10 Expansion ROMs, Resident System Extensions and RAM Programs.

The system can address up to 252 expansion ROMs, mapped over the top 16K of memory, starting at #C000. The Kernel supports two varieties of expansion ROM, foreground and background. A resident system extension (RSX) is similar in use to a background ROM, but must be loaded into RAM before it can be used.

A foreground ROM contains one or more programs, only one of which may be running at one time. The on-board BASIC is the default foreground program. Other possible foreground programs are:

- other systems, such as FORTH or CP/M.
- applications, such as a Word Processor or Spread Sheet.
- tools, such as an Assembler or Debugger.

A RAM program, once loaded, takes over the machine in much the same way as a foreground ROM program. Games will generally be RAM programs.

There may be up to 16 background ROMs, each of which provides some sort of service independent of the foreground program. It is expected that expansion peripherals will each have an associated background ROM containing suitable support routines. Other background ROMs may augment the existing machine software; for example, by providing further graphics functions.

A resident system extension (RSX), once loaded, provides some sort of service in the same way as a background ROM. An RSX might, for example, provide special support for a given printer - where it is more economical to provide the software on cassette rather than in ROM (or PROM).

10.1 ROM Addressing.

Expansion ROMs have ROM address in the range 0..251. To select a given ROM the Kernel sets its ROM address by writing to I/O address #DF00. If a ROM is fitted at the address selected, then all further read accesses to the top 16K of memory will return data from the expansion ROM. If no ROM is fitted at the currently selected ROM address the contents of the on-board ROM are returned.

When the machine is first turned on ROM 0 is selected as the foreground program. If no expansion ROM is fitted at ROM address 0, the on-board ROM is used, and BASIC is entered. If an expansion ROM is fitted at ROM address 0 it takes precedence over the on-board ROM.

In V1.0 firmware background ROMs must be fitted at ROM addresses in the range 1...7. Foreground ROMs must be fitted so that there are contiguous ROMs from address 1. When searching for a foreground ROM the kernel starts at address 0 and works upwards until the first address greater than 0 if found.

In V1.1 firmware background ROMs may be fitted at ROM addresses in the range 0...15. Foreground ROMs must be fitted contiguously from address 16 or at any background ROM address. When searching for a foreground ROM the kernel starts at address 0 and works upwards until the first unused address greater than 15 is found.

In either case if an expansion ROM 0 is fitted the on-board ROM can still be accessed at the first unused ROM address.

The Kernel supports a ‘far address’ which may be used to call subroutines in expansion ROMs. The ‘far address’ is a three byte object, the last byte of which is a ROM select number. Since the arrangement of ROMs in an expansion card is quite arbitrary the ROM select part of the 'far address' must be established at run time. The 'sideways' ROM addressing facility allows a foreground program to occupy up to four contiguous ROM select addresses, and supports subroutine calls between the ROMs without requiring the program to know the actual ROM address of any of them.

10.2 The Format of an Expansion ROM.

An expansion ROM may be up to 16K bytes long, the first byte being at address #C000. The first few bytes of the ROM are the 'ROM Prefix' and must take the form:

Byte 0	:	ROM type.
Byte 1	:	ROM Mark Number.
Byte 2	:	ROM Version Number.
Byte 3	:	ROM Modification Level.
Byte 4	:	External Command Table.

The ROM type specifies what sort of ROM this is and must take the following values:

0:	Foreground ROM.
1:	Background ROM.
2:	Extension ROM.

The on-board ROM must be unique in having bit 7 of the type byte set (thus its type byte is #80). This marker is used to detect the end of foreground ROMs. If a foreground program will not fit into a single ROM then the extra ROMs required should be marked as extension ROMs.

The mark number, version number and modification level may be set to any values required.

The external command table comprises a list of command names and a jumpblock. Each command name is implicitly associated with the same numbered entry in the jumpblock. The table takes the form:

Bytes 0..1	: Address of command name table
Bytes 2..4	: Jumpblock entry 0
Bytes 5..7	: Jumpblock entry 1
…etc	: … etc

The command name table is a list of names, each of which may be up to 16 characters long. The last character of each name must have bit 7 set but no other character may. The table is terminated by a null (character 0) after the last character of the last name. Apart from the fact that all characters must be in the range 0..127 and that the first character may not be a null, there are no restrictions on the characters in command names. However, if unsuitable characters are chosen it may prove impossible for programs such as BASIC to access the commands. BASIC expects alphabetic characters in the command names to be in upper case and will not allow characters such as a space or comma in the command name.

