code, it translates it to the true ASCII escape, code 233, which is properly handled by all software. To support the new RUBOUT protocol implemented in OS/8 V3D, the terminal driver correctly decrements the position counter for a backspace character. Also, you may specify that for one or more terminals backspace (code 210) is translated to any other single character code. Although it will generaly be more convenient to set the terminal parameters at system generation time, they can be changed on-line with simple commands to the CONTROL/B task. To warn users that are typing on video terminals in the case of input buffer overflow, the terminal bell is rung for each character that could not be accomodated in the input buffer. In order to be able to implement these features without increasing the memory requirements, we had to drop the terminal papertape support (ie. reader run is always enabled, you must stop the reader manually to prevent buffer overflow).

Detached terminals.

A new mechanism was introduced that allows you to run more backgrounds than there are terminals on the system. The system can be build with up to 7 backgrounds, even if you have less terminals. The terminals that are present, are connected one to one to the first so many backgrounds. When the system is running, a user may 'hook' his terminal to one of the unassigned backgrounds. His original background remains active, ie. if it was executing a program (BATCH!), it will continue to do so. Any terminal output from that background will continue to appear on the user's terminal. Meanwhile, the user has complete control over the new background. Whenever required, the user may switch back to his original background. Any detached background can be hooked on by any user. Terminal output is always directed to the terminal that was last hooked on. Note that each background requires its own disk area.

Lineprinter support.

Lineprinter support is enhanced by providing automatic spooling internal in the lineprinter driver, plus a number of configuration parameters to specify the type of interface (LE8E, LS8E, KL8E or DKC8AA), linewidth and character set.

Memory requirements.

Although the performance and functionality of MULTI8 have been increased considerably, its memory requirements have not. The implementation of a new terminal buffering system contributes most to this fact. In V6 each terminal had its own fixed buffer area. The size of this area depended on the terminal speed. For a 1200 Baud terminal a total of 400 words was considered the minimum. That means 1K of buffer space for a four terminal system. In practice, at any moment only a small part of this space was used. In V7A a pool of 16 word bloks is maintained that is shared by all terminals. To prevent one terminal from monopolizing the pool, no single queue can ever grow beyond the size of the remaining pool. This technique allows smooth operation of 4 terminals with only .5K of buffer space. In general, this technique halves the systems terminal buffer requirements.

Not only the resident part of the system has been reduced in size, the requirements of the various external emulator tasks have been reduced also. This has been accomplished by a refinement of the 'release devices' mechanism. In V6 all the emulator tasks in the system were RUN each time a user program returned to the keyboard monitor. This made that all emulator tasks were resident most of the time. They were generally programmed to exit with 'release' to prevent the considerable overhead of loading the task for each keyboard monitor entry in the system. In V7A emualtor tasks enter their name (or rather task control block pointer) dynamically in the table of emulators that are run when a user program is finnished. Thus only emulator tasks that are realy active, either for the exitting user or another user, are RUN.

The net result of these improvements is that you may run a system with up to 7 terminals in an 8K-foreground configuration. On a 32K machine this leaves 6 fields for background operation, which gives the system sufficient space for smooth virtual memory operation. If you have many peripherals on the system and/or need considerable foreground space for realtime tasks, you may need to allocate 12K for the foreground. It is, however, possible to temporarily lock one or more fields of a background program in core so that they can be used as buffer space for a foreground task.

Terminal speed.

A few remarks on the issue of terminal speed (Baud rate) are in place. With the advent of modern video terminals, the traditional 110 Baud terminal speed has been replaced by much higher rates. A temptation exists to set the terminal speed as high as the hardware supports, generally 9600 Baud. However, you should realize that this essentially puls the chair from under the timesharing concept. The rate at which computer systems can produce output is generally limited to a few thousand characters per second. Especially timesharing systems that implement a flexible character-oriented dialogue with the user, have a significant amount of work to do for each character that goes out. If you run your terminals at 9600 Baud (960 characters per second), two or three active terminals will completely drain the cpu resource. The system will not be able to keep the output buffers for the terminals filled and the output will appear in bursts. This is unpleasant to look at for the users. The system overhead is increased, because the programs are rescheduled very quick, which gives rise to increased swapping.

