SHARP PC-1500/A & TANDY PC-2

```
M     M  M     M  PPPPP    SSSS
MM   MM  MM   MM  P    P  S    S
M M M M  M M M M  P    P  S
M  M  M  M  M  M  PPPPP    SSSS
M     M  M     M  P            S
M     M  M     M  P       S    S
M     M  M     M  P        SSSS
```

```
Mini
         Multi-
                  Process
                         System
```

A small and powerful system

version **9-Apr-1998**

Christophe GOTTHEIMER

January 1994-January 2014

DISCLAIMER

This software is expected to run exclusively on a SHARP PC-1500 or PC-1500A or a TANDY PC-2. It will not run on SHARP PC-1600. To use it on an emulator, be sure that the LH5801 instructions related to the **V** or **MN** register are also emulated, else some kernel-jumps or some utilities may not work correctly and launch a crash.

This software is very complex. It is assumed to be standalone and requires all the memory provided by **CE-161** 16Kbytes extension. **It is NOT relocatable**.

****

I am not responsible for any damages, hardware or/and software, resulting from the usage of MMPS, its run-time, the standard executable utilities, and any processes developed later.
You are using this software at your own risk !

****

Be sure to perform a save of your important data before installing and running MMPS.

Fee free to report bugs, remarks, suggestions, ... to **cgh75015@gmail.com**

What is MMPS ?

MMPS is a small, but powerful system, which allows dynamic pages allocation, file system management and multi-processes schedule. It runs on SHARP-PC 1500 computers with 16KB of memory.

MMPS is completely written is assembly language using the monitor-assembler-debugger XMON(R).

MMPS is standalone and uses only for routines from the BASIC ROM. Even MMPS does not touch to a BASIC area, it is not compatible with the BASIC programs and variables.

The current directory contains:

COPYING	The GNU Public license
README	A small text file for impatient ;-)
MMPS.inc	MMPS header to develop with the **lhTools**
MMPS.macro.inc	MMPS macros for the kernel
MMPS.volatile.inc	MMPS volatile variables and address
mkmmps.sh	A script shell to build BIN and WAV
mmps-090498.asm	The assembly code of the MMPS kernel
mmps-fs+lib.asm	The assembly code of the FS and utilities
mmps-volatile.hex	The assembly source code of the MMPS volatile area
mmps-volatile.ds.hex	Volatile DS table code
mmps-volatile.heap.hex	Volatile HEAP area and table code
mmps-volatile.proc.hex	Volatile process, FS tables and kernel stack code
mmps.abw	The whole documentation for **AbiWord**
mmps.pdf	The whole documentation in **PDF**
BDGRD.asm	An example of executable
RUNPRO.asm	An example of executable
SHOWKEY.asm	An example of executable
RTLIB.asm	The RUN-TIME Library
STARTER.asm	The STARTER executable
FILES.asm	The FILES executable
PROCESS.asm	The PROCESS executable
DISPLAY.asm	The DISPLAY executable
DEBUG.asm	The DEBUG executable
MAKER.asm	The MAKER executable
Images/stddev/	All images WAV/BIN for the Standard development
Images/stdfull/	All images WAV/BIN for the Standard full
Images/1500Adev/	All images WAV/BIN for the 1500A development
Images/1500Afull/	All images WAV/BIN for the 1500A full
Images/Examples/	All images WAV/BIN of the examples

To rebuild the WAV of MMPS, use the script **mkmmps.sh**. To process, the **lhTools** version **0.4.4** or higher are mandatory, and so the **PocketTools**.

The current MMPS Kernel is **09Apr.1998**

Overview of MMPS

MMPS (**M***ini* **M***ulti*-**P***rocess* **S***ystem*) is a little kernel, which can run several processes, keeps the user data stored into files, provides some I/O possibilities like Virtual Consoles, IPC (**In**ter-**P**rocess-**C**ommunication), sending-and-receiving messages and events, pages allocation and release and owner resources.

The MMPS kernel may be split in 3 levels:

1. the page manager
2. the file system
3. the process scheduler

The page manager

It is the lowest level. It is totally independent from the others, but called by them each time an allocation or release is needed.

It is in charge of the pages allocation and release. Under MMPS, each memory unity is called 'PAGE'. A page has a size of 64 bytes and is always aligned on a 64-bytes frontier. It can manage up to 127 pages. A set of 1 to N contiguous pages is called 'BLOCK'.

When a allocation is done, the free block may be split into 2 blocks: one allocated and one kept free. On a same way, when release is done the freed block will be concatenated with the previous if it is free, with the following if is free. So only one big free block may result of the coalescence.

The file-system and the user processes ask for blocks. The blocks are owned by the calling process and keep its ownership until it releases them or it is ended.

In the kernel, the 2 higher levels ask for blocks allocation: the file-system to build the entries and to reserve the space needed by them; the process-scheduler to creates the stacks, the arguments pages and the messages transmission.
The kernel is working with physical addresses, and these addresses will be returned to the caller. No virtual memory access is yet possible (except a similar mechanism with the SHARED entries).

A process has to take in charge its own memory management and to avoid to access outside of its space. The kernel has no security against a process which will go out from its space, neither against a process which want to access to some forbidden or private areas.
If a process tries to access to a non valid address, the kernel will not be warned. But accessing to the data outside the process space may cause some irrecoverable crashes. If a process wants to access to an external areas, especially while writing, it has to be sure that no other processes may access to this area.

The pages allocated to the file-system are kept until it asks to release them. The processes stacks, messages and blocks are also in the same physical memory. After a lot entry creations, deletions, pages allocations, messages sent and processes started, it may occur that the memory map is very fragmented, and that no big blocks (composed by a big set of contiguous pages) are possible to allocate. In this case, nothing to done except to write a code to defragment the memory map. There is also no algorithm to allocate the pages, except that the closest block (in memory pages number) is fetched. When a block preceding and following two free blocks is released, the three blocks will become one block containing the sum of the three blocks pages. In this version of MMPS, no memory usage information is available.

The file-system

It is the second level. It is in charge of the entries management and the data storage. All the entries in the file system are named. The name is a set from 1 to 9 characters. The entries have also a type; 7 types are recognized by the file-system: the 'FILE', the 'CONSOLE', the 'QUEUE', the 'SHARED' and the 'LOCK'.

- The FILE is a block which contains code, text or data. It is used to store the executable code of the utilities, the sources, the texts or simply some data.
- The CONSOLE is a logical link to the physical DISPLAY and KEYBOARD of the computer. Several CONSOLE may be opened at the same time, but only one is active and linked to the DISPLAY/KEYBOARD device.
- The QUEUE is a channel between several processes; the process which created it is owner and has the READ access. The others have only the WRITE access. It is used for asynchronous inter-processes communication.
- The DUPLEX is a connected channel to a peer queue. Writing to a DUPLEX sends the message to the peer DUPLEX (read). It is an implementation of the PIPE. A DUPLEX is only possible between 2 processes.
- The SHARED is space in the memory which may be acceded by several processes, to shared some data. Only the action of READ and WRITE are possible on the SHARED entries. The shared are equivalent to a memory space with a virtual address access.
- The MAP is the export of a memory region as a file entry. The access right to the MAP are set when creating it.
- The LOCK is just a specific entry and may be used by a process to lock a critical resource.

The file-system allows also access rights control. Like the system does not support the real-ownership, these rights are just used to avoid some operations on the concerned entry. The following rights are supported: READ, WRITE and EXECUTE. An EXCLUSIVE open option is also possible to avoid to share an entry with another process. The CONSOLE and LOCK are always opened exclusively.
A link counter is also provided to avoid some operations on some case of entry, like deleting an opened entry.
For the I/O actions, a process needs to get a descriptor on an entry. It is done by 'OPEN'ing the entry. When it finishes its work, the descriptor has to be 'CLOSE'd. A descriptor is allocated by the file-system which stores on it some information like the open-mode, the entry type, the owner, the node-link and some other information depending of the entry-type. The descriptors owned by a process are closed at the process termination. Opening an entry will increment its link number and closing will decrement it. To open an entry, the calling process has to give the 'OPEN MODE', which will define the actions possible. Some actions are specific to the entry type and some other are common. This is very easy to write a transparent code, independent of the descriptor type. However, the LOCK and SHARED are always managed in a separate way.
An extra set of specific actions is also provides for the CONSOLE to manage the I/O.

The file-system can manage up to 32 descriptors (no limitation by type), and up to 64 entries (also without limitation by type).

When a process wants to create an entry, the file-system will ask to the page-manager to reserve some pages for it. These pages will be released when the entry will be removed. Closing some entries like CONSOLE, LOCK, QUEUE or DUPLEX will remove them.
The major constraint is that an entry has always a contiguous set of pages. So, the 'CREATE' action have to specify the number of pages to reserve for the new entry. The QUEUE, DUPLEX and the SHARED entries have a limit of 4 pages. The CONSOLE entries have always 1 page.
The FILE entries may have from 1 to N pages; the number of pages will give the maximum size that the FILE may have; a real-size is given which indicates the real count of bytes stored. An action to re size the FILE entries is provided.

The process-scheduler

It is the third and the highest level. It manages the process start, the process time slot allocation, the inter-process messages and the events control. It can create up to 8 processes, with priority and father return. When a process starts a new one, it becomes the 'FATHER' and the started process the 'SON'. A process is an entry in the process table, marked as active and recognized by the scheduler which will restart it. The scheduler allows some time slots to the processes; this number of slots given depends of its priority. The priority is a number from 0 (highest priority) to 15 (lowest priority). The priority of a process is the same as this of its father, but it can change it. A process of priority 0 will get 16 consecutive slots for its execution, and only 1 if its priority is 15. All the slots have the same period. The default priority is 8, so 8 slots.

All the processes started receive an unique number called process-id (PID); this process-id permits to another process to access to the named process.
The information recorded into the process entry are the process status (PAUSED, WAITING, ...), its priority, its stack pointer SP, the delay counters (PAUSE and TIME), the global address, its father process-id (FID) and a link to a FILE entry in the file-system.

When the scheduler starts a new process, it first asks to the file-system to find the named entry, checks if the type (FILE) and the access rights are good (READ and EXECUTE). After, it reads the 6 first bytes of the FILE to find some specific information like the executable signature, the number of pages to allocate for the stack and the global area. The father gives to its son the in and out descriptors and an address to a page called arguments page which will be copied into the son process space; the father can pass up to 63 bytes in the arguments page, the last is always 0. The FILE header contains how many pages should be asked for the stack and the offset of the global area. When the scheduler will start a process, it will create its stack. This stack is built in the new process space. This space is allocated and has the number of pages needed by the process (excluding the arguments page which is allocated in same block. In fact, the scheduler add 1 to the number of pages), add the offset of the global area. The global area is the memory area allocated to the process after the top of the stack.
This space is very important, because its address is passed to the son process at the beginning. It is used to store some global data which may be used later. The address GLOBAL - 2 contains always the address of the arguments page. If the space area is allocated and the in and out descriptors are valid, the son process is created and started. The father gets as return the process-id of its son.
When its son will terminate, the scheduler will warn the father by using an 'EVENT'. An EVENT is in fact an action which will ask to the scheduler to send an event message into the event queue of a process. The receiver process has to poll its event queue to get the oldest event. A process may enable or disable some events by changing its EVENT mask. On MMPS, 8 events are supported: 4 are reserved for the scheduler and the 4 others are free for the other processes.

The 4 events reserved by the scheduler are:

- 4 The son has returned to scheduler. In this case the value DE of the son is returned to the father. The scheduler 'PUSH'es a return address into the stack where the process has to return when it is finished. A simple RET is needed to return from a process.
- 5 The son was terminated by an external event.
- 6 The time alarm is reached.

7 The BREAK key was pressed on the CONSOLE linked to this process. This event is particular, because if it is not handled, the process will be terminated. It is a case to raise an event 5 to its father.

The 4 other events (0 to 3) are without specific meaning, and are free for usage. When raising an event 0 to 3, the DE value of the calling process will be passed to the triggered one.
While polling its event queue, a process will get some information from the kernel, the number of the event and the process-id of the process which raised it, and eventually the value of DE from the raising process. A process can raise an event to itself. The event queue is filled even the process is suspended.

Another feature for process communication provided by the scheduler is the messages. To use them, a process is the receiver and another the sender. When a process wants to receive some message, it asks to the scheduler to give to it the state 'READY-TO-RECEIVE' and will be suspended until a message will be sent. When a process wants to send a message to another, it asks to scheduler to send it. This is possible only if the target process is ready. A message is a set of bytes which will be copied from the sender space to the receiver space. The block needed is allocated by scheduler and returned to the receiver which will be restarted. The messages mechanism provides tools for synchronous inter-process-communication. A process can also SEND-and-RECEIVE a message; it can be useful for a message with acknowledgment. A message is a set from 0 to 255 bytes.

When a process is finished (returned or terminated), all its resources, descriptors and pages will be released, and its father will be warned. If a process has no more father, the FID will be set to the value FF. When a father is terminated, its son are still running and are not warned.

The scheduler

The MMPS scheduler will be triggered at each interruption, when a process call the **RCH** kernel-jump, or at each kernel-jump entry.

The scheduler is in charge to decide what process is ready to run and so to give the hand to it, or what process need to sleep according its state.

To run a process should be the next "*eligible*" process, that is the first process ready in the scheduler queue. The following status bits set the process sleeping until the "*waiting event*" is completed:

READY-TO-RECEIVE : The process is waiting for a message (**REC** or **SNR**); It will be rescheduled when a message is received from another process (**SEN** or **SNR**)

PAUSED : The process is paused for a delay (**PAU**); It will be rescheduled when the delay has elapsed

QUEUE-WAITING : Waiting for a message in a queue (**WAI**); It will be rescheduled when a message is written in the queue (**WRI**)

LOCK-WAITING : Waiting for owning a lock (**TAK**); It will be rescheduled when the lock becomes given (**GIV**) is taken by the process

TRACE : Waiting for trace control (DBG); it will be rescheduled when the debugger leader will request the execution of the next instruction (**TRA**)

Each time the scheduler runs a process, it looses one slot. When all slots are consumed, the process is put at the tail of the scheduler queue. Under MMPS, a kernel-jump consumes one slot. The number of slots allocated to a process may be changed by the kernel-jump **PRI**.

The **RCH** kernel-jump put directly the calling process to the tail of the scheduler queue.

The console task

The console TASK is in charge of the management of the keyboard and the screen devices.

It runs when a process calling a console request, input (**REA**) or output (**WRI**). It has the privilege to run "in place" of the process with its PID.

In MMPS, the time allocated to the console TASK is the sum of all the slots of all process requesting on a console. It is not a very nice implementation choice, and it may be evolute later.