The ROM prefix for the on-board ROM is:

```
            ORG     #C000               ;Start of the ROM

            DEFB    #80+0               ;On board ROM, Foreground
            DEFB    1                   ;Mark 1
            DEFB    0                   ;Version 0
            DEFB    0                   ;Modification 0

            DEFW    NAME_TABLE          ;Address of name table.

            JP      START_BASIC         ;The only entry in the jumpblock

NAME_TABLE: DEFB    'BASI','C'+#80      ;The only command name
            DEFB    0                   ;End of table marker
```

The ROM prefix for a serial I/O card might be:

```
            ORG     #C000               ;Start of ROM

            DEFB    1                   ;Background ROM
            DEFB    0                   ;Mark 0
            DEFB    5                   ;Version 5
            DEFB    0                   ;Modification 0

            DEFW    NAME_TABLE          ;Address of name table

            JP      EMS_ENTRY     ;0 Background ROM power-up entry
            JP      RESET               ;1
            JP      SET_BAUD_RATE       ;2
            JP      GET_CHARACTER       ;3
            JP      PUT_CHARACTER       ;4
                    ...etc

NAME_TABLE: DEFB    'SIO DRIVE','R'+#80 ;0
            DEFB    'SIO.RESE','T'+#80  ;1
            DEFB    'SIO.SET.BAU','D'+#80 ;2
            DEFB    'SIO.GET.CHA','R'+#80 ;3
            DEFB    'SIO.PUT.CHA','R'+#80 ;4
                    ...etc
            DEFB    0                   ;End of table marker
```

Note that the command name table entry for the power-up entry includes a space. This is still a legal name but the BASIC will never be able to generate it because of the way it uses spaces. Because BASIC cannot generate the name it is impossible for a BASIC user to call the power-up entry by mistake (see section 10.4)

10.3 Foreground ROMs and RAM Programs.

Each of the entries to the foreground ROM is expected to represent a separate program, whose name is given by the corresponding entry in the name table. The first entry of ROM 0 is the default power-up entry point at the end of EMS.

Once a RAM program has been loaded it is treated much like a foreground ROM, except that it does not have a ROM prefix, and the required entry point is determined separately.

Just before a foreground program is entered the machine is reset to its EMS state; i.e. all the hardware and all the firmware are initialized. The environment and entry conditions are as follows:

Memory:

Section 2 describes the memory layout of the system. Three areas of memory are available to the program.

1. The Static Variable Area.

The area from #AC00 to #B0FF inclusive is reserved for use by the foreground program - although it may use more or less as it requires. It is also possible to reserve a foreground data area starting at #0040 if this is required.

2. The Stack.

The hardware stack is set to an area immediately below #C000 which is at least 256 bytes long.

3. The Main Memory Pool.

Most of the rest of memory will be available to the foreground program, depending on what memory is taken by any background ROMs which the foreground program chooses to initialize.

Registers:

The base and limit of the free memory area are passed to the program in registers.

BC = Address of the highest usable byte in memory. (#B0FF)
DE = Address of the lowest byte in the memory pool. (#0040)
HL = Address of the highest byte in the memory pool. (#ABFF)

Note that the program is free to use any memory between the address given in DE and the address in BC inclusive (i.e. #0040 to #B0FF). The contents of HL reflect the standard allocation for static variables; the program is free to use more, or less, as the mood takes it. Also a foreground data area may be reserved at the bottom of store as well. The program should set HL and DE to reflect the area it is using for variables before initializing any background ROMs (see below).

SP is set to the machine provided area #C000. The program can depend on at least 256 bytes of stack.

The contents of the other registers is indeterminate. Note that the alternate register set (AF' BC' DE' HL') is not available to the program. (But see Appendix XI).

ROM select and state:

For ROM programs:	The foreground ROM is selected. The upper ROM is enabled. The lower ROM is disabled.
For RAM programs:	No ROM is selected. The upper ROM is disabled The lower ROM is disabled.

General:

Interrupts are enabled.

All hardware and firmware is in its initial state. In particular any expansion devices fitted have been reset, but not yet initialized.

It is the foreground program's responsibility to initialize any background ROMs required and to load and initialize any RSXs. The Kernel entry 'KL ROM WALK' looks for background ROMs and initialises any that it finds. The Kernel entry 'KL INIT BACK' will initialize a particular background ROM. These entries must be passed the addresses of the lowest and highest bytes in the memory pool which is why the foreground program must reserve its fixed data area before winding up the background ROMs. The background ROMs may allocate memory for their own use by moving either or both boundaries. If, therefore, the foreground program does allow background ROMs to function it must cope with a memory pool whilst bounds are not fixed until after all background ROMs have been initialized. Note that the location of the foreground program's data areas are fixed whilst a background program must deal with variable data areas.

