

Dk'tronics 64K/256K Memory Expansion  
For the Amstrad CPC computers

**dk'tronics**



- EXPANDS THE 464/464+ MEMORY TO 320K.
- SUPPLIED WITH BANK SWITCHING RSX SOFTWARE.
- COMPATIBLE WITH dk'tronic 256k SILICON DISC.
- REQUIRES NO EXTRA POWER SUPPLY.

GOVAN ROAD, FENTON INDUSTRIAL ESTATE, FENTON, STOKE-ON-TRENT, ST4 2RS.  
TEL 0782 744707 FAX 0782 744292

**64K**  
**MEMORY**  
**EXPANSION**

FOR THE  
AMSTRAD/ SCHNEIDER  
464/464+ COMPUTER

**DATEL**  
Electronics  
L I M I T E D

CONTENTS

	PAGE.	
<u>64K AND 256K MEMORY EXPANSION UNITS.</u>		
1.0	PREFACE	1
1.1	INSTALLATION	2
1.2	USING YOUR EXTRA RAM	2
1.3	RAM TEST	3
1.4	EXTENDED BASIC COMMANDS	3
1.5	WINDOWS AND PULLDOWN MENUS	4
1.5.1	MORE WINDOWING	5
1.6	ARRAYS, VARIABLES AND STRINGS	6
1.6.1	MORE ABOUT ARRAYS	9
1.6.2	STRING STORAGE	10
1.7	ANIMATION AND PICTURE SHOWS	11
1.8	ADVANCED PROGRAMMING	13
1.9	PEEKING AND POKING	15
1.10	PROGRAMMING WITHOUT RSX's	16
1.11	TECHNICAL DETAILS	17
1.11.1	THE LOAD ADDRESS	17
1.11.2	SAVING TO DISC	17
1.11.3	INCREASING CP/M 2.2 TPA	17
1.11.4	COMMERCIAL PROGRAM COMPATIBILITY	17
1.11.5	USING CP/M 2.2	18
1.12	ERROR MESSAGES	18
1.13	REFERENCE OF RSX COMMANDS	19
1.14	TECHNICAL DETAILS (HARDWARE)	20
1.15	CUSTOMIZING YOUR CP/M+ DISC	22

### 64K and 256K MEMORY EXPANSIONS.

These units are available for the CPC 464, 664 and 6128 computers.

By using the 64K upgrade the 464 and 664 computers will have the same amount and configuration of RAM as the CPC 6128. The 256K gives an extra 192K on top of this! The expansion will allow the use of CP/M+ as supplied with the CPC 6128 with its massive 61K TPA opening up an even larger software base for Amstrad users. There is also an utility for increasing the TPA on CP/M 2.2 to 61K.

The RAM can be accessed by means of bank switching using a single I/O port. Memory is actually switched in and out of the 64K Z80 address space in 16K sub-blocks, as are the ROMs. The port determines which particular combination of the original four 16K sub-blocks and any new sub-blocks from the expansion RAM will occupy the 64K address space at any time. Control of the I/O port can be from either BASIC or machine code.

To use the additional 64K/256K of RAM, the expansion is supplied with bank switching software (although it can be switched without this software).

The software adds some extra BASIC commands, RSXs, which make it possible to use the second 64K (or 3rd, 4th and 5th in the case of the 256K expansion) for storage for screens, windows, graphics and BASIC arrays. This ability means that you can write much larger BASIC programs, as most of the memory on the unexpanded CPC464/664 is normally used for arrays, variables and graphics.

The additional BASIC commands are :

BANK,n	Map a bank of 16K directly into memory space.
SWAP	Alternate between the low and high screens.
LOW	Change to the low screen.
HIGH	Change to the high screen. (Default screen).
SAVES,n	Store a screen to a 16K bank.
LOADS,n	Retrieve a screen from a 16K bank.
SAVEW,w,n	Store a window's contents into expansion RAM.
LOADW,w,n	Load a window with data from the expansion RAM.
SAVED,n,s,l	Transfer original RAM to expansion RAM.
LOADD,n,s,l	Load original RAM from expansion RAM.
PEEK,n,s,v	Read the value of a byte in expansion RAM.
POKE,n,s,v	Change a byte in the expansion RAM.