The LCD indicators are set or cleared depending of the pending requests:

- **I** : input request
- **II** : output request

The console TASK handles the following keys:

- **SELECT** : Select the next console as active*
- **OFF** : Display while the OFF is pressed the PID of the process owning the console, the process name and the console name as follow:

  ```
  P NNNNNNNNN CCCCCCCCC
  ```
- **ON/BREAK** : Send an event KIL to the process owning the console, or kill it if the process do not handle the event

If an output (**II** is set) request is pending:

- **RCL** : Clear the pending output (II is cleared)

If an input (**I** is set) request is pending:

- **left-arrow** : Go back one character
- **right-arrow** : Go forward one character
- **shift-DEL** : Go back to begin of the line
- **shift-INS** : Go forward to end of the line
- **CL** : Clear from cursor to the end of the line
- **shift-CA** : Clear all
- **ENTER** : Terminate the input request, all characters in the line are returned to the calling process
- **down-arrow** : Terminate the input request, all characters from the begin of the line to the cursor are returned to the calling process
- **up-arrow** : Return end-of-file error (**CARRY** is cleared and **&D3** is returned into CPU register **A**)

Note : The* ‘SELECT’ *key is the ‘double up-down arrow’ key on the left of the* **RCL** *key.*

Installing and starting MMPS

Before to install MMPS, be sure to have saved your important programs or data, because MMPS requires a large amount memory. See the last chapter for the initialization and discussion about the different MMPS images provided.

After, a load from the tape using **CLOAD M** is first needed. This is done in 3 steps:

1. the MMPS kernel (see '**MMPS 9/04/1998**')
2. the run time and the utilities within the minimum file system (see '**MMPS FS+LIB**')
3. the initialization volatile area (see '**MMPS VOLATILE**')

When the system loaded into the memory, it may be called directly by the BASIC instruction **CALL &C5** under the editor or a program.

The file-system and the volatile data are kept when exiting MMPS. To start MMPS again, just do **CALL &C5** under the BASIC editor, or in a program.

When starting, the system will first perform a 'DOWN'; it means that it will terminate all the processes, close all the descriptors and release all the blocks and pages not reserved for the file system.
After it will create a CONSOLE entry named 'CONSOLE', put it as the active one and try to start the process 'STARTER' on this console.
If one of these actions failed, the kernel will provoke a computer soft reset and give the hand back to the BASIC. The message '**NEW0? :CHECK**' will be displayed. In this case, a trace-back is written in binary into the variable BASIC **E$** (at address **&7650**). The binary data should be retrieved using the **PEEK** function or under a monitor. The **E$** content is the following:

FF AA BB CC DD EE HH LL PCPC SPSP

If all has worked, **STARTER** will display the MMPS banner (see '**MMPS 09Apr.1998**') and be waiting for a command. To clear a message pending on a CONSOLE, use the '**RCL**' key.
STARTER assumes that all the commands given are the name of a process to start followed by its arguments. If the process is started, **STARTER** will display '**>***N*' where *N* is PID of the process. If the process can not be launched correctly, **STARTER** will display '**!***EE*' where *EE* is error code returned by MMPS.
When a new process starts, **STARTER** activates and selects the console to it. To change the active console, use the '**DOUBLE-ARROW**' key located on the left of '**RCL**'.
When a process is normally finished, **STARTER** catches the event and displays '**>***N* **+4:***RRRR*' with *N* the PID of the ended process and *RRRR* the returned value.
When a process is killed, **STARTER** catches the event and display '**>***N* **-5**' with **N** the PID of the ended process.
Another **STARTER** process may be started. It is finished by pressing the '**UP-ARROW**' key. In this case, the process just returns to the scheduler except if **STARTER** has the PID 0 (that means that it is the son of the kernel) and in this case it will down the system. When the system is 'DOWN'ing, it will terminated all the processes, close all the descriptors and release all the pages. After, it will restore the BASIC stack and give it back the control. It is not possible to start more than one process neither to start another process **STARTER** at the 'BOOT' time (except by writing another process and give to it the name '**STARTER**').

Using MMPS

All the processes may ask to MMPS to do a specific action, like allocate some pages, create an entry, read, write, start a process.

A process dialogs with the kernel through a 'KERNEL JUMP'. On MMPS, a set of 62 kernel jumps is provided.

When a process wants to do a kernel-jump, it has first to fill the 8-bits registers B, C, D, E, H, L or 16-bits BC, DE, HL or mixing the both types with the arguments required by the kernel-jump (some kernel-jumps do not need argument), and after to 'JUMP' to the kernel with instructions '**LDA** *kernel_jump_number*; **SBR &E3**'. When the 'JUMP' will return to the process, the kernel executed the request or returned an error if something went wrong. The process is suspended during the kernel-jump and restarted when it is finished.

All the kernel-jump use the same convention: at the return, some registers may be modified if the kernel-jump has to send back some values to the caller process. For all kernel-jumps, two cases may occur:

- The kernel-jump went good, the eventual values are returned into the registers and the CARRY flag (**C**) is set (**SCF**). The register **A** contents the kernel-jump-number. The others registers **BC**, **DE** and **HL** may be modified depending of the kernel-jump.
- The kernel-jump did not go good, the CARRY flag is clear (**RCF**) and an error code is returned into **A**. In this case, no registers except **A** are modified.

A kernel-jump is always executed in one time, not-preemptive and not-interruptible. A kernel-jump is always finished before to restart a process. The kernel-jumps are executed sequentially and by the kernel itself which takes the PID of the calling process.

A kernel-jump begins and ends by a process rescheduling. When a process asks for a very short kernel-jump like getting its PID, it looses one slot, the same as a process which asks for a long one like starting a process. This is not very equitable for the process, the very short or long kernel-jumps are very rare and with an almost equal ratio.

Here after is the list of all the kernel-jumps supported:

BAL - Allocate a block
BCH - Check block validity
BFR - Release a block
BRE - Change the size of a block
FSR - Read the file-system
FEN - Get status of a file-system entry
TMP - Create a unique temporary entry name
LOK - Create and open a LOCK entry
CON - Create and open a CONSOLE entry
FIL - Create and open a FILE entry
QUE - Create and open a QUEUE entry
DPX - Create and open a DUPLEX entry
SHR - Create and open a SHARED entry
MAP - Create and open a MAP entry
DEL - Delete an entry from the file-system
ACS - Change the access rights of an entry

RNM - Rename an entry
SIZ - Re size a ENTRY entry
DSN - Get entry name from an opened descriptor
DES - Get entry type from an opened descriptor
OPN - Open an entry and get a descriptor
CLO - Close a descriptor
REA - Read a buffer from a descriptor
WRI - Write a buffer to a descriptor
TEL - Get the current position of a descriptor
SEK - Set the current position of a descriptor
WAI - Suspensive wait from a descriptor
CNT - Connect a DUPLEX to a peer DUPLEX
GET - Get data from a SHARED/MAP entry
PUT - Put data to a SHARED/MAP entry
GIV - Give a LOCK
TAK - Take a LOCK
SCN - Set a CONSOLE descriptor active
FLU - Flush a CONSOLE output
RFR - Refresh a CONSOLE output
EDI - Start an editable REA from a CONSOLE descriptor
HDR - Build a executable header
GBL - Get the global area address
PID - Get the PID of the calling process
FID - Get the father PID (FID) of the calling process
XEQ - Execute a new process
END - Terminate a process
PRI - Change the priority of a process
STA - Get the status information of a process
PAU - Pause the calling process for a delay
EVT - Create the event queue to receive the asynchronous events
KIL - Destroy the event queue
MSK - Set the events mask
POL - Poll the event queue and get the oldest event if some are present
RAI - Raise an event to a process
TIM - Initialize a TIME delay
ABT - Abort the TIME delay
SEN - Send a message to a process
SNR - Send and receive a message to/from a process
REC - Receive a message from a process
DBG - Execute in debug mode a new process
TRA - Trace the next instruction of a debugged process
REG - Get the registers of a debugged process
COD - Get code of a debugged process
DAT - Get data of a debugged process
RCH - Reschedule
DWN - Shutdown the system

Constraints and limitations

MMPS is a small system and comes with some constraints and limitations.
Some of them are due to the use of some routines coming from the BASIC ROM. The others result from the choices made to represent the tables and the internal kernel data. This choices were always made in a way to 'OPTIMIZE' the accesses to these data and also the space to store them.

MMPS was developed on a SHARP PC-1500 computer with 16KB of memory.

The specifications asked to keep free 8KBytes for the working area. The kernel-code having to be less the 8KBytes. It had also to be compatible with the BASIC, and for that never use the BASIC variables (except the trace-back into **E$** in case of boot problem) for its work.

So the memory 'MAPPING' was the following:

All images:

&0000-&00C4	Area for the RESERVE mode.
&00C5-&17FF	MMPS kernel code
&1800-&1FFF	Run-Time and utilities (**STARTER**, ...)

Standard Development (Heap 1536 bytes, 24 pages, 32 descriptors):

&4000-&41FF	MMPS private volatile data
&4200-&47FF	MMPS heap area (process, file-system...)

Standard Full (Heap 8128 bytes, 127 pages, 42 descriptors):

&2000-&3FFF	MMPS heap area (process, file-system...)
&4000-&42FF	MMPS private volatile data

1500A Development (Heap 4096 bytes 64 pages, 42 descriptors):

&4000-&4FFF	MMPS heap area (process, file-system...)
&7D00-&7FFF	MMPS private volatile data

1500A Full (Heap 8128 bytes, 127 pages, 42 descriptors):

&2000-&3FFF	MMPS heap area (process, file-system...)
&7D00-&7FFF	MMPS private volatile data

Constraints

The highest constraint is that when a process wants to create a FILE, QUEUE or SHARED entry, it has to know the maximal number of pages that the entry will need. Even it is possible to re size an entry, it is just possible to decrease the number of pages reserved and not to increase it. With this architecture, all the FILE, QUEUE and SHARED entries are contiguous. It is a great advantage for the processes, because the executable-code is not loaded when it is started, but executed directly from the FILE area. It also permits to avoid to the file-system to have to manage a indirect block allocation to store the data, and give a very fast response time. But this representation obliges to write a complete relocatable code and data access, because a process can never know at what address it will be executed, and where its data will be allocated. In this way, the code has to be fully re-entrant.

Another constraint is that MMPS does not support a virtual addressing mode. It is working with the physical addresses and does not have any security against a process which goes out from it space or accesses to a another process's memory area. The programmer has to take in charge the management of the memory mapping for his process.

****** WARNING ******
A process which will write to another areas than these allocated to it may cause some irrecoverable crashes, because it has modified some stacks or some data in the kernel private area. After a crash, the best way is to reload the kernel from the tape and the others files from the backups, because a non-controlled write may change also some instructions in a code.
****

Limitations

There are some limitation under MMPS. These are:

- 127 pages of 64 bytes are available, so 8128 bytes.
- The file-system can manage up to 16 pages, be a 64 entries (no limitation by entry type). A set of 42 descriptors is available. It permits to open up to 42 entries (no limitation of descriptors per process).
- The QUEUE and SHARED entries may have a maximum of 4 pages allocated.
- In the MMPS file-system, the entries names are limited to 9 characters. No directory tree is supported.
- The process-scheduler can start up to 8 processes at the same time.

A standard development version is limited to 24 pages and 32 descriptors. This is due to let a large space available for the development tools (XMON) and for the work-in-progress.
For the PC-1500A, the 1500A development version is limited to 64 pages.

Known bugs

At this time, a real bug exists: if the auto-power-off timeout is elapsed under the **EDI** or **REA** kernel-jumps (waiting on a console), MMPS will be frozen when powering on, because the scheduler is not restarted. So, press very fast the RESET button on the reverse side of the PC-1500/PC-2.
There is no way to bypass this problem. So be careful.

When running a process under debugger, the son process termination does not wake-up the debugger leader. Use the **ON/BREAK** key in the debugger console to kill **DEBUG**.

Terms on MMPS

On MMPS, some terms are used to indicate some entity. Here after is an alphabetic list and the explanation about these terms.

BLOCK A block is a set from 1 to N contiguous pages. Like it is composed of pages, it is always aligned on a 64-bytes frontier. A block is owned by a process (or free). When a process asks for N pages to allocated, the kernel will create a block.

CONSOLE A console is an entry-type in the file-system which represent a logical link between a buffer (1 page) and the physical DISPLAY-KEYBORAD device of the computer. On MMPS, the DISPLAY and the KEYBOARD may not be dissociated. A mechanism of virtual consoles allow several processes to shared the physical device; but only one process is really attached to it at one time: in this case, its console it said ACTIVE.

DESCRIPTOR A descriptor is a 'LINK' between a process and an entry. When a process opens an entry, the file-system will return it an unique number, called descriptor; the link process-entry is created. All the actions to do (I/O, ...) will be done through this descriptor. The link is remove when the process will close the entry.

DUPLEX A duplex is an entry-type in the file-system which allows two processes to communicate in a connected way. Writing to a DUPLEX will give data available on the remote side. If a process closes a DUPLEX, the peer will be warned. A queue is like a tube where some data will go from a point (write) to another (read). The read is done sequentially in the writing order; First-In, First-Out (FIFO).

ENTRY An entry in the file-system is a record of information (16 bytes) which permits to a process to allow some pages to store some data. This area is named (9 characters) and has a type. It permits, when a process opens an entry, to define the set of actions possible on this entry.

EVENT An event is an asynchronous information (like a signal) send to a process to warn it about a special action. There are 8 events on MMPS. Some events (4, 5, 6 and 7) are reserved to kernel for warning a process about some performed and completed actions. The others (0, 1, 2 and 3) are free and may be used by the others processes. The events may be accepted or not by a process by enabling or disabling the EVENT-MASK.

FID A FID (father process-id) is the PID of the process which created the calling process.All the processes have a FID. When a process creates a new one, it becomes the father and the started process will be the son. If a father is finished, all its sons will get the FID at &FF. The process created by the kernel get the FID **&AA**.

FILE A file is an entry-type in the file-system. It is composed of a block (1 to N pages) where a process can store (write) or get (read) some data. A file which contains an executable-code may be started as a process. A file may contains every-types of data (instructions, binary-data, ASCII-data) and is always accessible.

FILE-SYSTEM The file-system is the name of a part of the kernel which is in charge of the entries management, the I/O, and the descriptors.

GLOBAL The global address is the base-address of the area allocated to a process after the top of the stack.

KERNEL The MMPS operating-system itself. it is split into three parts: the page manager, the file-system and the scheduler.

KERNEL-JUMP When a process wants to use the features provided by the system, like create an entry, start a son process, allocate a block, it should do it through the kernel. For that, it jumps into the kernel critical procedures: these are the kernel-jumps. A kernel-jump is the entry point of specific part of code. There are 62 kernel-jumps under MMPS.

LOCK A lock is an entry in the file-system used to lock a resource. No I/O actions are possible on it.

MAP The map is an entry in the file-system used to export a memory block as a file. The foreign processes have right to read and/or write to the map using the standard kernel-jumps, but the owning process may continue to access the memory directly.

MESSAGE A message is a set of bytes copied from the sender space to the receiver space. A message transmission should be synchronized between the two processes. The receiver should be ready-to-receive when the sender wants to send. The messages are used for a synchronous inter-process communication.

PAGE A page is the unity used to represent the memory in the kernel. It is a set of 64 contiguous bytes aligned on a 64-bytes frontier. A page is free or allocated to a process.

PID A PID (process-id) is an unique number from 0 to 7 which permits to access to a process.

PROCESS A process is a executable-code (instruction) and a data area (stack, arguments page...). It is known by the process-scheduler which will allow some execution slots to run. When starting, a process receive an unique number called process-id (PID). All the reference to a process is done with its PID.