If you run your terminals at a medium speed (1200 or 2400 Baud is suggested), the system will at most times be able to keep up with the output. Characters appear on the screen as a smooth stream, which gives the user the impression that he is beeing served continuously. When the system has filled an output buffer, it can set that background aside for several seconds to deal with other users. There are but few situations that require a high transmission rate, eg. a graphics terminal. Note that there are in fact different types of terminals that give very different loads to the system. On one extreme, a terminal used by inexperienced users to run some simple standard programs or perform editing of small files, will only lightly load the system. A PDP8 could easily handle 20 or more of these. On the other hand, experienced programmers using complicated programs, large files, screen editors, will at times ask for a faster machine, even when they are the sole user! The latter type of user will certainly complain if he has to share the machine with more then 3 or 4 colleagues.

Building procedure.

Much effort has been devoted to easy the building procedure. The many sometimes akward command sequences one had to follow to build MULTI8 V6 have been replaced with a simple and consistent procedure. Many things have been automated; the system automatically creates the file that holds the task images (in fact, they are stored in an extension of MULTI8.SV), the swapfile (SWPFIL.M8), origined at a track or cylinder boundary. The device length table in PIP is automatically updated to reflect the lengths of the user disks. Many of the former configuration parameters are now assigned default values, eg. the interrupt priorities, that will prove adequate for most installations. At startup-time, all user disks are automatically initialized with a fresh copy of OS/8.

A special program is supplied that interactively determines the system parameters and generates suitable parameter- and batch-files. Also, the taskbuilder (which is now part of the initialization code) is able to apply patches to system tables, based on information in the binary task files. This removes the need to edit the monitor source files in most of the cases, even for user-added tasks and devices.

Reliability

The system is protected against spurious interrupts that can not be cleared by the normal code in the skipchain. If the system loops more than 4096 times through the skipchain, it enters a wait-loop for approximately one second and then resets the machine (Clear All Flags instruction). After reenabling the system clock, it tries to resume processing, which may succeed or not, depending on many factors, such as what peripherals were running when the fault occured.

The machines memory size is dynamically determined at startup time. In fact if the foreground fields plus the first background field are present, the system automatically excludes fields that are missing, eg. because of a memory failure. If a memory failure occurs, stop the system, remove the defective memory, insure that the field 0 to 3 (4 for 12K foreground systems) are present, and start the system up again. It will run as long as 3 background fields are operable.

Converting user tasks.

The new version of the MULTI8 System Manual describes the new protocols inplemented in V7A. Here we give a list of modifications that must be made to tasks written by current users of V6.

- The second word of the Task Header (the task length and connect information) may NOT be used as a scratch location; its contents must be preserved to allow the system to properly execute the RELEASE and SWPOUT functions.

- If bit 5 of the task length word is set, the task is RUN when the system starts. The RUNTAB in the powerup code has been removed.

- Driver tasks must be prepared to handle a close call. A close call is identified by the fact that the AC is zero (for a normal call, the AC contains a pointer to the request paramaters). One way to handle close calls is:

```
        SNA             /CLOSE CALL ?
         JMP CLOSE      /YES
        .....

CLOSE,  ACM1            /RETURN EVENTNUMBER -1
        JMS MONITOR
         EXIT SWPOUT
```

Note that a negative event number, when used in a WAIT, results in an immediate return, with the AC set to the complement of the event number.

- The subroutine RESERV has been replaced by a monitor function RESERV. This function may be combined with RETURN and its options. In blockdrivers the following sequence is very usefull:

```
        ....
        JMS MONITOR
           RESERV RETURN CONTINUE
        DCA EVENT       /THE EVENT WHERE WE WILL SIGNAL COMPLETION.
                        /LINK AND DATAFIELD ARE PRESERVED
```

In one monitor call this reserves an event, returns that event number to the calling task, and delivers the event number to the driver task as well.

- KILL has been replaced by BREAK. If you had tasks that could only be stopped by KILL, you have to convert them. These tasks should periodically execute

```
        JMS MONITOR
           BREAK
           0            /0=TEST MY OWN BREAK FLAG
         JMP STOP       /FLAG WAS SET, DO SOMETHING...
        .....           /FLAG WAS NOT SET, CONTINUE
```

- The foreground break character has been changed to CONTROL/F (formerly CONTROL/SHIFT/L or CONTROL/BACKSLASH).

- Tasks that needed modifications to system tables can have these modifications done at task-build time. A task may be preceded by a so-called task preamble. It consists of one or more patch-groups. Each patch-group has the following form:

```
        *0              /MUST BE HERE FOR EACH PATCH GROUP
        CDF 10          /FIELD WHERE PATCH MUST BE MADE
        EMTAB+33        /ADDRESS TO BE PATCHED
        "Q^100+"Q&3777  /VALUE TO STORE
```

In the example, the task name QQ has been stored in the 33rd location of the trapped IOT dispatch table. Any location can be patched by this methode.