These commands make such features as pull down menus, full screen animation and large spreadsheet type programs or databases very easily programmed from BASIC as never before possible on the unexpanded CPC464 and 664 computers.

WARNING.

Ensure that the power to your Amstrad computer is switched OFF before you fit the interface to the expansion socket. Failure to comply with these instructions may cause permanent damage to the RAM pack or the computer.

1.1 Installation.

Power down your Amstrad computer. Plug the RAM pack into the socket on the back of the computer. On the CPC 464 this socket is labelled 'Floppy Disc', on the CPC 664 and CPC 6128 the socket is labelled 'Expansion'. Other expansions such as the Amstrad Disc interface for the CPC 464, DK'tronics Lightpen and Speech Synthesizer, or ROM expansions can be fitted into the expansion socket on the back of the RAM pack. Now switch on the computer.

The computer should power up as normal. If it fails to do so, check that all the connections are correctly made. Note that all DK'tronics products have a key location on the connector to ensure that there can be no alignment problems. OTHER interfaces may not have this keyway (the Amstrad disc interface is the most familiar example). Hence any connection problems will usually lie between the RAM pack and these expansions. If this is the case, try reconnecting the interfaces BEFORE inserting the RAM pack into the computer. This will give you a better view when lining up the pins.

If the computer fails to power up, or crashes on power up (Miscellaneous patterns all over the screen!), the monitor may cut out the power to the computer. On the colour monitor, just switch the MONITOR off and then attempt to reconnect as above. The monochrome monitor may have to remain switched off for several seconds before power will be reinstated to the computer.

It is very unlikely that the computer will fail to power up with the RAM pack alone. If this is the case, then the fault will probably lie with the RAM pack. \* Return the RAM pack to RAM ELECTRONICS if this is the case.

\* IT IS ESSENTIAL THAT YOU COMPLETE YOUR WARRANTY REGISTRATION CARD AND RETURN IT TO US IMMEDIATELY UPON PURCHASING THIS PRODUCT FROM YOUR DEALER (UK ONLY).

1.2 USING YOUR EXTRA RAM.

There are two ways to use the extra RAM. There is a cassette supplied with the RAM pack containing extensions to BASIC. Here the extra RAM can be used simply from BASIC programs. Alternatively, the RAM is accessible both from BASIC and machine code using the OUT command. The experienced programmer will be able to use the RAM for whatever he pleases and write custom software for that purpose. Commercial programs will no doubt use this approach.

The second method is described in detail in section 1.10. The first way is explained in the following chapters:-

With the computer set up as above, load the RSX software from the cassette tape supplied:-

On disc systems type '|TAPE' and press <ENTER>

- b) Type 'RUN' and press <ENTER>.
- c) The loading sequence is described in detail in your Amstrad user manual.
- d) When the program has finished loading, you will be asked to enter a loading address. Just press <ENTER> for now. (See section 11.)
- e) The computer will test the RAM and then print out how much RAM you have got, then the computer memory will be clear ready for your own programs.

### 1.3 RAM TEST.

When the RSX code is first loaded, it does an extensive RAM test. Should the RAM not function correctly the program will inform you that an error has been found. Along with this, it will print out diagnostic information to help in the repair of the RAM pack.

In the unlikely event that an error is found, please note the information that is given and return the RAM pack for replacement or repair.  
(See warranty registration note.)

### 1.4 EXTENDED BASIC COMMANDS.

There are a total of twelve extra commands provided by the RSXs on tape. Some may have parameters, some will not. Sometimes the command may have different formats and numbers of parameters. We have tried to discuss each command in its simplest form and later sections will describe added parameters which make the command more flexible and economic on memory.

You may have noticed that during the RAM test, the computer printed out the number of the 'bank' it was testing. Each bank is 16K of memory. For the 64K expansion there are 4 banks while the 256K RAM pack has 16 banks. To access a particular part of the expansion's memory there has to be a bank number and possibly a bank address.