QUEUE A queue is an entry-type in the file-system which allows two or several processes to communicate. On process has the read access and the other have the write access. A queue is like a tube where some data

will go from a point (write) to another (read). The read is done sequentially in the writing order; First-In, First-Out (FIFO).

SCHEDULER The scheduler is the part of the kernel which is in charge of the processes management. It handles the timer interruption, creates the new processes, restarts the running processes, allows the slots and perform the messages transmissions.

SHARED A shared is an entry in the file-system which allow several processes to shared a memory area. These processes may access to the shared with a mechanism similar to the virtual memory.

SON A son is a process started by another process (its father). A link is kept between the father and its son: when a process terminates, an event (4 or 5) will be raised to its father to warn it.

Mechanism of the kernel-jumps

On MMPS, the actions executed by the kernel are call kernel-jump.

The system is not a process, but a part of code with an entry point (the kernel-jump) and an exit.

When a process is doing a kernel-jump, it looses one time slots.

After, it enters into a critical code, not-interruptible, and can access (under some conditions) to the kernel private area. The kernel-jumps are executed by the processes itself with its PID.
There is one exception for the virtual-consoles. On the SHARP PC-1500, the keyboard is managed by polling; so, it is not possible to wake-up a process when a key is pressed. Under MMPS, there is an hidden process to manage to I/O on the consoles. This 'ghost' process is started each a process is suspended on an console I/O. It has the particularity to run with the PID of the I/O request owner.

A process enters into a kernel-jump by the 2 assembly instructions:

```
LDA  kernel-jump-number
SBR  &E3
```

where *kernel-jump-number* defines the action to do.
The arguments passed to a kernel-jump are filled into the 8-bits registers **B**, **C**, **D**, **E**, **H**, **L**, or the 16-bits **BC**, **DE**, **HL**. When the process exits from the kernel-jump, the following cases may occur:

- The kernel-jump worked good, the 8-bits and 16-bits registers are modified if they are used to return some values and the CARRY flag is set (**SCF**).
- The kernel-jump detected an error, the 8-bits and 16-bits registers are not modified, the CARRY flag is cleared (**RCF**) and the register A contains the error-code.

Except DWN, all the kernel-jumps return.

Here after is the description of a kernel-jump:

```
LD    BC,bc-argument     ;; load the bc-argument into BC
LD    E,e-argument       ;; load the e-argument into E
LDA   kernel-jump-number;; number of the requested kernel-jump
SBR   E3                 ;; jump to the kernel
JR    NC,error           ;; handle the errors
```

When the **SBR &E3** is executed, it will push the registers into the stack, disable the interruptions, set the kernel-jump bit in the process entry and asks for a rescheduling.

Kernel data structure

Here is described the data structure used by the MMPS kernel. These are the PROCESS-TABLE, the PAGE-ALLOCATION-TABLE, the DESCRIPTOR-TABLE and the entries of the FILE-SYSTEM. These structure are given to permit to write some processes to show information about these structures.

THE PROCESS HAVE TO AVOID TO MODIFY THESE DATA STRUCTURE BECAUSE IT MAY RESULT AN IRRECOVERABLE CRASH.

Some useful addresses:

&xxFF := The pointer of the current process
&xxFD-&4xxFE := Current process PID internally used inside kernel-jump
&xxFC := Console descriptor currently attached to the CONSOLE task
&xxF6-&xxFB:= CONSOLE task status
&xxF4-&xxF5 := Seed for temporary name (**TMP**)
&xxF1-&xxA0:= Internal stack for MMPS and CONSOLE task
&xx80-&xx9F := Pointers table to the block of the file system entries

with **&xx = &40** for Standard * images, and **&7D** for 1500A * images.

The PROCESS-TABLE

This table is located at the address **&4000** and has a length of 128 bytes for Standard images. This table is located at the address **&7D00** and has a length of 128 bytes for 1500A images. It is also readable by opening the entry **SYSVAR**, offset **0x000**. It contains 8 items of 16 bytes which have the following meaning:

```
byte 0: process status (= &FF ::= free)
      bit7 ::= READY-TO-RECEIVE
      bit6 ::= PAUSED
      bit5 ::= QUEUE/DUPLEX WAITING
      bit4 ::= LOCK WAITING
      bit3 ::= CONSOLE-INPUT
      bit2 ::= CONSOLE-OUTPUT
      bit1 ::= TRACED
      bit0 ::= KERNEL-JUMP RUNNING
byte 1: priority and slot-counter
      bit7-4 ::= priority (0 .. 15)
      bit3-0 ::= slot counter
byte 2,3: DELAY counter
byte 4,5: stack pointer SP
byte 6: events mask
      bit7 ::= event7
      bit0 ::= event0
byte 7,8: event queue address
byte 9,10: TIMER counter
byte 11,12: global address
byte 13: father PID and extended process status
      bit7-4 ::= father internal-PID (internal-FID)
            &0 ::= pid0
```

```
            &1 ::= pid1
            &2 ::= pid2...
            &7 ::= pid7
            &A ::= KERNEL
            &F ::= no father
      bit3 ::= not used
      bit2 ::= not used
      bit1 ::= TIMER running
      bit0 ::= DEBUG RUNNING
byte 14,15: physical address of the process entry
```

The PAGE-ALLOCATION-TABLE

This table is located at the address **&**4100 and has a length of 48 bytes (24 pages) for Standard Development image. This table is located at the address **&4100** and has a length of 254 bytes (127 pages) for Standard Full image. This table is located at the address **&7F00** and has a length of 128 bytes (64 pages) for 1500A Development image. This table is located at the address **&7F00** and has a length of 254 bytes (127 pages) for 1500A Full image. It is also readable by opening the entry **SYSVAR**. offset **0x100** (**stddev**) or **0x200** (other). It contains the block items of 2 bytes which have the following meaning:

```
byte 0:     internal PID of the owner
      &00 ::= pid0
      &10 ::= pid1
      &20 ::= pid2...
      &70 ::= pid7
      &AA ::= KERNEL pid
      &FF ::= free
byte 1: number of pages
```

The DESCRIPTOR-TABLE

This table is located at the address **&4140** and has a length of 192 bytes (32 descriptors) for the Standard Development image. This table is located at the address **&4101** and has a length of 252 bytes (42 descriptors) for the Standard Full image. This table is located at the address **&7E01** and has a length of 252 bytes (42 descriptors) for the 1500A images. It is also readable by opening the entry **SYSVAR**. offset **0x140** (**stddev**) or **0x101** (other). It contains the descriptors items of 6 bytes which have the following meaning:

```
byte 0: entry-type and open-mode (= &00 ::= free)
      bit7-4: entry-type
            0000 ::= LOCK
            0001 ::= CONSOLE
            0010 ::= FILE
            0100 ::= QUEUE or DUPLEX
            1000 ::= SHARED or MAP
      bit3: descriptor validity
      bit2-0: open-mode
            100 ::= READ open
            010 ::= WRITE open
            001 ::= APPEND open (FILE)
                    Shared or Duplex bit (SHARED or DUPLEX)
byte 1: descriptor owner internal-PID
```

```
      &00 ::= pid0
      &10 ::= pid1
      &20 ::= pid2...
      &70 ::= pid7
byte 2,3: physical address of the entry
byte 4,5: offset from the top (FILE)
          connected peer DS (DUPLEX)
          DS chained list (LOCK)
```

The FILE-SYSTEM entry

The file-system entry is a structure of 16 bytes allocated into the pages reserved for the FILE-SYSTEM. These pages are dynamically allocated by the file system. Up to 16 pages may be allocated. The pointers to the node pages are stored at address **&4080** to **&409F** for the Standard images. The pointers to the node pages are stored at address **&7D80** to **&7D9F** for the 1500A images. It is also readable by opening the entry **SYSVAR**. offset **0x080**. Each page contains 4 entries which have the following meaning:

```
byte 0: entry-type and access-mode (= &00 ::= free)
      bit7-4:     entry-type
            0000 ::= LOCK
            0001 ::= CONSOLE
            0010 ::= FILE
            0100 ::= QUEUE or DUPLEX
            1000 ::= SHARED or MAP
      bit3: exclusive open (bit set when the entry is open with the
            EXCLUSIVE flag; this bit is set for the CONSOLE and
            LOCK).
      bit2-0: access-mode
            100 ::= READ allowed
            010 ::= WRITE allowed
            001 ::= EXECUTE allowed (FILE); if set, the KERNEL will
                  permit the run it.)
                     Duplex or Map bit (DUPLEX or MAP)
byte 1: link counter (= &00 ::= no link)
      (to be deleted or exclusively open, an entry should have a link
      counter equal to &00.)
byte 2,3: physical address of the entry-space
          DS owner (LOCK)
byte 4: number of pages reserved for the entry-space
byte 5,6: real-size (FILE)
          head of DS chained list (LOCK)
byte 7-15: entry-name (terminated by &00 if the name is less than 9
           characters.)
```

Executable header

Under MMPS, to be executable, a FILE entry should match some patterns and have an correct header.
The patterns to match are:

1. To be a FILE entry
2. To have the READ and EXECUTE bits set

The executable header corresponds to the 6 first bytes of the executable file with the following meaning:

```
byte 0:     executable signature &F3
byte 1,2:   real-size (copy of the bytes 5,6 of the entry)
byte 3:     number of pages to allocate for the stack and the global
            area excluding the arguments-page.
byte 4,5:   offset of the global area in the global block.
```

BAL - Block allocate

Call:

```
LD   L,number-of-pages
LDA  00
SBR  E3   ;; kernel jump
```

Action:

Allocate the number of pages stored into **L**, and return the physical address of the first byte into **BC**. The calling process becomes owner of the allocated pages which are added to its space. All its pages will be released when the process will be terminated. A page is a group of 64 bytes always aligned on 64 bytes frontier. A block is a group of N contiguous pages.

Return:

CARRY if success; **BC** contains the address of the allocated block.
No CARRY on error, error code into **A**.

Error Code:

B0 No enough page free

BCH - Block validity check

Call:

```
LD   BC,block-address
LDA  01
SBR  E3   ;; kernel jump
```

Action:

Check if block at the address stored in **BC** is valid. The block should be owned by the calling process.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1 No owner of this resource

BFR - Block free

Call:

```
LD   BC,block-address
LDA
SBR  E3   ;; kernel jump
```

Action:

Release a block allocated at the address stored in **BC**. The block should be owned by the calling process. The complete block is released.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1 No owner of this resource

BRE - Block reallocate

Call:

```
LD   BC,block-address
LD   L,number-of-pages
LDA  03
SBR  E3   ;; kernel jump
```

Action:

Reallocate (change the number of pages of) the allocated block pointed by **BC** with the new size given in **L** (the new size should be less or equal to the old one) and release the reminder.

Return:

CARRY if success.
No CARRY on error, error code into A.

Error Code:

B1	Not owner of this resource
B2	New size is greater than the old one

FSR - File System entry read

Call:

```
LD   BC,name
LD   D,fsds (&FF at first call)
LD   HL,entry-struct
LDA  04
SBR  E3   ;; kernel jump
```

Action:

Read the file system entry. At the beginning, the fsds **D** should be initialized with **&FF**. The filesystem entry information are stored into the entry structure pointed by **HL**, the name is stored in the name buffer pointed by **BC**. When no more entry are read, the error code **F1** is returned. The entry structure is described in the **FEN** kernel jump.

The output of **FSR** is the same as **FEN**.

Return:

CARRY if success; **D** contains the updated fsds for a next call.

No CARRY on error, error code into **A**.

Error Code:

F1 No such node

FEN - File system entry

Call:

```
LD   BC,name
LD   HL,entry-struct
LDA  05
SBR  E3   ;; kernel jump
```

Action:
Read the node information of a named entry in **BC** and stores them into the entry structure pointed by **HL**. The entry structure contains the following information:

```
byte 0, bits 7 to 4     Entry type
                        0000  LOCK entry
                        0001  CONSOLE entry
                        0010  FILE entry
                        0100  QUEUE or DUPLEX entry
                        1000  SHARED or MAP entry
byte 0, bits 3 to 0     Access rights
                        1000  Exclusively opened
                        0100  Read access
                        0010  Write access
                        0001  Executable access (FILE)
                              Duplex or Map bit (DUPLEX or MAP)
byte 1                  Number of links
byte 2                  Number of pages reserved
byte 3 & 4        Real length
```

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

F1 No such entry

TMP - Create a unique temporary entry name

Call:

```
LD   BC,generic-name
LDA  06
SBR  E3   ;; kernel jump
```

Action:

Create a temporary unique name using the first 3 characters of the buffer pointed by **BC** and returns the full name in this buffer, terminated by a **&00** byte.

Return:

CARRY and **BC** is filled with the temporary name.

Error Code:

None.

L0K - Create and open a LOCK entry

Call:

```
LD   BC,name
LDA  07
SBR  E3   ;; kernel jump
```

Action:

Create, if not already done, a LOCK entry named by **BC**, open it and return the descriptor in **E**. When a LOCK descriptor is closed, the LOCK entry is deleted from the file system. A LOCK is opened with the EXCLUSIVE mode. No page are needed to create a LOCK entry.

Return:

CARRY if success; **E** contains the opened descriptor.
No CARRY on error, error code into **A**.

Error Code:

F0	No entry free
F2	Entry already exists
D0	No DS free

CON - Create and open a CONSOLE entry

Call:

```
LD   BC,name
LDA  08
SBR  E3   ;; kernel jump
```

Action:

Create, if not already done, a CONSOLE entry named by **BC**, open it and return the descriptor in **E**. A CONSOLE is always opened with the modes READ, WRITE and EXCLUSIVE. When a CONSOLE is opened, it is reset. When a CONSOLE is closed, it is deleted from the file system. Creating a CONSOLE entry needs one page.

Return:

CARRY if success; **E** contains the opened descriptor.
No CARRY on error, error code into **A**.

Error Code:

B0	Not enough page free
F0	No entry free
F2	Entry already exists
D0	No DS free

FIL - Create and open a FILE entry

Call:

```
LD   BC,name
LD   E,access
LD   L,number-of-pages
LDA  09
SBR  E3   ;; kernel jump
```

Action:

Create, if not already done, a FILE entry named by **BC**, with the access rights given by **E**, reserve the number of pages given by **L** for it, open it and return the descriptor in **E**. If not enough page are available to create the FILE entry, an error is returned. The access rights are given by **E**; **E** is an OR of the following:

```
0100 := READ access
0010 := WRITE access
0001 := EXECUTABLE access
```

The real length of a FILE entry is set to 0, even the number of pages is reserved. The length will be set after each **WRI** to this FILE.

Return:

CARRY if success; **E** contains the opened descriptor.
No CARRY on error, error code into **A**.

Error Code:

B0	Not enough page free
F0	No entry free
F2	Entry already exists
D0	No DS free

QUE - Create and open a QUEUE entry

Call:

```
LD   BC,name
LD   L,number-of-pages
LDA  0A
SBR  E3   ;; kernel jump
```

Action:

Create, if not already done, a QUEUE entry name by **BC**, reserve the number of pages given in **L** for it, open it, and return the descriptor in **E**. Only the process which created the QUEUE entry has the READ access. When the QUEUE descriptor with the READ access is closed, the QUEUE entry is deleted from the file system and all the descriptors linked to it are invalidated. It is not possible to create a QUEUE with more than 4 pages. If there is not enough page available, the QUEUE entry is not created.