If background ROMs are not initialized then the memory map is very simple, but since discs, light pens, etc are likely to use background ROMs for support software it is rather limiting not to allow background ROMs even for an apparently 'dedicated' game.

The on-board BASIC initializes all background ROMs at EMS. The user chooses whether to load any RSXs from tape.

10.4 Background ROMs.

Background ROMs lie dormant until initialized by the foreground program. During initialization the background software may allocate itself some memory and initialize any data structures an hardware. Providing the initialization is successful the Kernel places the ROM on the list of possible takers for external commands.

The first entry in a background ROM's jumpblock is it initialisation routine. This routine must only be called by the firmware when the ROM is initialized it is not meant for the user to call. Tricks such as including a space in the name makes it impossible for BASIC to generate the correct name and hence impossible for a BASIC user to call the entry. The entry and exit conditions for the initialisation routine are:

Entry:

DE contains the address of the lowest byte in the memory pool.
HL contains the address of the highest byte in the memory pool.

Exit:

If the initialization was successful

Carry true.
DE contains the new address of the lowest byte in the memory pool.
HL contains the new address of the highest byte in the memory pool.

If the initialization failed:

Carry false.
DE and HL preserved.

Always:

A, BC and other flags corrupt.
All other registers preserved.

Notes:

The upper ROM is enabled and selected.
The lower ROM is disabled.

The routine may not use the alternate register set.

The ROM may allocate itself memory either at the top or the bottom of the memory pool (or both), simply by changing the appropriate register and returning the new value. For example, to reserve 256 bytes given an address of #AB7F as the top of the pool the program would subtract 256 from HL giving a new top of pool address of #AA7F. The area preserved would be from #AA80 to #AB7F inclusive.

The carry false return is only recognized in V1.1 firmware. In V1.0 firmware this will be treated as if carry was returned true.

When the initialization routine returns, the Kernel stores the address of the base of the upper area which the ROM has allocated itself (i.e. HL+1). Whenever an entry in the ROM is called this address is passed in the IY index register. This allows the ROM routine to access its upper variable area easily enough even though it was allocated dynamically. Access to any lower variable area should be done via pointers in the upper area. Since background ROMs do not use absolute areas of memory, problems of background ROMs clashing with each other or with the foreground program will never arise. Note that a background ROM is very likely to expect that its upper data area lies above #0040 so that it is accessible irrespective of whether the lower ROM is enabled or not.

If the initialization is successful then the Kernel also places the ROM on its list of possible handlers of external commands (see below). Note that when the list is scanned for external commands the latest addition is tried first. The entry KL ROM WALK processes the ROMs in reverse address order (15, 14, ...0), ignoring any gaps of foreground ROMs, thus the ROMs will be searched in the order 0, 1, ...15.

10.5 Resident System Extensions.

An RSX is similar to a background ROM. Responsibility for loading an RSX and providing it with memory lies with the foreground program. To fit in with the dynamic allocation of memory to background ROMs it is recommended that RSXs should be position independent or relocated when loaded. An RSX could be relocated by writing a short BASIC 'loader' program which reads the RSX in a format which may be relocated easily and POKEs into store.

Once an RSX is load it may be placed on the list of possible handlers of external commands (see following page) by calling KL LOG EXT, passing it the address of the RSXs external command table and a four byte block of memory (in the central 32K of RAM) for the Kernel's use. The format of the table is exactly the same as for a background ROM (see section 10.2). The only difference is in the interpretation of the table - the first entry in the jumpblock is not called automatically by the Kernel and thus need not be the RSX's initialization routine.

For example, the way to add an external command table for a graphics extension for BASIC might be:

```
INITIALIZE:    LD     HL,WORK_SPACE          ;RSX power-up Routine
               LD     BC,RSX_TABLE
               JP     KL_LOG_EXT

WORK_SPACE:    DEFS   4                      ;Area for Kernel to use

RSX_TABLE:     DEFW   NAME_TABLE
               JP     DRAW_CIRCLE            ;0
               JP     DRAW_TRIANGLE          ;1
               JP     FILL_AREA              ;2

NAME_TABLE:    DEFB   'CIRCL','E'+#80        ;0
               DEFB   'TRIANGL','E'+#80      ;1
               DEFB   'FIL','L'+#80          ;2
               DEFB   #00
```

Note that when the list is scanned for external commands the latest addition is tried first. Since RSX's will, in general, be loaded after background ROMs have been initialized, RSX commands will take precedence over those in background ROMs. The entry and exit conditions for external commands are discussed in the following section (section 10.6).