For example, type:- '|SAVES,1' and press <ENTER>

The computer will respond with READY. What you have done is to store what was on the screen into bank 1.

Now clear the screen using CLS. To get the screen's contents back, type:- '|LOADS,1' and press <ENTER>

You can save as many screens as you have memory for. That means four screens on the 64K RAM and sixteen screens for the 256K RAM. Screen displays could be created from another program or drawn using a lightpen. Store these on tape or disc then load them back into RAM for use throughout the program. Screen displays which take a long time to create within a program, for example mazes, can be created once, then stored for instant use whenever necessary.

The command can be summarized:-

```
|SAVES,[n] save data to bank (n= the bank number)
|LOADS,[n] load data from bank
```

### 1.5 WINDOWS AND PULLDOWN MENUS.

One of the features that makes the Amstrad's windows less flexible than those on larger business machines, is the fact that the contents of a window which overlaps another are lost when the other window is used.

There are two new commands which allow the contents of windows to be saved and reloaded from RAM. This will allow the use of true pulldown menus, that can cover text, but not remove it.

EXAMPLE:-

```
NEW
10 MODE 1
20 FOR i=0.05 TO 1 STEP 0.05 : REM Draw grid on screen
30   MOVE 640*i,0 : DRAW 640*i,400
40   MOVE 0,400*i : DRAW 640,400*i
50 NEXT i
60 WHILE INKEY$="" : WEND : REM Wait for a key press
70 WINDOW#1, INT(RND(1)*19+1),INT(RND(0)*9+INT(RND(1)*5+17)),
   INT(RND(1)*14+1),INT(RND(0)*14+INT(RND(1)*10+5))
80 PEN#1, 2 : PAPER#1, 3
90 |SAVEW,1,1 : REM Save contents of window into RAM
100 CLS#1 : REM Clear window
110 WHILE INKEY$="" : REM Wait for 2nd key press
120 PRINT#1, "This is a window"
130 WEND
140 |LOADW,1,1 : REM restore window's contents
150 GOTO 60
```

The above program uses two new commands: |LOADW and |SAVEW. As you are probably aware, there are eight windows (0-7) which can be defined. The first parameter is the reference to a window. The second is the bank number.

```
|SAVEW, [window number], (bank] save window to bank
```

```
|LOADW, (window number], (bank] load window from bank
```

See the chapters in the user manual about windows for more details.

### 1.5.1 MORE WINDOWING.

A window of any size, even the whole screen, will fit into a single bank of expansion RAM. This is fine if your window is nearly a full screen or will vary in size like the above example. On the other hand if your window was defined as 10 x 10 in Mode 1, then the amount of memory needed to store this window would be less than 16K. In fact only 1600 bytes are needed (see below). Thus to use a whole bank would mean wasting over 14K of memory.

To deal with this problem, the RSX window command can take an extra parameter to define where you want the window's contents to reside in the RAM bank. In the 10 x 10 window you could place the data anywhere between 0 and 14783.

The command can be written:-

```
[SAVEW, [window number], [bank], [bank address]
```

```
[LOADW, [window number], [bank], [bank address]
```

The bank address is an address between 0 and 16383. The amount of data in bytes used to store a window needs to be taken away from the top value and this leaves the range between which the data can be stored. If you put the data at the bottom of the RAM bank, at address 0, then the memory from 1600 to 16383 is free for other windows or data arrays.

### HOW TO CALCULATE A WINDOW'S SIZE.

In order to have more than one window per bank, you need to know how much memory the window will take up. If the window will vary in size between two limits, use the higher of the two. Depending on which mode you are using, the figures are calculated as below.

In each case:      X1 is the left most x coordinate  
                      X2 is the right most x coordinate  
                      Y1 is the top y coordinate  
                      Y2 is the bottom y coordinate

MODE 0 SIZE=(X2-X1+1) \* 4 \* (Y2-Y1+1) \* 8

MODE 1 SIZE=(X2-X1+1) \* 2 \* (Y2-Y1+1) \* 8

MODE 2 SIZE=(X2-X1+1) \* (Y2-Y1+1) \* 8

The computer will give an error if the window is too large to fit in the space you have allotted for it. Also if the size is miscalculated the windows may overlap in the bank and cause strange effects.