Return:

CARRY if success; **E** contains the opened descriptor.
No CARRY on error, error code into **A**.

Error Code:

B0	Not enough page free
F0	No entry free
F2	Entry already exists
D0	No DS free

DPX - Create and open a DUPLEX entry

Call:

```
LD   BC,name
LD   L,number-of-pages
LDA  0B
SBR  E3   ;; kernel jump
```

Action:

Create, if not already done, a DUPLEX entry name by **BC**, reserve the number of pages given in **L** for it, open it, and return the descriptor in **E**. Writing to a DUPLEX entry will give data available to the peer DUPLEX. In a same way, to be usable, the DUPLEX pair should be connected by a call to the CNT kernel-jump. It is not possible to create a DUPLEX with more than 4 pages. If there is not enough page available, the DUPLEX entry is not created.

Return:

CARRY if success; **E** contains the opened descriptor.
No CARRY on error, error code into **A**.

Error Code:

B0	Not enough page free
F0	No entry free
F2	Entry already exists
D0	No DS free

SHR - Create and open a SHARED entry

Call:

```
LD   BC,name
LD   L,number-of-pages
LDA  0C
SBR  E3   ;; kernel jump
```

Action:

Create, if not already done, a SHARED entry name by **BC**, reserve the number of pages given in **L** for it, open it, and return the descriptor in **E**. The SHARED entry has no owner. When a SHARED descriptor is close, the SHARED entry will remind in the file system until the link number is 0; at this time, it will be deleted. It is not possible to create a SHARED entry with more than 4 pages. If not enough page are available, the SHARED entry is not created and an error is returned.

Return:

CARRY if success; **E** contains the opened descriptor.
No CARRY on error, error code into **A**.

Error Code:

B0	Not enough page free
F0	No entry free
F2	Entry already exists
D0	No DS free

MAP - Create and open a MAP entry

Call:

```
LD   BC,name
LD   E,access-right
LD   HL,block-address
LDA  0D
SBR  E3   ;; kernel jump
```

Action:

Create, if not already done, a MAP entry name by **BC**, fixes the access rights given by **E** and set the base address to the block pointed by **HL** for it, open it, and return the descriptor in **E**. The access rights are given by **E**; **E** is an OR of the following:

```
0100 := READ access
0010 := WRITE access
```

The block should belong in the ownership of the calling process. If the MAP is closed, all descriptors related to this MAP are invalidated. No page are needed to create a MAP entry.

Return:

CARRY if success; **E** contains the opened descriptor.
No CARRY on error, error code into **A**.

Error Code:

- **B0** Not enough page free
- **B1** Not owner
- **F0** No entry free
- **F2** Entry already exists
- **F3** Can not map
- **D0** No DS free

DEL - Delete an entry

Call:

```
LD   BC,name
LDA  0E
SBR  E3   ;; kernel jump
```

Action:

Delete the entry named by **BC** from the file system. To be deleted, an entry should be free (no link) and the WRITE access should be allowed. The pages reserved are released, and the entry is deleted from the file system. It is not possible to delete an entry opened or linked to a running process.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

F1	No such entry
F3	Entry is busy
F4	Access denied
D1	Exclusively opened

ACS - Change the access right of an entry

Call:

```
LD   BC,name
LD   E,access
LDA  0F
SBR  E3   ;; kernel jump
```

Action:

Change the access right of the entry named by **BC** with the new access given by **E**; **E** is an OR of the following values:

```
0100 := READ access
0010 := WRITE access
0001 := EXECUTE access (FILE)
```

If this entry is opened, the descriptors will keep the old rights. The new rights will be taken at the next open (**OPN**). The access rights of the LOCK and the CONSOLES entry have no meaning, because the access rights are set while creating the entry, and these are exclusively opened. It is not possible to change the access rights if the entry is exclusively opened.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

F1	No such entry
D1	Exclusively opened

RNM - Rename an entry

Call:

```
LD   BC,name
LD   DE,new-name
LDA  10
SBR  E3   ;; kernel jump
```

Action:
Rename the entry named by **BC** with the new name given by **DE**. The rename is dynamic, all the processes linked to it will take the new name, because the process table keeps only the address of the entry. It is not possible to rename an entry with an already existing name or if the entry is exclusively opened.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

F1	No such entry
F2	Entry already exists
D1	Exclusively opened

SIZ - Re-size an FILE entry

Call:

```
LD   BC,name
LD   L,number-of-pages
LDA  11
SBR  E3,0D
```

Action:

Re-size the FILE entry named by **BC** with the new number of pages given in **L**. To be re-sized, the WRITE access should be allowed, the new size should be less than the old one, but greater than the real size and the number of link should be 0. It is not possible to re size while a descriptor to the FILE entry exists. It is not possible to re-size a FILE entry with 0 page.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

F1	No such entry
F3	Entry is busy
F4	Access denied
F5	Can not re-size

DSN - Get entry name from a opened descriptor

Call:

```
LD   BC,name
LDA  12
SBR  E3   ;; kernel jump
```

Action:

Return the entry name of the descriptor E into BC.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1 Not owner of this resource
D2 Not a valid DS

DES - Get entry type from a opened descriptor

Call:

```
LD   E,descriptor
LDA  13
SBR  E3   ;; kernel jump
```

Action:

Return the entry type (entry type and open mode) of the descriptor **E** into **H**. The following bits are used:

```
bits7-4  Descriptor type
 0000 := LOCK entry
 0001 := CONSOLE entry
 0010 := FILE entry
 0100 := QUEUE or DUPLEX (bit0 = 1) entry
 1000 := SHARED or MAP (bit0 = 1) entry
bits3-0  Open mode
 1000 := VALIDITY bit
 0100 := READ mode
 0010 := WRITE mode
 0001 := APPEND mode (FILE)
         DUPLEX (bits7-4 = 0100)
         MAP (bits7-4 = 1000)
```

If the VALIDITY bit (1000) is not set, no operation are possible on this descriptor except to close it.

Return:

CARRY if success; **H** contains the entry type and the open mode.
No CARRY on error, error code into **A**.

Error Code:

B1 Not owner of this resource
D2 Not a valid DS

OPN - Open an entry

Call:

```
LD   BC,name
LD   D,open-mode
LDA  14
SBR  E3   ;; kernel jump
```

Action:

Open the entry named by **BC** with the open mode given by **D** (only for FILE, SHARED and MAP entries, the QUEUE entries are always opened with only the WRITE mode). To be opened, an entry should have the access rights asked by the open mode and should not be already exclusively opened. The open mode is an OR of the following values:

```
1000 := EXCLUSIVE mode
0100 := READ mode
0010 := WRITE mode
0001 := APPEND mode (FILE only)
```

The descriptor is returned in **E**. Opening with the APPEND mode will sets the position at the end. Opening with only the WRITE mode will rewind the descriptor and resets the FILE. APPEND mode should be given with READ or WRITE mode. Opening with WRITE mode will set the EXCLUSIVE mode. It is not possible to open with the EXCLUSIVE mode if at least one link exists. Opening an entry will increase by 1 its number of links.

Return:

CARRY if success; **E** contains the opened descriptor.
No CARRY on error, error code into **A**.

Error Code:

F1	No such entry
F3	Entry is busy
F4	Access denied
D0	No DS free
D1	Exclusively opened

CLO - Close an opened entry

Call:

```
LD   E,descriptor
LDA  15
SBR  E3   ;; kernel jump
```

Action:

Close the entry linked to the descriptor given by **E**. If the entry is a MAP, a DUPLEX, a CONSOLE or a LOCK, the entry is deleted from the file system.

- If the entry is a QUEUE and the descriptor has the READ access, the entry is deleted, and all the other descriptors linked to this entry are invalidated; else the close work normally.
- If the entry is a DUPLEX, the connection is broken and the entry is deleted.
- If the entry is a SHARED the entry will remind till the last descriptor is closed.
- If the entry is a MAP, the block is unmapped and all the other descriptors linked to this entry are invalidated.
- If the entry is a FILE, it closes the descriptor and the entry reminds.
- If the entry is a CONSOLE and the current active CONSOLE is linked to it, the kernel will display **Closed!** and no action will be possible except to change the active CONSOLE by the SELECT key.

Closing an entry will decrease by 1 its number of links.

Return:

CARRY if success. The descriptor in **E** is invalidated and is no more usable.
No CARRY on error, error code into **A**.

Error Code:

B1 Not owner of this resource
D2 Not a valid DS

REA - Read data from a descriptor

Call:

```
LD   BC,data-buffer
LD   E,descriptor
LD   L,number-of-bytes
LDA  16
SBR  E3   ;; kernel jump
```

Action:

Read the number of bytes given by **L** from the opened descriptor **E** and store the data to the buffer pointed by **BC**.
While reading from a QUEUE or a DUPLEX entry, the number of bytes specified is discarded, and a complete message is stored. Only the process which created the QUEUE can read from it. The real number of bytes read is returned in L.
Reading from a CONSOLE will suspend the process until the number of bytes is reached or the ENTER or the DOWN-ARROW is pressed; If the UP-ARROW is pressed, the error D3 (End of file) is returned; If only 1 byte is asked from a CONSOLE descriptor, REA will return even ENTER, DOWN-ARROW or UP-ARROW are not pressed.
It is not possible to read from a LOCK nor a SHARED nor a MAP.
To read, the descriptor should be opened with the READ mode.

Return:

CARRY if success; **L** contains the number of bytes read.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
F4	Access denied
D2	Not a valid DS
D3	End of file
D4	Queue is empty

WRI - Write data to a descriptor

Call:

```
LD   BC,data-buffer
LD   E,descriptor
LD   L,number-of-bytes
LDA  17
SBR  E3   ;; kernel jump
```

Action:

Write the number of bytes given by **L** from the buffer pointed by **BC** to the descriptor given by **E**. The real number of bytes written is returned into **L**.
Writing to a CONSOLE entry will suspend the process if a **WRI** request is already pending. If a process tries to write more bytes than possible, the system will write all the bytes until the end of file, discards the remaining and returns the real number of bytes written.
When writing into a FILE descriptor, the real length is updated.
When writing to a FILE more bytes than the amount free, the system will write all the bytes until the complete FILE space is full, and return the real number of bytes written.
If a process tries to write more bytes than free into a QUEUE or a DUPLEX entry, an error is returned but nothing will be written.
It is not possible to write to a LOCK nor to a SHARED or a MAP entry.
To write, the descriptor should be opened with the WRITE mode.

Return:

CARRY if success; **L** contains the real number of bytes written.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
F4	Access denied
D2	Not a valid DS
D5	Queue is full

TEL - Get the current position of a descriptor

Call:

```
LD   E,descriptor
LDA  18
SBR  E3   ;; kernel jump
```

Action:

Return the current position of the descriptor **E** into **HL**. For a CONSOLE, a QUEUE or a DUPLEX descriptor returns always 0. It is not possible to get the position from a LOCK nor a SHARED nor a MAP descriptor.

Return:

CARRY if success; **HL** contains the current position.
No CARRY on error, error code into **A**.

Error Code:

B1 Not owner of this resource
D2 Not a valid DS

SEK - Set the current position of a descriptor

Call:

```
LD   E,descriptor
LD   HL,offset
LDA  19
SBR  E3   ;; kernel jump
```

Action:

Set the current position of the descriptor **E** to the offset given by **HL**, and return the real offset set into **HL**. The offset is always specified from the beginning of the descriptor. It is not possible to set the position after the end. SEK is without effect for the CONSOLE or QUEUE or DUPLEX descriptors.
It is not possible to set the position for a LOCK nor a SHARED nor a MAP descriptor.

Return:

CARRY if success; **HL** contains the real offset set.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid DS
D3	End of file

WAI - Wait from a descriptor

Call:

```
LD   E,descriptor
LDA  1A
SBR  E3   ;; kernel jump
```

Action:

Suspend the calling process until a message is available from the QUEUE or the DUPLEX descriptor given by **E**, and return the number of bytes present in the QUEUE or the DUPLEX into **L**. Waiting from a CONSOLE or from a FILE always return, but is not suspensive.
It is not possible to wait from a LOCK nor a SHARED nor a MAP entry.
Only the process which have the READ access can wait from it.

Return:

CARRY if success; **L** contains the number of bytes present in the QUEUE.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid descriptor

CNT - Connect a DUPLEX to a peer DUPLEX

Call:

```
LD   BC,name
LD   E,descriptor
LDA  A,1B
SBR  E3   ;; kernel jump
```

Action:

Realize a connection between the DUPLEX descriptor given in **E** and the peer DUPLEX named by **BC**. To be successful, both DUPLEX should be free from any connection.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid descriptor
D6	Can not connect

GET - Get data from a SHARED/MAP descriptor

Call:

```
LD   BC,data-buffer
LD   E,descriptor
LD   H,offset
LD   L,number-of-bytes
LDA  1C
SBR  E3   ;; kernel jump
```

Action:

Get the number of bytes **L** located at the offset **H** from the SHARED or MAP descriptor **E** and put then into the buffer pointed by BC. If the offset H + the number of bytes L are greater than the SHARED or MAP, an error is returned and no get is done.
It is not possible to do this action on a non SHARED or MAP entry.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

- **B1** Not owner of this resource
- **D2** Not a valid DS
- **D7** Out of shared

PUT - Put data to a SHARED/MAP descriptor

Call:

```
LD   BC,data-buffer
LD   E,descriptor
LD   H,offset
LD   L,number-of-bytes
LDA  1D
SBR  E3   ;; kernel jump
```

Action:

Put the number of bytes L located at the offset **H** from the buffer pointed by **BC** to the SHARED or MAP descriptor **E**. If the offset H + the number of bytes L are greater than the SHARED or MAP, an error is returned and no put is done.
It is not possible to do this action on a non SHARED or MAP entry.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid DS
D7	Out of shared

GIV - Give a LOCK

Call:

```
LD   E,descriptor
LDA  1E
SBR  E3   ;; kernel jump
```

Action:

Release the LOCK pointed by the descriptor in **E**. If a process is waiting for the LOCK (**TAK**), it will be rescheduled and will become owner of the descriptor.
It is not possible to do this action on a non LOCK entry.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid DS

TAK - Take a LOCK

Call:

```
LD   E,descriptor
LDA  1F
SBR  E3   ;; kernel jump
```

Action:

Take the LOCK pointed by the descriptor in **E**. If a process is already owner of the LOCK, the calling process will be suspended until the LOCK is released (**GIV**).
It is not possible to do this action on a non LOCK entry.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid DS

SCN - Set a CONSOLE active

Call:

```
LD   E,descriptor
LDA  20
SBR  E3   ;; kernel jump
```

Action:

Activate the CONSOLE linked to the descriptor **E**. It is possible only if the current active CONSOLE is closed, or if the current active CONSOLE is owned by the calling process, and has no request pending.
Activate a CONSOLE will start the CONSOLE-IO process on this CONSOLE.
The current CONSOLE, the process-id PID and the process name linked to are displayed by pressing the **OFF** key.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid DS

FLU - Flush a CONSOLE output

Call:

```
LD   E,descriptor
LDA  21
SBR  E3   ;; kernel jump
```

Action:

Flush the output request of the CONSOLE descriptor **E**, if there is one, and suspend the calling process like a **WRI** request were pending. This action can be used to activate the output before to end a process.
The pending message is displayed until the **RCL** is pressed to refresh the console.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid DS

RFR - Refresh a CONSOLE output

Call:

```
LD   E,descriptor
LDA  22
SBR  E3   ;; kernel jump
```

Action:

Refresh the output request of the CONSOLE descriptor **E**, if there is one, kill all pending message. This avoid the calling process to be suspended if a **WRI** is pending.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B1 Not owner of this resource
D2 Not a valid DS

EDI - Start a editable REA on a CONSOLE

Call:

```
LD   BC,buffer
LD   E,descriptor
LD   H,cursor
LD   L,number-of-bytes
LDA  23
SBR  E3   ;; kernel jump
```

Action:

Fill the input buffer of a CONSOLE descriptor given in **E** with the string in the buffer pointed by **BC**, start editing with the cursor at the position **H**, make a READ request and return the real number of bytes read in **L**.