10.6 External Commands.

Once the foreground program has decided that it has an external command on its hands it should call the Kernel entry KL FIND COMMAND, passing to it a string giving the command name. This routine first attempts to find an RSX or a background ROM whose external command table contains the command. Only those RSXs and ROMs which have been suitably initialized are taken into consideration. If the command is found then the 'far address' of the corresponding jumpblock entry is returned (see section 2.3). If the command is not found the routine starts at ROM 0 and searches for a foreground ROM whose external command table contains the command. If a foreground ROM is found,then the system resets and enters the appropriate foreground program. If no match for the command can be found a failure indication is returned.

Note that the external command mechanism allows both for finding of background and RSX routines, and for switching of foreground programs. Note also that the first command name in a background ROM corresponds to the implicit initialization entry, and should not be used as a command.

The first time a background or RSX routine is used the external command mechanism should be used to establish its jumpblock address. This may then be stored and used directly for subsequent calls of the routine. It is foolish to assume that a particular background ROM is always plugged into the same socket or that a relocatable RSX is always located at the same address.

The first time a background or RSX routine is used the external command mechanism should be used to establish its jumpblock address. This may then be stored and used directly for subsequent calls of the routine. It is foolish to assume that a particular background ROM is always plugged into the same socket or that a relocatable RSX is always located at the same address.

It is the foreground program's responsibility to invoke the external command once its address has been found, and to pass its parameters in a suitable form. BASIC in the on-board ROM functions as follows, and should serve as a model for other foreground programs if only to allow common use of commands by other systems:

> An external command is identified by a vertical bar ('|') followed by the command name, optionally followed by a list of parameters. The bar does not form part of the command name. The command name must consist of alphabetic characters (which are converted to upper case), numeric characters or dots.
>
> Parameters are passed by value, that is each parameter may be a numeric expression, the calculated value of which is passed, or an address. The number and type of parameters must be agreed between the BASIC program and the command because BASIC performs no checking.
>
> Each parameter passed is a two byte number, whose interpretation depends on its type:

Integer expression:	two's complement value of the Integer result.
Real expression:	the Real result forced to Unsigned Integer.
Variable reference:	address of the value of a variable (for a string this is the address of the descriptor).

> A string descriptor is three bytes long. Byte 0 contains the length of the string. Bytes 1 and 2 contain the address where the string is stored. If the string length is 0 then the address of the string is meaningless. String variables may be changed providing that the string descriptor is not altered in any way.

Entry:

> A contains the number of parameters.
> IX contains the address of the parameters.
> IY contains the address of the ROM's upper data area if the command was found in a background ROM. If the command was found in an RSX's external command table then IY is undefined.

Exit:

> AF, BC, DE, HL, IX and IY corrupt.
> Alternate register set untouched.

Notes:

> Index register IX contains the address of the parameters. If there are no parameters then the ith parameter is at offset $(n-i)\times 2$ from the index register address - so the 1st parameter is at the largest offset, and the last parameter is pointed to by IX.
>
> The IY register is set by the Kernel and not by BASIC. The A and IX registers and the parameter area are set by BASIC.

10.7 Examples.

a) A simple external command.

This example uses the BIOS routine SET MESSAGE that is available as an external command under AMSDOS. SET MESSAGE turns on or off the disc error messages and has the following interface:

```
SET MESSAGE                         Command name: Control A

 Entry conditions:

           A = #00 => Turn disc error messages on.
           A = #FF => Turn disc error messages off.

 Exit conditions:

           A = Previous state.
           HL and flags corrupt.
```

Before it is possible to use the external command it is necessary to establish the store and far address of the routine. This may be performed as follows:

```
        LD      HL,CMD_NAME                 ;Pointer to command name
        CALL    KL_FIND_COMMAND             ;Ask Kernel where it is
        JR      NC,ERROR_ROUTINE            ;Command not found error
;
        LD      (CMD_FAR_ADDRESS+0),HL      ;Store address
        LD      A,C
        LD      (CMD_FAR_ADDRESS+2),A       ; Store ROM number

        …

CMD_NAME:           DEFB #01+#80            ;Control A = #01
CMD_FAR_ADDRESS:    DEFS 3                  ;Area for storing far address
```