- Emulator tasks are no longer permanently entered in ASEMTB, the table of emulator tasks that are RUN when a user enters the OS/8 KBM or CD. Emulator tasks that work for more than one call for a single user must enter their name (or, more efficiently, their TCBP) in

ASEMTB dynamically. The following code could be used for this task:

```
        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 ITS 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 its 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
```

- When the user enters the OS/8 KBM or CD, his emulator task that have entered their name or TCBP in ASEMTB are RUN, but not with AC=0, but with L=1. The AC will always point to the BG's data area in field 1. This makes it possible for one emulator task to be active for several BGs, and still receive control when a user leaves his program. For example, the task might control access to a common data base, and have a list of locked record numbers for each active user. The new protocol allows such a task to unlock all records that were in use by a user who terminated his program.

- On return from an emulator task, the AC signals 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 (UFLDO          //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 remains 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 are in the instruction field, ie. the field where the trapped instruction resides.

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

```
        CDF 10              //YOU KNOW WHY!
        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
        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.

- With some simple equates in the configuration file you can included extra entries in the skipchain with customized code for your pet device. As an example, you could make the following equates for an extra KL8E:

```
        UDEV1=420           /DEFINE SYMBOL 'UDEV1', AND GIVE IT THE
                            /VALUE OF THE DEVICE CODE.
        SKPDV1=6001+UDEV1 /THIS IS THE SKIP-IOT
        CLRDV1=6002+UDEV1 /THIS IS THE CLERA-FLAG IOT
```

these definitions (in CONFIG.PA) result in the following entry in the system's skipchain:

```
*DEV1^4+INT                 ///EVENT NUMBER IS 'DEV1'
        SKPDV1              ///YOUR DEFINED SKIP-IOT
         JMP .+3            ///
        CLRDV1              ///YOUR DEFINED CLEAR-FLAG IOT
        JMS I ZHRDINT       ///SIGNAL EVENT FOR 'DEV1'
```

You can use this entry in your tasks either by connecting to it, or with a simple

```
        JMS MONITOR
           WAIT
           DEV1
```

Four of these dummy entries have been included in the system code, UDEV1, UDEV2, UDEV3 and UDEV4, with their corresponding SKPDVn, CLRDVn and DEVn. The first pair if the dummy entries (UDEV1 and UDEV2) are located at the very beginning of the skipchain, and thus have the highest interrupt service priority. UDEV3 and UDEV4 have medium priority.

Disk configuration.

The idea of 'virtual disk files', as used in MULTI8 V6, has been abandoned. It had only restricted aplicability and introduced extra problems, eg. all disk drives known to OS/8 had to be online for a MULTI8 startup.

Now that MULTI8 V7A initializes user disk automatically, it is not longer necessary to have access to user disks from OS/8. This makes it possible to use a more general scheme for user disk allocation.

The default allocation on a RK8E or equivalent system is as follows:

```
        DSK0=RKA0              DSK4=RKA2
        DSK1=RKB0              DSK5=RKB2
        DSK2=RKA1              DSK6=RKA3
        DSK3=RKB1              DSK7=RKB3
```

If one or more of the generic names (eg. RKA2) exist in the OS/8 configuration used when MULTI8 is started, this (these) disk(s) will be accessible under these names for all users. They are called 'public disks', and can be used for common file storage, as user mountable data disk, etc.

You can change the default disk layout by defining parameters in CONFIG.PA. In fact, for each user disk there are four parameters that define where that disk area is located and how large it is:

```
USYSn           name of device driver, default ="S^100+"Y&3777.
UNITn           unit number, default =n.
FROMn           starting block number, default =0.
SIZEn           length of user area, default =size of system disk,
                eg. 6260 for RK8E.
```

As an example, the following definitions will devide RKB0 in to parts for user 1 and 2. User 1 gets the largest area, 4000 (octal) blocks. User 2 gets the remainder, 6260-4000=2260 blocks.

```
SIZE1=4000      (USYS1, UNIT1 and FROM1 are default)
UNIT2=1         (to locate DSK2 on RKB0 instead of RKA1)
FROM2=4000      (start where DSK1 ends)
SIZE2=2260      (length that remains for DSK2)
```

to complete the example, assume that this is a 2-drive RK8E system (2 RK05J or a single RK05F). Assume that the second drive (platter) should be used for common file storage. Generate an OS/8 system (with BUILD.SV) with the following devices:

SYS, RKA1, RKB1, TTY, LPT, etc...

Note that RKB0 is left out. In general you should delete the OS/8 devices that coincide with the user disk areas, to prevent inadvertent access to user disk area's.