This action looks like **REA** but the input buffer is filled before instead of to be cleaned, and the process can specify the initial position of the cursor.

The buffer is filled by the string present in the buffer, until a non-printable character is found (character < **&20**), after fill the remaining buffer with the CR character (**&0D**).

If the cursor is specified after the string end, it will be put on last character before CR.

If the cursor is after the number of bytes read, it will return at the first character read.

It is also possible to give an empty string.

The input buffer will be filled up to 26 characters.

At the return, the buffer pointed by **BC** will be filled with the characters read and the number of characters will be return into **L**.

It is not possible to use this action if a **REA** request is already running.

Return:

CARRY if success; **L** contains the number of bytes read.

No CARRY on error, error code into **A**.

Error Code:

B1	Not owner of this resource
D2	Not a valid DS

HDR - Build an executable header

Call:

```
LD   D,number-of-page
LD   E,descriptor
LD   HL,global-address-offset
LDA  24
SBR  E3   ;; kernel jump
```

Action:

Build an executable header in the FILE entry pointed by the descriptor given by **E**. The number of pages to be allocated when launching a process (**XEQ**) is given by **D** and the offset of the global address in **HL**.

The executable header as the following format:

```
byte 0:     executable signature &F3
byte 1,2:   real-size (copy of the bytes 5,6 of the entry)
byte 3:     number of pages to allocate for the stack and the
            global area excluding the arguments-page.
byte 4,5:   offset of the global area in the global block.
```

The descriptor should be opened in WRITE mode.

Return:

CARRY if success.

No CARRY on error, error code into **A**.

Error Code:

F4 Access denied

D2 Not a valid DS

GBL - Get the global address

Call:

```
LDA  25
SBR  E3   ;; kernel jump
```

Action:

Return the global address of the calling process into **BC**. The global address of a process is located on the top of the stack at the offset fixed by building the executable header (**HDR**).

BC - 2 is a pointer to the arguments page initialized by **XEQ** when starting. The global address is the same value set in **BC** when a process is started and **(BC - 2)** is the value of **HL**.

Return:

CARRY and **BC** contains the global address.

Error Code:

None

`PID` - Get the process-id

Call:

```
LDA  26
SBR  E3   ;; kernel jump
```

Action:

Return the process-id PID of the calling process into **`H`**.

Return:

CARRY and **`H`** contains the process PID.

Error Code:

None.

FID - GET the father process-id

Call:

```
LDA  27
SBR  E3   ;; kernel jump
```

Action:

Return the father process-id FID of the calling process into **H**.

Return:

CARRY and **H** contains the father process FID.

Error Code:

None.

XEQ - Execute a process

Call:

```
LD   BC,name
LD   D,input-descriptor
LD   E,output-descriptor
LD   HL,arguments
LDA  28
SBR  E3   ;; kernel jump
```

Action:

Start the process named by **BC**; pass to it the descriptors in and out given by **D** and **E** and the arguments page pointed by **HL**. The son process-id is returned to the father into **H**. The son process becomes owner of the descriptors **D** and **E**. The arguments page is copied into the son space with a limit of 63 bytes; the last is always 0. The descriptors in and out given to the son should be owned by the father and be valid. It is possible to give only descriptors to CONSOLE, FILE, DUPLEX or QUEUE. The descriptor **D** and **E** may be the same.

The son process starts with the global address into **BC**, the descriptors in and out into **D** and **E** and the arguments page pointer into **HL**. The arguments page address is copied into **(BC - 2)**.
The son process gets the same priority as its father, its event queue is removed and the event mask is set to 0.
A return address is pushed into the stack to handle the son termination.
When the top-level RET is executed, the process is terminated, all its resources are released, all its descriptors closed and the event 4 is raised to its father with the son returned value of **DE**.
When a father is returned, all its sons are still running, but their father process-id is set to **&FF**.
The executable FILE should have the access READ and EXECUTE. Executing a process will increase by 1 the number of links of the entry; when the process returns, the number of links will be decrease by 1.
It is not possible to start a process if the FILE entry is exclusively opened.
The number of pages allocated to the son while starting are set into the executable FILE header:

```
byte 0:     executable signature &F3
byte 1,2:   real-size (copy of the bytes 5,6 of the entry)
byte 3:     number of page to allocated for the stack and the
            global area excluding the arguments-page.
byte 4,5:   offset of the global area in the global block.
```

Return:

CARRY if success; **H** contains the son process-id.
No CARRY on error, error code into **A**.

Error Code:

- **B1** Not owner of this resource
- **D1** Exclusively opened
- **D2** Not a valid DS
- **F1** No such entry

F4 Access denied
C0 No process free
C2 Not a valid DS for XEQ
C3 Not a valid signature

END - Terminate a process

Call:

```
LD   H,process-id or &FF
LDA  29
SBR  E3   ;; kernel jump
```

Action:
Terminate the process pointed by **H** or itself if **&FF**. When a process is terminated, all its resources are released, all its descriptors are closed and the event 5 is raised to its father. All its sons are still running, but their father process-id FID is set to **&FF**.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

C1	No such process

PRI - Change the priority of a process

Call:

```
LD   H,process-id
LD   L,new-priority
LDA  2A
SBR  E3   ;; kernel jump
```

Action:

Change the priority of the process pointed by **H** (or itself if **&FF**) with the priority in **L**. The priority is a number from **&0** to **&F**. **&0** is the highest and **&F** is the lowest. The priority is the number of consecutive slots allocated by the scheduler to a process until 16 is reached. A priority 0 will allow 16 slots and priority 15 will allow 1 slot.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

C1 No such process

STA - Process status

Call:

```
LD   BC,process-structure
LD   H,process-id
LDA  2B
SBR  E3   ;; kernel jump
```

Action:

Return the process status of the process pointed by **H** (**&FF** for itself) into the process structure pointed by **BC**. The information are:

```
byte 0      Status bits
            80 := READY-to-RECEIVE
            40 := PAUSED
            20 := QUEUE/DUPLEX Waiting
            10 := LOCK Waiting
            08 := CONSOLE input pending
            04 := CONSOLE output pending
            02 := TRACED
            01 := KERNEL jump pending
byte 1      Priority (4 lower bits)
byte 2      Event mask
byte 3
      bits7-4     Extended status bits
                  02 := TIMER running
                  01 := DEBUG
      bits3-0     Father process-id (4 lower bits)
bytes 4-12  Process name
```

Return:

CARRY if success; **BC** is filled with the process-structure.
No CARRY on error, error code into **A**.

Error Code:

C1 No such process

PAU - Pause the calling process for a delay

Call:

```
LD   DE,delay
LDA  2C
SBR  E3   ;; kernel jump
```

Action:

Pause for the delay **DE** the calling process. The process is suspended until the delay is elapsed. The delay is in scheduler slots, and its decremented each time the scheduler is running. When the delay reaches **&FFFF**, the process is restarted.

Return:

CARRY.

Error Code:

None.

EVT - Create the event queue for asynchronous events

Call:

```
LD   L,enabled-event-mask
LDA  2D
SBR  E3   ;; kernel jump
```

Action:

Create the event queue for the calling process to receive the asynchronous events and set the event mask to **L**. The mask is an OR of the events from 0 to 7. It is not possible to create an event queue if one is already created.

When an asynchronous event is raised to the process, and if the event is enabled, the event is queued and register the raising process-id, the event number, and the eventual value DE from the raising process (events 0-4).

The events 4 to 7 are reserved to the kernel:

```
4 := Son returned to the scheduler with the value DE
5 := Son terminated
6 := TIME is reached
7 := BREAK
```

The events 0 to 3 are available for user purpose. The value **DE** is meaningful.

Return:

CARRY if success.

No CARRY on error, error code into **A**.

Error Code:

C4 Event queue already created

KIL - Destroy the events queue

Call:

```
LDA  2E
SBR  E3   ;; kernel jump
```

Action:

Destroy the events queue of the calling process, and reset the event-enable-mask to **&00**. All pending events are lost.

Return:

CARRY.

Error Code:

None.

MSK - Set the events mask

Call:

```
LD   H,disabled-event-mask
LD   L,enabled-event-mask
LDA  2F
SBR  E3   ;; kernel jump
```

Action:

Change the events mask by clearing the events given by **H** and setting the events given by **L**. If **H** and **L** are **&00**, the mask is unchanged and is useful to get the current event mask in **L**.
This action has no effect if the events queue is not created.

Return:

CARRY and **L** contains the current event mask.

Error Code:

None.

POL - Poll the event queue and get the oldest event

Call:

```
LDA  30
SBR  E3   ;; kernel jump
```

Action:

Poll the events queue and if an event is pending, return the raising process-id into **H**, the event number into **L**, and the sent value of **DE** into **DE** for the event 0 to 4. If no event is pending an error is returned.

Return:

CARRY if success; **H** contains the process-id, **L** the event number and **DE** the sent value for events 0 to 4.
No CARRY on error, error code into **A**.

Error Code:

C4 No event queue
D4 Queue empty

RAI - Raise an event to process

Call:

```
LD   DE,value
LD   H,process-id
LD   L,event-number
LDA  31
SBR  E3   ;; kernel jump
```

Action:
Raise the event number in **L** with the value from **DE** to the process pointed by **H**. Only the events from 0 to 3 may be raised by a process. This action has no effect if the receiver process has no events queue.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

C1 No such process
C5 No such event number

TIM - Initialized the TIME event

Call:

```
LD   DE,delay
LDA  32
SBR  E3   ;; kernel jump
```

Action:

Initialize a TIME event with the delay in **DE**. The scheduler will decrement the TIME delay and when **&FFFF** is reached, it will push the event 6 to the process. Only one TIME is allowed by process and it is not possible to initialize a new one until the first is not reached. This action is without effect if the calling process does not handled the event 6.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

C6 TIME is running

ABT - Abort the TIME delay

Call:

```
LDA  33
SBR  E3   ;; kernel jump
```

Action:

Abort the TIME delay if one is running.

Return:

CARRY.

Error Code:

None.

SEN - Send a message to a process

Call:

```
LD   BC,message-address
LD   H,process-id
LD   L,length
LDA  34
SBR  E3   ;; kernel jump
```

Action:

Send the message pointed by **BC** with the length **L** to the process pointed by **H**. The destination process should be in the state READY-TO-RECEIVE. The message will be copied into the destination process space after the needed number of pages is allocated. If not enough pages are available to send the message, an error is returned.
The destination process will restart when a message is received with the address of the message in **BC**, its length in **L** and the sender process-id in **H**. The receiver process need to free the block pointed by **BC** after the message has been treated.
It is not possible to send a message to itself.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

B0	No enough page free
C1	No such process
C7	Process not READY-TO-RECEIVE

SNR - Send and receive a message

Call:

```
LD   BC,message-address
LD   H,process-id
LD   L,length
LDA  35
SBR  E3   ;; kernel jump
```

Action:

Like **SEN**, send a message to another process, and wait for a reception of a message. When the message is sent, the message block is released and the process goes into the state READY-TO-RECEIVE. Like **REC**, the calling process is restarted when a message is received, and **BC** will contain the address of the message, **L** its length and **H** the sender process-id.

The receiver process need to free the block pointed by **BC** after the message has been treated.

It is not possible to send a message to itself.

Return:

CARRY if success; **BC** contains the message address, **L** its length and **H** the sender process-id.

No CARRY on error, error code into A.

Error Code:

B0	No enough page free
C1	No such process
C7	Process not READY-TO-RECEIVE

REC - Receive a message from a process

Call:

```
LDA  36
SBR  E3   ;; kernel jump
```

Action:

The calling process is suspended until a message is received; when the process execution restarts, **BC** will contain the address of the message, **L** its length and **H** the sender process-id.

The message is copied into the receiver space after the pages are allocated. If there is not enough page available to send the message, the receiver process will be not restarted.

The receiver process need to free the block pointed by **BC** after the message has been treated.

Return:

CARRY and **BC** contains the message address, **L** its length and **H** the sender process-id.

Error Code:

None.

DBG - Execute a process under debug

Call:

```
LD   BC,name
LD   D,input-descriptor
LD   E,output-descriptor
LD   HL,arguments
LDA  37
SBR  E3   ;; kernel jump
```

Action:

Start the process named by **BC**; pass to it the descriptors in and out given by **D** and **E** and the arguments page pointed by **HL**. The son process-id is returned to the father into **H**. The son process becomes owner of the descriptors **D** and **E**. The arguments page is copied into the son space with a limit of 63 bytes; the last is always 0. The descriptors in and out given to the son should be owned by the father and be valid. It is possible to give only descriptors to CONSOLE, FILE, DUPLEX or QUEUE. The descriptor **D** and **E** may be the same.

Under debug, the son process is stopped. It is fully controlled by its father, through the kernel jumps **TRA**, **REG**, **COD** and **DAT**.

Like **XEQ**, the son process starts with the global address into **BC**, the descriptors in and out into **D** and **E** and the arguments page pointer into **HL**. The arguments page address is copied into **(BC - 2)**.
The son process gets the same priority as its father, its event queue is removed and the event mask is set to 0.
A return address is pushed into the stack to handle the son termination.
When the top-level RET is executed, the process is terminated, all its resources are released, all its descriptors closed and the event 4 is raised to its father with the son returned value of **DE**.
If the father process is ended, the debugged process is also ended.
The executable FILE should have the access READ and EXECUTE. Executing a process will increase by 1 the number of links of the entry; when the process returns, the number of links will be decrease by 1.
It is not possible to start a process if the FILE entry is exclusively opened.
The number of pages allocated to the son while starting are set into the executable FILE header:

```
byte 0:     executable signature &F3
byte 1,2:   real-size (copy of the bytes 5,6 of the entry)
byte 3:     number of page to allocated for the stack and the
            global area excluding the arguments-page.
byte 4,5:   offset of the global area in the global block.
```

Return:

CARRY if success; **H** contains the son process-id.
No CARRY on error, error code into **A**.