Having found the far address of the routine it can now be called. For example:

```
        LD      A,0                         ;Enable messages
        RST     3                           ;Far CALL
        DEFW    CMD_FAR_ADDRESS             ;Pointer to far address
```

b) A complex external command

This example uses the INCHAR external command provided by the serial interface. INCHAR reads a character from the Serial Interface and has the following interface:

```
        INCHAR                                  Command name: INCHAR

          Entry conditions:

                  A = Number of parameters (should be 2).
                  IX = Address of parameter block.
                          IX+2 = Address to store status.
                          IX+0 = Address to store character read.

          Exit conditions:

                  AF, BC, DE, HL, IX and IY corrupt.
```

Before it is possible to use the external command it is necessary to establish and store the far address of the routine. This may be performed as follows:

```
        LD      HL,CMD_TABLE                ;Pointer to command name
        CALL    KL_FIND_COMMAND             ;Ask Kernel where it is
        JR      NC,ERROR_ROUTINE            ;Command not found error
;
        LD      (CMD_FAR_ADDRESS+0),HL      ;Store address
        LD      A,C
        LD      (CMD_FAR_ADDRESS+2),A       ;Store ROM number
        …

CMD_NAME:               DEFB 'INCHA','R'+#80
CMD_FAR_ADDRESS:        DEFS 3 ;Area for storing far address
```

Having found the far address of the routine it can now be called. For example:

```
        LD      A,2                         ;2 parameters
        LD      IX,PARAM_BLOCK              ;Address of parameter block
        RST     3                           ;FAR CALL
        DEFW    CMD_FAR_ADDRESS             ;Pointer to far address
        LD      HL,(STATUS)                 ;HL=Serial Interface status
        LD      A,(CHAR)                    ;A=Character read (if any)
        …

PARAM_BLOCK             DEFW STATUS         ;First parameter is status
                        DEFW CHAR           ;Second parameter is character
;
STATUS:                 DEFW #0000
CHAR:                   DEFW #0000
```

c) Passing different types of parameter

This exmple uses an invented external command which takes a string of characters, looks these up in an index and returns a reference number. The external command is assumed to be designed to be called from BASIC as follows:

```
|REFNUM,@CHARTRING$,INDEXNUM,@REFNUM
```

i.e. The first parameter is a string (whose address is passed) which is to be looked up. The second parameter is a number specifying which index to use, and the third parameter is a variable (whose address is passed) which is to be set to the required reference number.

The far address of the routine can be established in the same way as was described in the previous two examples. To call this routine from a machine code program it is necessary to set up the parameter block and a string descriptor. The following subroutine does this:

```
GET_REF_NUM:                            ;Entry:  HL=Address of string.
                                        ;        A =Length of string.
                                        ;        DE=Index number.
                                        ;Exit:   HL=Reference number.
                                        ;        AF,BC,DE,IX,IY corrupt.
        LD      (STR_DESCRIPTOR+0),A    ;Store length of string.
        LD      (STR_DESCRIPTOR+1),HL   ;Store address of string
        LD      (PARAM_BLOCK+2),DE      ;Store index number
;
        LD      A,3                     ;3 parameters
        LD      IX,PARAM_BLOCK          ;Address of parameter block
        RST     3                       ;FAR CALL
        DEFW    CMD_FAR_ADDRESS         ;Pointer to far address
;
        LD      HL,REF_NUM              ;HL=Reference number
        RET

PARAM_BLOCK: DEFW STR_DESCRIPTOR ;First paramater is address of string
                                        ; desriptor
                DEFW #0000              ;Second parameter is index number
                DEFW REFNUM             ;Third parameter is address of store
                                        ; for reference number
;
STR_DESCRIPTOR          DEFB #00        ;Length
                        DEFW #0000      ;Address
;
REFNUM                  DEFW #0000
```

The external command routine that is being called has to pick the parameters out of the parameter block and it might work as follows:

```
            LD      L,(IX+0)
            LD      H,(IX+1)            ; HL=Address of string descriptor
;
            LD      A,(HL)
            INC     HL                  ;A=Length of the string

            LD      E,(HL)
            INC     HL
            LD      D,(HL)
            EX      HL,DE               ;HL=Address of string.
;
            LD      E,(IX+2)
            LD      D,(IX+3)            ;DE=Index number

            ...                         ;Look up string

            LD      (IX+4),L
            LD      (IX+5),H            ;Store resulting reference
            RET                         ;number
```