Error Code:

B1 Not owner of this resource
D1 Exclusively opened

D2	Not a valid DS
F1	No such entry
F4	Access denied
C0	No process free
C2	Not a valid DS for XEQ
C3	Not a valid signature

TRA - Trace the next instruction of a debugged process

Call:

```
LD   H,process-id
LDA  38
SBR  E3   ;; kernel jump
```

Action:

Trace the next instruction of the process pointed by **H**. This process should have been launched by **DBG**. The calling process is suspended until the debugged process has executed its instruction.

Return:

CARRY if success.
No CARRY on error, error code into **A**.

Error Code:

C1 No such process
E0 Process not debugged

REG - Get registers of a debugged process

Call:

```
LD   BC,register-structure
LD   H,process-id
LDA  39
SBR  E3   ;; kernel jump
```

Action:

Get the registers into **BC** of the process pointed by **H**. This process should have been launched by **DBG**. The register-structure is:

```
byte 0-1  := SP
byte 2-3  := HL
byte 4-5  := DE
byte 6-7  := BC
byte 8    := A
byte 9-10 := PC
byte 11   := F
```

Return:

CARRY if success; **BC** is filled with the registers.
No CARRY on error, error code into **A**.

Error Code:

C1	No such process
E0	Process not debugged

COD - Get code from a debugged process

Call:

```
LD   BC,code-buffer
LD   H,process-id
LD   L,number-of-bytes
LDA  3A
SBR  E3   ;; kernel jump
```

Action:

Get the number of bytes given by **L** to the code buffer **BC** of the process pointed by **H**, starting at the current **PC** address of the debugged process. This process should have been launched by **DBG**.

Return:

CARRY if success; **BC** is filled with the code.
No CARRY on error, error code into **A**.

Error Code:

C1	No such process
E0	Process not debugged

DAT - Get data from a debugged process

Call:

```
LD   BC,data-buffer
LD   DE,data-address
LD   H,process-id
LD   L,number-of-bytes
LDA  3B
SBR  E3   ;; kernel jump
```

Action:

Get the number of bytes given by **L** to the data buffer **BC** of the process pointed by **H**, starting at the address specified by **DE**. The data space, from **DE** to **(DE + L)** should be located in the same block and this block should be owned by the debugged process. This process should have been launched by **DBG**.

Return:

CARRY if success; **BC** is filled with the data.
No CARRY on error, error code into **A**.

Error Code:

C1	No such process
E0	Process not debugged
E1	Data outside process space

RCH - Reschedule

Call:

```
LDA  3C
SBR  E3   ;; kernel jump
```

Action:

Reschedule the process activity; Suspend the calling process and start the next process ready. If no other process may be started, the calling process is restarted.
When the kernel-jump return, the process is rescheduled.

Return:

CARRY.

Error Code:

None.

DWN - Down the system and return to BASIC

Call:

```
LDA  3D
SBR  E3   ;; kernel jump
```

Action:

Down the system: end all the running processes without warning the fathers, close all the descriptors, release all the resources and return to BASIC. This is the only way to exit properly from the system. No privilege are asked to execute it. The system is always calling **DWN** before starting. This kernel jump should never return.

Return:

DWN never returns.

Error Code:

None.

Programs examples

Here after an example of a funny program, called **RUNPRO**. This executable will blink the **RUN** or the **PRO** flags on the LCD.

The program first tries to create a lock (**RUN.lock**); If it is successful, the **RUN** flag is set or clear. Else, it tries to create a second lock (**PRO.lock**); If it is successful, the **PRO** flag is set or clear; Else the executable returns with the error code **&F2**.

The clear/set are delayed, and the both **PRO** flag is managed 2 times faster than the **RUN** flag.

```
      .IF   INCLUDED?
      ;; Align on next 64-bytes frontier
      .ALIGN:     40
      .ELSE
      ;; Origin does not matter, because code
      ;; and data are fully relocatable.
      .ORIGIN: 40C5
      .ENDIF

      .CODE

      ;; RUNPRO. A funny executable blinking the
      ;; RUN or the PRO flags on the LCD.

      ;; The program first tries to create a lock
      ;; (RUN.lock),;If it is successful, the RUN
      ;; flag is set or clear. Else, it tries to
      ;; create a second lock (PRO.lock); If it
      ;; is successful, the PRO flag is set or
      ;; clear; Else the executable returns with
      ;; the error code &F2.

      ;; The clear/set are delayed, and so, the
      ;; PRO flag is managed 2 times faster than
      ;; the RUN flag.

      ;; MMPS kernel-jump and macros
      ;;
      .INCLUDE:   MMPS.inc

RUNPRO:
      .LOCAL

      .BYTE
      BLDHEADER   RUNPRO.asm$$._length 01 0000

      .CODE
      ;; RUN bit in LOWLCDFLAG
RUNbit      .EQU  &40
      ;; PRO bit in LOWLCDFLAG
PRObit      .EQU  &20

      ;; The process enters here
      ;;   BC contains the global address
      ;;   DE the in and out descriptors
      ;;   HL points to the arguments
```

```
      PUSH  DE
      LD    H,RUNbit
      LD    BC,PC
      LDA   *_*offRUN
trylock:
      ADD   BC
      ;; try to create a lock (LOK) named by BC
      MMPS  krn.LOK
      STA   E
      JR    C,loop
      ;; error! So test if PRObit. If yes exit with error code
      LDA   H
      CPA   PRObit
      JR    Z,outerr
      ;; shift right and try again with next
      RCF
      SR
      STA   H
      LDA   0A
      JR    trylock
loop:
      LDA   H
      SL
      ;; set the delay value in DE (PRO x 2 faster the RUN)
      STA   E
      LD    D,00
      LD    B,<LOWLCDFLAG
      LD    C,>LOWLCDFLAG
forever:
      LDA   H
      XOR   FF
      STA   H
      ;; critical section - sure to be proof of interrupt
      DI
      CP    H,80
      JR    C,clearbit
      ;; set the flag bit
      LDA   H
      OR    (BC)
      JR    loadbit
clearbit:
      ;; clear the flag bit
      LDA   (BC)
      STA   L
      LDA   H
      STA   (BC)
      LDA   L
      AND   (BC)
loadbit:
      STA   (BC)
      ;; critical section - restore the interrupts
      EI
      ;; delay the process for the value DE
      MMPS  krn.PAU
      JR    C,forever
      STA   E
outerr:
      LD    D,00
out:
```

```
        POP   BC
        ;; return to father with code in DE
        RET

        .TEXT
offRUN:
        ;; lock name for RUN flag
        "RUN.lock\00\00"
        ;; lock name for PRO flag
        "PRO.lock\00\00"

        .END
```

The next example **SHOWKEY** is a small program reading one key from the input console and displaying the key code in hexadecimal.

```
      .IF   INCLUDED?
      ;; Align on next 64-bytes frontier
      .ALIGN:     40
      .ELSE
      ;; Origin does not matter, because code
      ;; and data are fully relocatable.
      .ORIGIN: 40C5
      .ENDIF

      .CODE

      ;; SHOWKEY - Get a key from a console and print
      ;; the ASCII code to the same console
      ;;

      ;; MMPS kernel-jump and macros
      ;;
      .INCLUDE:   MMPS.inc

SHOWKEY:
      .LOCAL

      ;; MMPS executable header. Perhaps a macro, no?
      ;
      .BYTE
      BLDHEADER   SHOWKEY.asm$$._length 01 0000

      ;;
      .CODE
      ;; The process enters here
      ;;   BC = global address
      ;;   D in and E out descriptor
      ;;   HL = argument address
      LD    BC,HL

      ;; Get the descriptor type and access into H
      MMPS  krn.DES
      JR    NC,outerr
      LDA   H

      ;; We expect a console exclusively
      BIT   10
      JR    Z,outerrcon

forever:
      ;; Get on key from the console. When key is
      ;; pressed, we return immediately
      LD    L,01
      MMPS  krn.REA

      ;; Error on read ? We just exit
      JR    NC,outerr

      LDA   (BC)
      PUSH  A
      CALL  highTOHEX
```

```
      STI   (BC)
      POP   A
      CALL  lowTOHEX
      STD   (BC)

      ;; Refresh the console. This will avoid
      ;; to be susped if a message is pending.
      MMPS  krn.RFR

      ;; Write the ASCII code to console
      LD    L,02
      MMPS  krn.WRI
      JR    forever

outerrcon:
      ;; Return No such descriptor
      LDA   err.NOD
outerr:
      ;; Error. Return error code from A into
      ;; E and set D to 0.
      LD    D,00
      STA   E
      RET

      .END
```

The example **BDGRD** described here is a mechanism of client/server using an asynchronous queue (QUE). The first instance of the executable **BDGRD** will create a queue (**QUE**) named **qBDGRD** and inherit the read access; it is the server. The second instance (and all others) will not be able the create the queue and will just reach to **OPN** it; it is the client. The client wait for a message from its console, and send it to the server, the server will print it on his console.

The example has no interest except to learn how to work with MMPS. Instead of to print the messages, it is possible to save them in a file, etc...
This principle is more or less what is done when you send a document to a printer spooler, or like the **syslogd** daemon on the UNIX systems.

```
      .IF   INCLUDED?
      ;; Align on next 64-bytes frontier
      .ALIGN:     40
      .ELSE
      ;; Origin does not matter, because code
      ;; and data are fully relocatable.
      .ORIGIN: 40C5
      .ENDIF

      .CODE

      ;; BDGRD - Server/client Queue tests for MMPS
      ;; this test will test a mechanism of the
      ;; asynchronous queues. These queues are
      ;; anonymous.

      ;; The first instance of BDGRD will create the
      ;; queue qBDGRD. It has the read access and
      ;; becomes the server. Only one server may
      ;; exists, because MMPS will fails to create
      ;; another queue of the same name. All messages
      ;; read from the queue will be written to the
      ;; out descriptor D.

      ;; All the following instances will try to
      ;; open the queue qBDGRD. They will have the
      ;; write access, and become client.
      ;; Client will read from its in descriptor
      ;; and write to the server through the queue.

      ;; MMPS kernel-jump and macros
      ;;
      .INCLUDE:   MMPS.inc

BDGRD:
      .LOCAL

      ;; MMPS executable header.
      ;;
      .BYTE
      BLDHEADER   BDGRD.asm$$._length 01 0000

      ;;
      .CODE
      ;; The process enters here
      ;;   BC = global address
      ;;   D in and E out descriptor
```

```
      ;;   HL = argument address

      ;; Expected to be server, so put 's' in H
      ;;
      LD    H,$s

      ;; Server queue name in BC
      ;;
      LD    BC,PC
      LDA   *_*qName
      ADD   BC
      LD    L,01
      ;; Create a queue
      ;;
      MMPS  krn.QUE
      JR    C,doalloc

      ;; Unable to create a queue. Not server
      ;; So client just tres to open it
      ;;
      LD    E,04
      MMPS  krn.OPN
      JR    NC,outerr

      ;; Unable to create a queue. Not server
      ;; So client just tries to open it
      ;;
      LD    E,04
      MMPS  krn.OPN
      JR    NC,outerr

      ;; We are client, so put 'c' in H
      ;;
      LD    H,$c

      ;; If we open the queue, we read from D
      ;; and write to the queue. So exchange
      ;; If we open the queue, we read from D
      ;; and write to the queue. So exchange
      ;; the D and E descriptors
      ;;
      LDA   D
      STA   L
      LDA   E
      STA   D
      LDA   L
      STA   E

doalloc:
      ;; Ok. So we need a page (64 bytes) as
      ;; a buffer to receive and send messages.
      ;; We will allocate it.
      ;; Request 1 page in L and the address
      ;; of the block is returned into BC
      ;;
      LD    L,01
      MMPS  krn.BAL
      JR    NC,outerr
```

```
forever:
      ;; Main loop: wait from descriptor E
      ;; MMPS will suspend for wait only if E is
      ;; a queue else we return with CARRY set
      ;;
      MMPS  krn.WAI
      JR    NC,outerr

      ;; Get our PID to build PID to hexa digit
      ;; and add 's' or 'c'
      ;;
      PUSH  HL
      MMPS  krn.PID
      LDA   H
      RCF
      ADC   $0
      STI   (BC)
      POP   HL
      LDA   H
      STI   (BC)

      ;; Because in header we request 1 page
      ;; for global area, so 1 page = 64 bytes
      ;; Prepare to read 63 bytes max from
      ;; descriptor R
      LD    L,3D
      MMPS  krn.REA
      JR    NC,outerr

      PUSH  DE
      ;; Now we exchange D and E to write on
      ;; the out descriptor
      ;;
      LDA   D
      STA   E

      ;; Refresh the output. Nice if E is a
      ;; console...
      MMPS  krn.RFR

      ;; Write the received buffer, L contains
      ;; the number of bytes really read but we
      ;; have to add 2 bytes and  decrement BC
      ;; to point to the PID} header
      ;;
      DEC   BC
      DEC   BC
      INC   L
      INC   L
      MMPS  krn.WRI

      ;; Restore BC for next message to be read
      ;;
      INC   BC
      INC   BC

      ;; Back to forever loop waiting the next
      ;; incoming message
      POP   DE
```

```
      JR    C,forever

outerr:
      ;; Error. Return error code from A into
      ;; E and set D to 0.
      LD    D,00
      STA   E
      RET

      .TEXT
      ;; Name of the server queue: qBDGRD
qName:
      "qBDGR\00"

      .END
```

The program below is **STARTER**. This is the first executable started by the kernel. It has in charge to wait an entry on the console, assume that is a executable name, so try to launch it (**XEQ**), finally receive and display the asynchronous events from the kernel, like a son process termination.

```
      .IF   INCLUDED?
      ;; Align on next 64-bytes frontier
      .ALIGN:     40
      .ELSE
      ;; Origin does not matter, because code
      ;; and data are fully relocatable.
      .ORIGIN: 40C5
      .ENDIF

      .CODE

      ;; STARTER - The first executable launched
      ;; by the MMPS kernel boot.

      ;; It creates a event queue to receive the
      ;; events 4, 5 and 7 from the kernel. It
      ;; polls the events queue and display the
      ;; catched both event 4 (normal son process
      ;; termination with the return code) and
      ;; event 5 (ended son process).

      ;; It reads from the console, assumes that
      ;; an executable name is given followed by
      ;; some arguments, opens a new console and
      ;; activates it, and tries to lauch the
      ;; process on this console.
      ;; If this fails, it displays the error
      ;; code.

      ;; Note that the RCL key should be pressed
      ;; to refresh the console output.

      ;; If just ENTER is read, it just polls
      ;; the events queue.

      ;; If End-Of-File (UP-Arrow) is received
      ;; and the STARTER PID is 0, a kernel
      ;; shutdown (DWN) is performed.

      ;; MMPS kernel-jump and macros
      ;;
      .INCLUDE:   MMPS.inc

STARTER:
      .LOCAL

      .BYTE

      BLDHEADER   STARTER.asm$$._length 01 0040

      .CODE

      ;; program enters here
      ;;   BC contains the global address
```

```
      ;;   DE the in and out descriptors
      ;;   HL points to the arguments
      PUSH  DE
      LD    BC,PC
      LDA   *_*starterCON
      ADD   BC
      LD    DE,BC
      LD    BC,HL

      ;; set event mask to 7, 5 and 4
      ;; and create the event queue
      LD    L,B0
      MMPS  krn.EVT
      LD    L,02
copyseed:
      LDI   (DE)
      STI   (BC)
      DJC   copyseed

      AND   (BC),00
      LDA   0D
      ADD   BC
      POP   DE

checkevent:
      PUSH  DE

checkeventloop:
      ;; check for any events in queue
      MMPS  krn.POL
      JR    NC,noevent

      ;; event is present: due to mask, we expect
      ;;   7 := BREAK key  -> do nothing
      ;;   5 := son killed -> display '>p -5'
      ;;   4 := son exited -> display '>p +4:ddee'
      LDA   *_*evt4STR[evtPChere]
      CP    L,04
      JR    Z,doevent
      CP    L,05
      JR    NZ,checkeventloop
      LDA   *_*evt5STR[evtPChere]

doevent:
      PUSH  BC
      PUSH  DE
      PUSH  HL
      LD    DE,BC
      LD    BC,SP
      INC   BC
      LD    HL,BC
      LD    BC,PC
evtPChere:
      ADD   BC
      ;; Here call the FMTSRING from the run
      ;; time with in
      ;;    BC the format string
      ;;    DE the formatted string
      ;;    HL the structure to format
```

```
        CALL  FMTSTRING

        SCF
        LDA   E
        POP   BC
        POP   BC
        POP   BC
        SBC   C
        STA   L
        POP   DE

        ;; write events to out descriptor E
        MMPS  krn.WRI
        JR    checkevent

noevent:
        POP   DE

        ;; read 27 characters from in descriptor E
        ;; we read more than 26 characters to force
        ;; the user to press the ENTER key
        LD    L,#27
        MMPS  krn.REA
        ;; error from REA? perhaps EOF
        JR    NC,checkforout

        ;; nothing read
        CP    L,00
        JR    Z,checkevent
        PUSH  DE
        PUSH  BC
        LDA   C
        SCF
        SBC   10
        STA   C
        ;; create a TMP name from seed CONp.xxxx
        MMPS  krn.TMP

        ;; open a new console
        MMPS  krn.CON
        POP   HL
        JR    NC,confailed

        CALL  NEXTARG
        JR    Z,closecon

        ;; select the console as active
        MMPS  krn.SCN
        LDA   E
        STA   D
        ;; launch the son !
        ;;   BC the name of the son
        ;;   D and E the descriptor to the console
        ;;   HL the remaining assumed as arguments
        MMPS  krn.XEQ

        ;; oops ! launch failed
        JR    NC,closecon
```

```
      ;; Display >P with P the PID of the son
      LDA   3E
      STI   (BC)
      LDA   H
      OR    30
      STD   (BC)
      POP   DE

      ;; refresh the console. If a WRI is pending
      ;; process will suspend
      MMPS  krn.RFR
      ;;

      ;; display >p with p = son pid's
      LD    L,02
      MMPS  krn.WRI
      JR    checkevent

closecon:
      STA   H
      ;; Mmmm... launch failed. Close the console
      MMPS  krn.CLO

showerr:
      ;; Show !ee
      POP   DE

      ;; restore our console
      MMPS  krn.SCN
      LDA   21
      STI   (BC)
      LDA   H
      CALL  highTOHEX
      STI   (BC)
      LDA   H
      CALL  lowTOHEX
      STD   (BC)
      DEC   BC

      ;; write !ee with ee the error code from XEQ
      LD    L,03
      MMPS  krn.WRI
      JR    checkevent

confailed:
      LD    BC,HL
      STA   H
      JR    showerr

checkforout:
      ;; error code is D3 ? End-of-file
      CPA   err.EOF
      JR    NZ,fatal

      ;; who am i ? is my PID 0 ?
      MMPS  krn.PID
      CP    H,00
      JR    NZ,out
```

```
      ;; Yes ! User ask me to DWN the kernel
      MMPS  krn.DWN
      ;; DWN should never return
      ;; here we enter into the FATAL hook of MMPS
fatal:
      SBR   (93)

out:
      ;; STARTER exits normally with &0000
      CLA
      STA   D
      STA   E
      RET

      SBC   C
      SBC   C
      SBC   C
      SBC   C

      .TEXT
evt4STR:
      ;; string for '>p +v:eeee'
      ">.L +.L:.W\00"

evt5STR:
      ;; string for '>p -v'
      ">.L -.L\00"

starterCON:
      ;; seed for console name
      "CON\00"

      .END
```

In this example, the FATAL hook of MMPS is presented. When called by a **SBR (&93)**, the registers will be saved into the BASIC variable **E$** in binary as follow:

FF AA BB CC DD EE HH LL PCPC SPSP

and the RESET vector SBR (&FE) is called. This is useful to debug an unexpected situation (like a return from **DWN**).

The **DEBUG** executable. **DEBUG** launches a process (**DBG**) with its arguments and trace it (**TRA**). It gets the CPU registers values (**REG**), displays them and at each key pressed, execute the next instruction (**TRA**) of the debugged process.

```
        .IF   INCLUDED?
        ;; Align on next 64-bytes frontier
        .ALIGN:     40
        .ELSE
        ;; Origin does not matter, because code
        ;; and data are fully relocatable.
        .ORIGIN: 40C5
        .ENDIF

        .CODE

        ;; DEBUG - A process debugger for MMPS
        ;; This process has in charge to launch under
        ;; debugger (DBG) the process named in argument.
        ;; The remaining arguments are passed to the
        ;; debugged son.

        ;; It displays the traced process CPU registers
        ;; and when a key is pressed, trace the next
        ;; instruction and updates the CPU registers
        ;; display.

        ;; DEBUG creates and activates its own console
        ;; for its usage, so it keeps unchanged the
        ;; original IN and OUT descriptors set by the
        ;; father process for the debugged process.

        ;; MMPS kernel-jump and macros
        ;;
        .INCLUDE:   MMPS.inc

DEBUG:
        .LOCAL

        ;; MMPS executable header.
        ;;
        .BYTE
        BLDHEADER   DEBUG.asm$$._length 01 0040

        .CODE
        ;; The process enters here
        ;;   BC contains the global address
        ;;   DE the in and out descriptors
        ;;   HL points to the arguments
        ;;
        PUSH  DE

        ;; BC points to the seed for our console
        ;; and we copy the name to the top of our
        ;; stack.
        LD    BC,PC
        LDA   *_*debugCON
        ADD   BC
        LD    DE,BC
        LD    BC,SP
```

```
      LDA   C
      AND   C0
      STA   C
      PUSH  HL
      PUSH  BC
      LD    L,03
copyseed:
      LDI   (DE)
      STI   (BC)
      DJC   copyseed
      POP   BC
      POP   HL

      ;; Create a TMP name from seed DEBp.xxxx
      MMPS  krn.TMP

      ;; Open a console for DEBUG. Like that
      ;; the in and out descriptor coming
      ;; from our father will be passed to
      ;; the debugged process. Nice, no ?
      MMPS  krn.CON

      STA   D
      JR    NC,popouterr

      ;; Activate  our console
      MMPS  krn.SCN

      ;; We expect to have the name of the
      ;; process to debug in the argument page.
      ;; Fetch the next word, and this will
      ;; start the argument for the debugged
      ;; process we launch. We restore also
      ;; the original in and out descriptor
      ;; receive from our father
      CALL  NEXTARG
      LDA   E
      POP   DE
      PUSH  A
      MMPS  krn.DBG

      STA   D
      POP   A
      ;; Error on debug. Save the error code
      ;; to A and retrieve the out descriptor
      JR    NC,outerr

      STA   E
      AND   (BC),00
      LDI   (BC)

forever:
      LDA   H
      PUSH  A
      ;; Get the CPU register of the debugged
      ;; process
      MMPS  krn.REG

      PUSH  DE
```

```
      LD    HL,BC
      LDA   20
      ADD   BC
      LD    DE,BC
      PUSH  DE

      ;; Here call the FMTSRING from the run
      ;; time with in
      ;;    BC the format string
      ;;    DE the formatted string
      ;;    HL the structure to format
      LD    BC,PC
      LDA   *_*debugREGstr
      ADD   BC
      CALL  FMTSTRING

      ;; The formatted string is 26 characters:
      ;;    SSSS HHLLDDEEBBCCAA PCPCFF
      LD    L,#26
      POP   BC
      POP   DE

      ;; Refresh the console. This will avoid
      ;; to be susped if a message is pending.
      MMPS  krn.RFR

      ;; Write the formatted string of the CPU
      ;; registers to the out descriptor
      MMPS  krn.WRI

      LDA   C
      AND   C0
      STA   C

      ;; Wait for one key pressed
      LD    L,01
      MMPS  krn.REA
      POP   A

      ;; Trace the debugged process. We will
      ;; suspend until the process has executed
      ;; its next instruction
      STA   H
      MMPS  krn.TRA
      JR    C,forever

      ;; Error when tracing. We exit...
out:
      ;; Return to father with &0000
      CLA
      STA   D
      STA   E
      RET

popouterr:
      POP   BC
outerr:
      ;; Error code to E and 0 to D
      LDA   D
```

```
      STA   E
      LD    D,00
      RET

      .TEXT
debugREGstr:
      ".W .W.W.W.B .W.B\00"

      .TEXT
      ;; Seed for the DEBUGger console name
debugCON:
      "DEB\00"

      .END
```

When running a process under debugger, the son process termination does not wake-up the debugger leader. Use the **ON/BREAK** key in the debugger console to kill **DEBUG**.

MAKER is an executable used to copy a program code from the BASIC area into the MMPS file-system. It also fill the executable header correctly (**HDR**) and fixes the access right for execution (**fs.REA|fs.XEQ**). The name of the executable created is given as argument to **MAKER**. The code is loaded into the PC-1500/A BASIC memory as a BASIC program (do not try to edit it).

```
        .IF   INCLUDED?
        ;; Align on next 64-bytes frontier
        .ALIGN:     40
        .ELSE
        ;; Origin does not matter, because code
        ;; and data are fully relocatable.
        .ORIGIN: 40C5
        .ENDIF

        .CODE

        ;; MAKER - Build a executable file from the
        ;; BASIC area. The foreign assembled executable
        ;; should be loaded into the BASIC area with
        ;; a CLOAD
        ;; -**- DO NOT TRY TO EDIT THIS PROGRAM -**-

        ;; MAKER will create a file with the length
        ;; retrieved from the BASIC area, copy the
        ;; code and build the MMPS header.
        ;; The name of the executable is passed in
        ;; argument to MAKER

        ;; MMPS kernel-jump and macros
        ;;
        .INCLUDE:   MMPS.inc

MAKER:
        .LOCAL

        ;; MMPS executable header.
        ;
        .BYTE
        BLDHEADER   MAKER.asm$$._length 01 0040

        ;;
        .CODE
        ;; The process enters here
        ;;   BC = global address
        ;;   D in and E out descriptor
        ;;   HL = argument address
        PUSH  DE

        ;; First we create our lock to avoid another
        ;; MAKER process to run on the same executable
        LD    BC,PC
        LDA   *_*makerLOK
        ADD   BC
        MMPS  krn.LOK
        POP   DE
        JR    NC,outerrnodel
```

```
      PUSH  DE
      ;; Compute the size of the executable to install.
      ;; This is ENDBASE - STARTBASIC.
      LDS   (>STARTBASIC)
      LD    DE,BC
      LDS   (>ENDBASIC)
      SCF
      LDA   C
      SBC   E
      STA   C
      LDA   B
      SBC   D
      STA   B
      PUSH  BC

      ;; Under MMPS, the space for a file should be
      ;; at the creation. The space to allocate is in
      ;; number of pages (a page = 64 bytes). So we
      ;; first add 63 bytes and divide the sum by 64
      LDA   3F
      ADD   BC
      LDA   C
      AEX
      AND   0C
      SR
      SR
      STA   C
      LDA   B
      AND   3F
      SL
      SL
      RCF
      ADC   C
      JR    NZ,l3

      ;; The number of page is 0. The BASIC is empty.
      POP   MN
      JR    popout

l3:
      ;; Copy the name of the executable to create
      ;; for the argument line and create the file.
      ;; We use the access RWX (07)
      LD    BC,HL
      LD    E,07
      STA   L
      MMPS  krn.FIL
      JR    NC,pop2outerr

      ;; For the copy, MAKER access directly the
      ;; BASIC area. Nice, we set the source address
      ;; and require 64 bytes (or less) to write
      ;; into the file
      LDS   (>STARTBASIC)
copyfrom:
      POP   HL
      PUSH  BC
      PUSH  DE
      LDA   H
```

```
        JR    NZ,l1
        ;; If H = 0 and L < 64, so this is the last
        ;; page, and we copy just the required amount
        ;; of bytes
        LDA   L
        CPA   40
        JR    NC,l2
l1:
        ;; We substract 64 bytes to the size
        SCF
        LDA   L
        SBC   40
        STA   L
        LDA   H
        SBC   N
        STA   H
        LDA   40
l2:
        POP   DE
        POP   BC
        PUSH  HL
        STA   L

        ;; Load the amount of byte to be written in L
        ;; Duplicate it in H to check if all is copied
        STA   H
        MMPS  krn.WRI
        JR    NC,pop2outerr

        ;; Check if all bytes where  written
        LDA   L
        CPA   H
        JR    NZ,pop2outeof

        ;; Next page to be copied
        ADD   BC

        ;; It is finished ?
        CPA   40
        JR    C,copyfrom

        POP   MN

        ;; Now we prepare to call the build of the
        ;; executable header (HDR). The user code
        ;; contains the number of pages to allocate
        ;; when launching the program (XEQ) and the
        ;; offset of the global address. This is the
        ;; amount of memory reserved for the stack
        ;; and the variables. In most case, this
        ;; will be 0 and 0. Because HDR will add
        ;; 1 pages for the stack and an offset of
        ;; &0040 to reserve the stack space.
        LDS   (>STARTBASIC)
        LDA   05
        ADD   BC
        LDD   (BC)
        STA   L
        LDD   (BC)
```

```
      STA   H
      LDA   (BC)
      STA   D
      MMPS  krn.HDR
      JR    NC,pop1outerr

      ;; Done. Close the file and exit with return
      ;; code &0000
      MMPS  krn.CLO
popout:
      POP   DE
      CLA
      STA   D
      STA   E
      RET

pop2outeof:
      LDA   err.EOF
      ;; Error when calling the kernel-jumps.
pop2outerr:
      POP   MN
pop1outerr:
      POP   MN

      ;; If the error code specify that the file
      ;; already exist, we skip the file deletion.
      CPA   err.EIS
      JR    Z,outerrnodel

      ;; Push the error code, because we want to
      ;; return it to our father.
      PUSH  A

      ;; The file has to be closed before to try
      ;; to delete it.
      MMPS  krn.CLO

      ;; Retrieve our global address. The name is
      ;; the argument page.
      MMPS  krn.GBL

      DEC   BC
      LDD   (BC)
      STA   L
      LDA   (BC)
      STA   B
      LDA   L
      STA   C

      ;; The file is deleted.
      MMPS  krn.DEL

      POP   A

outerrnodel:
      ;; Error. Return error code from A into
      ;; E and set D to 0.
      LD    D,00
      STA   E
```

```
      RET


      .TEXT
makerLOK:
      ;; Name of lock to prevent a new instance
      ;; of MAKER to run.
      "MKR.lock"

      .END
```

To test **MAKER**, return to BASIC and load (**CLOAD**) the executable **SHOWKEY.wav** from the **Images/Examples** directory.
Now, start MMPS (**CALL &C5**) and press **RCL**. Type **MAKER SHOWKEY** and wait for the **Closed!** message. Use the **SEL** key (on the left of **RCL**) to select the **STARTER** console. Press **RCL** and **ENTER**. If you see **>p: +4:0000** the **SHOWKEY** program is now installed.
The executable **FILES** should display an entry like this;

```
      FoRWE 01 01 003A SHOWKEY
```

Now launch **SHOWKEY**. Each time a key is pressed, its code is printed in hexadecimal. Use **ON/BREAK** to exit.

Note that this is not usable with the Standard Development (**stddev**) kernel.

FILES lists all entries in the File-System. It displays the type (**LCFQS**), the access right, the number of links, the number of pages reserved for this entry, its real length and its name. Each time **RCL** is pressed, the next entry is displayed.

```
      .IF   INCLUDED?
      ;; Align on next 64-bytes frontier
      .ALIGN:     40
      .ELSE
      ;; Origin does not matter, because code
      ;; and data are fully relocatable.
      .ORIGIN: 40C5
      .ENDIF

      .CODE

      ;; FILES - A file-system node entries list
      ;; FILES fetches all the file-system entries
      ;; nodes (FSR), gets the status of each (FEN)
      ;; and display the node type, the access
      ;; rights, the number of pages allocated, the
      ;; real entry ;; length and the entry name.

      ;; When the console output is refreshed by
      ;; pressing the key RCL, the next entry is
      ;; displayed

      ;; MMPS kernel-jump and macros
      ;;
      .INCLUDE:   MMPS.inc

FILES:
      .LOCAL

      ;; MMPS executable header.
      ;;
      .BYTE
      BLDHEADER   FILES.asm$$._length 01 0040

      .CODE
      ;; The process enters here
      ;;   BC contains the global address
      ;;   DE the in and out descriptors
      ;;   HL points to the arguments
      ;;
      ;; FSR need to have &FF in D at first
      ;; call
      LD    D,FF

forever:
      ;; HL points to argument page. We use
      ;; this dummy page allocated to pass
      ;; the arguments. Because we do not
      ;; expect some, it is our buffer !
      ;; This is not nice, but it reduces
      ;; the space needed.
      LD    BC,HL

      ;; Read the file system entries in
      ;; argument page + 5
```

```
      LDA   05
      ADD   BC
      MMPS  krn.FSR
      JR    NC,out_FS_END

      PUSH  HL
      LDA   0B
      ADD   BC
      PUSH  DE
      LD    DE,BC
      PUSH  DE

      LD    BC,PC
      LDA   *_*filesSTR
      ADD   BC
      ;; Here call the FMTSRING from the run
      ;; time with in
      ;;    BC the format string
      ;;    DE the formatted string
      ;;    HL the structure to format
      CALL  FMTSTRING

      POP   BC
      POP   DE

      ;; Restore all registers and write the
      ;; formatted string to the out descriptor
      LD    L,1A
      MMPS  krn.WRI

      POP   HL
      JR    forever

out_FS_END:
      ;; Flush the out descriptor. This is nice
      ;; if we output to a console, because we
      ;; will suspend until RCL is pressed. It
      ;; gives time for user to read the output.
      MMPS  krn.FLU

      ;; Return to father with &0000
      CLA
      STA   D
      STA   E
      RET
      SBC   C

      .TEXT
filesSTR:
            ".N.-.A .B .B .W .S        \00"

      .END
```

PROCESS lists all process created. It displays the PID, the process status bits, the priority, the events queue mask, the FID and the executable name. Each time **RCL** is pressed, the next valid process is displayed.

```
      .IF   INCLUDED?
      ;; Align on next 64-bytes frontier
      .ALIGN:     40
      .ELSE
      ;; Origin does not matter, because code
      ;; and data are fully relocatable.
      .ORIGIN: 40C5
      .ENDIF

      .CODE

      ;; PROCESS - Display all process created
      ;; PROCESS try to get the process status
      ;; (STA) for all process (0..7). If it
      ;; succeed, the process status bits and
      ;; and the executable name are displayed.
      ;; Else, go to next process-id.

      ;; When the console output is refreshed by
      ;; pressing the key RCL, the next process
      ;;s is displayed

      ;; MMPS kernel-jump and macros
      ;;
      .INCLUDE:   MMPS.inc

PROCESS:
      .LOCAL

      .BYTE
      BLDHEADER   PROCESS.asm$$._length 01 0040

      .CODE
      ;; program enters here
      ;;   BC contains the global address
      ;;   DE the in and out descriptors
      ;;   HL points to the arguments

      ;; HL points to argument page. We use
      ;; this dummy page allocated to pass
      ;; the arguments. Because we do not
      ;; expect some, it is our buffer !
      ;; This is not nice, but it reduces
      ;; the space needed.
      LD    BC,HL

      ;; The first process had pid 0
      LD    H,00

forever:
      ;; Prepare for STA, pid in H and BC
      ;; points to the process structure
      LDA   H
      PUSH  A
```

```
      STI   (BC)
      MMPS  krn.STA
      POP   A
      STA   H
      DEC   BC
      ;; If carry clear, the process does
      ;; not exists, so go next
      JR    NC,nextproc

      ;; Save all registers
      PUSH  HL
      PUSH  BC
      PUSH  DE

      LD    DE,BC
      LDA   10
      ADD   DE
      LD    HL,BC
      PUSH  DE

      LD    BC,PC
      LDA   *_*processSTR
      ADD   BC
      ;; Here call the FMTSRING from the run
      ;; time with in
      ;;    BC the format string
      ;;    DE the formatted string
      ;;    HL the structure to format
      CALL  FMTSTRING

      ;; Restore all registers
      POP   BC
      POP   DE

      ;; Write the formatted buffer to the out
      ;; descriptor
      LD    L,18
      MMPS  krn.WRI

      POP   BC
      POP   HL
nextproc:
      INC   H
      ;; Up to 8 process under MMPS
      CP    H,08
      JR    NC,forever

      ;; Flush the out descriptor. This is nice
      ;; if we output to a console, because we
      ;; will suspend until RCL is pressed. It
      ;; gives time for user to read the output.
      MMPS  krn.FLU

      ;; Return to father with &0000
      CLA
      STA   D
      STA   E
      RET
      SBC   C
```

```
      .TEXT
processSTR:
      ".L .P .L .B .L  .S          \00"

      .END
```

DISPLAY dumps in hexadecimal the content of the entry given as argument. It displays the name, the offset and 4 bytes from the entry. Each time **RCL** is pressed, the next 4 bytes are read and displayed.

```
      .IF   INCLUDED?
      ;; Align on next 64-bytes frontier
      .ALIGN:     40
      .ELSE
      ;; Origin does not matter, because code
      ;; and data are fully relocatable.
      .ORIGIN: 40C5
      .ENDIF

      .CODE

      ;; DISPLAY - A display file utility for MMPS
      ;; DISPLAY will dump in hexadecimal the file
      ;; or the entry named in argument.

      ;; It reads the file 4 bytes by 4 bytes, and
      ;; prints the file name, the offset and the
      ;; 4 bytes in hexadecimal form.

      ;; When the console output is refreshed by
      ;; pressing the key RCL, the next 4 bytes
      ;; are displayed

      ;; MMPS kernel-jump and macros
      ;;
      .INCLUDE:   MMPS.inc

DISPLAY:
      .LOCAL

      ;; MMPS executable header.
      ;;
      .BYTE
      BLDHEADER   DISPLAY.asm$$._length 01 0040

      .CODE
      ;; The process enters here
      ;;   BC contains the global address
      ;;   DE the in and out descriptors
      ;;   HL points to the arguments
      ;;
      LDA   E
      PUSH  A

      ;; Get file name to display from the
      ;; argument page and open this file
      ;; in read-only mode
      LD    BC,HL
      LD    D,04
      MMPS  krn.OPN
      JR    C,fileopen
outerr:
      ;; Set &00 to D and MMPS error code
      ;; into E and return to father
      LD    D,00
```

```
      STA   E
      POP   A
      RET

fileopen:
      POP   A
      STA   D

      ;; HL points to argument page. We use
      ;; this dummy page allocated to pass
      ;; the arguments. Because we do not
      ;; expect some, it is our buffer !
      ;; This is not nice, but it reduces
      ;; the space needed.
      PUSH  HL

      ;; Skip the name. The buffer will start
      ;; after, so the name, the offset and
      ;; 4 bytes will be formatted.
      LD    C,08
skipname:
      LDI   (HL)
      JR    Z,formatsp
      DEC   C
      JR    C,skipname
formatsp:
      LDA   \x20  ;; Space character
      DEC   HL
loopsp:
      STI   (HL)
      DEC   C
      JR    C,loopsp

      POP   BC
forever:

      ;; Main loop
      LD    HL,BC
      LDA   1A
      ADD   BC
      PUSH  HL

      ;; Get into HL the current offset in the
      ;; file pointed by the descriptor E
      MMPS  krn.TEL

      LDA   H
      STI   (BC)
      LDA   L
      STI   (BC)

      ;; Read 4 bytes from descriptor E
      LD    L,04
      MMPS  krn.REA
      LDA   L
      POP   HL

      ;; Error or no byte read (L=0), so we
      ;; expect that is the end-of-file
```

```
        JR    NC,out
        JR    Z,out

        PUSH  DE
        PUSH  HL
        PUSH  A
        LDA   09
        ADD   HL
        PUSH  HL
        POP   DE
        LD    HL,BC

        LD    BC,PC
        LDA   *_*displaySTR
        ADD   BC
        DEC   HL
        DEC   HL
        ;; Here call the FMTSRING from the run
        ;; time with in
        ;;    BC the format string
        ;;    DE the formatted string
        ;;    HL the structure to format
        CALL  FMTSTRING

        POP   A
        POP   BC
        STA   H
        SL
        RCF
        ADC   H
        ADC   0E
        STA   L
        POP   A
        PUSH  A
        STA   E
        ;; Write the formatted string to the out
        ;; descriptor
        MMPS  krn.WRI

        POP   DE
        JR    forever

out:
        LDA   D
        STA   E
        ;; Flush the out descriptor. This is nice
        ;; if we output to a console, because we
        ;; will suspend until RCL is pressed. It
        ;; gives time for user to read the output.
        MMPS  krn.FLU

        ;; Return to father with &0000
        CLA
        STA   D
        STA   E
        RET
        SBC   C

        .TEXT
```

```
displaySTR:
      ":.W .B .B .B\00"

      .END
```

The minimal file system embed a run time. This library contains some useful routines:

highTOHEX - &1819
lowTOHEX - &181A

Convert the high or low 4-bits digit of **A** into a hexadecimal character
0123456789ABCDEF
In:
A contains a 8-bits value
Out:
A contains a hexadecimal character

NEXTARG - &1830

Fetch the space in a string, nulls the space and return a pointer to the first argument.
In:
HL contains address of the string to parse
Out:
BC contains the origin address (**HL** in)
HL contains the address of the next character after a space.
In he string pointed by BC, the leading space are replaced by a **&00**.

FMTSTRING - &1850

Format an output buffer according to a format string and arguments.
In:
BC contains the address of format string ended by **&00**
DE contains the address of the output buffer
HL contains the address of a binary buffer to be formatted
Out:
BC, **DE** and **HL** points to the end of their respective buffers
The string is filled in the buffer pointed by **DE**. The string ends by a **&00**.

The format string pointed by **BC** is on the form:
.<format char>.<fc>.<fc>
where **<format char>** is

- **.** := put a **.**
- **-** := skip next byte without putting in buffer (equ. to **INC HL**)
- **H** := put the high digit (bits7-4) in hexa
- **L** := put the low digit (bits3-0) in hexa
- **B** := put the 8-bits byte in hexa
- **W** := put the 16-bits word in hexa
- **b** := put the 8-bits byte in binary
- **h** := put the high digit (bits7-4) in binary
- **l** := put the low digit (bits3-0) in binary
- **C := put the character**
- **S** := put the string until &00 is found
- **N** := put the node entry type, ie, one of '**LCFQS**'
- **A** := put the access mode, ie, one of '**ORWMCE**'. Cleared bits are in lower case

P := put the process bits, ie, one of '**DOILQPR**' added with a leading '**K**' while the process has enters a kernel-jump

any other character is copied directly to the formatted buffer.

After each format, the **HL** register is incremented.

In the minimum file system, 6 executable are present:

STARTER	process started by the kernel. A very-very-very-light shell
FILES	list all entries present in the file system, type, size, name
PROCESS	list all process created into the scheduler, PID, status, FID, name
DISPLAY	display in hexadecimal the content of the entry given as argument
DEBUG	debug the process given as argument
MAKER	copy code from BASIC and create executable header

Some other entry are available to use with **DISPLAY** and to get some debug information about the MMPS kernel:

SYSVAR	the volatile area of MMPS
SYSMEM	the memory of MMPS

MMPS images

Several images are provided within this ZIP; The Standard Development (**stddev**) or Full (**stdfull**) images may be installed in all PC-1500, PC-1500A and PC-2. The 1500A Development (**1500Adev**) or 1500A Full (**1500Afull**) are only workable with a PC-1500A.

Note that MMPS requires a CE-161 (16Kbytes) extension module.

If you choose the Standard Development (**stddev**), do the following, in **PRO** mode:

```
POKE &7865,&48,&00
NEW
```

and load the 3 wav with **CLOAD M**

```
Images/stddev/mmps-090498-stddev.wav
Images/stddev/mmps-fs+lib-stddev.wav
Images/stddev/mmps-volatile-stddev.wav
```

This image is really for MMPS kernel development. For a normal usage, the images below are preferred.

If you choose the Standard Full (**stdfull**), do the following, in **PRO** mode:

```
NEW &4300
```

and load the 3 wav with **CLOAD M**

```
Images/stdfull/mmps-090498-stdfull.wav
Images/stdfull/mmps-fs+lib-stdfull.wav
Images/stdfull/mmps-volatile-stdfull.wav
```

If you choose the 1500A Development (**1500Adev**), do the following, in **PRO** mode:

```
NEW &5000
```

and load the 3 wav with **CLOAD M**

```
Images/1500Adev/mmps-090498-1500Adev.wav
Images/1500Adev/mmps-fs+lib-1500Adev.wav
Images/1500Adev/mmps-volatile-1500Adev.wav
```

If you choose the 1500A Full (**1500Afull**), do the following, in **PRO** mode:

```
NEW &4000
```

and load the 3 wav with **CLOAD M**

```
Images/1500Afull/mmps-090498-1500Afull.wav
Images/1500Afull/mmps-fs+lib-1500Afull.wav
Images/1500Afull/mmps-volatile-1500Afull.wav
```

Always load the 3 wav from the same image. Do not mix them, because it may result some unrecoverable crash.

License