### EXAMPLE 2:-

```
10 PEN 1 : PAPER 0 : MODE 1
20 size = 14 * 2 * 10 * 8
30 LOCATE 1, 13 : PRINT " 'n' for new window 'd' to remove window"
40 WINDOW 1, 14, 1, 10 : PAPER 3 : CLS
50 bankaddress=0 : level=0
60 PRINT#level, "Window",level
70 keypress$=LOWERS(INKEY$)
```

```
80 IF keypress$="n" THEN GOSUB 110
90 IF keypress$="d" THEN GOSUB 190
100 GOTO 60
110 IF level=7 THEN RETURN
120 level=level+1
130 WINDOW#level, 1+level*3, 14+level*3, 1+level*2, 10+level*2
140 |SAVEW,level,1,bankaddress
150 bankaddress=bankaddress+size
160 PEN#level,0 : PAPER#level, (level AND 1) + 1
170 CLS#level
180 RETURN
190 IF level=0 THEN RETURN
200 bankaddress=bankaddress - size
210 |LOADW,level,1,bankaddress
220 level=level-1
230 RETURN
```

The above program only uses one bank of RAM but all 8 windows are defined. The variable 'level' is used to stand for the level of windows and the variable 'bankaddress' points to the next free place in the bank RAM.

#### 1.6 ARRAYS, VARIABLES AND STRINGS.

There are two general purpose data moving commands to allow data from the program to be moved to and from the RAM pack.

These two commands are :

```
|SAVED, [bank], [Start location], [length], [bank address]
|LOADD, [bank], [Start location], [length], [bank address]
```

The first parameter references which bank you want to use. The start location is a memory address where there is some data. The amount of data is given as the length. Optionally a bank address can be given to allow more than one type of data to be stored in the RAM.

It is possible to save all kinds of data using these commands, but we will firstly discuss how to save simple numerical arrays these being the easiest to understand.

Say for example that your program deals with stock control of up to 60 items. You may have a string array containing the names and a numerical array containing the number of each item you have in stock.

This would use about 1K for the names and 300 bytes for the stock figures. However what if you update the stock value every week and you want to keep the last year of stock on record! Or even the last five years. Now the figures would take up about 15K or even 75K.

These could be comfortably stored on disc, or even tape for a year's stock, and the data read every time a calculation was needed, but you will probably agree that a long time would be spent waiting for reading the data each time a distribution is calculated for each item.

Obviously, it would be easier to load all the records into RAM, then access the data immediately:-

Instead of defining an array of dimensions 'stock(60,52)' taking over 15K of valuable RAM which could be used for programs, define an array 'stock(60)'. Read all the data from disc a week at a time, and store each week of data into bank RAM. To do this you need to know two things. One, where does the array lie in memory? and two, how many bytes is it necessary to save?

1) Where is an array stored?

The address of any variable can be quickly found using the '@' before a variable. For example, dimension the above array:-

```
DIM stock (60)
```

```
Now type: PRINT @stock(0)
```

The computer will reply by giving the memory address where the first element of the array is stored. Try:-

```
PRINT @stock(1)
```

The number returned will be five higher in value. This is the address of the second variable.

The '@' prefix will work in front of any variable. The first item of an array is obviously '@stock(0)'. If you are using multi-dimensional arrays, the first item is '@stock(0,0)' or '@stock(0,0,0,0)' depending on the number of dimensions.

2) How long is an array?

First of all, different types of array take different numbers of bytes per element. For real numbers, there are 5 bytes per element. Integer arrays take 2 bytes per element. String arrays are of variable length. And will be dealt with later.

Next, the number of dimensions and elements needs to be taken into account. Remember that elements start from 0. This means that an array of 'stock(60)' has 61 elements. Whether or not you prefer to use the 0 element is up to you, but if you forget it, there could be some unexplainable bugs appearing in your program. Once you know the real number of elements in every dimension, simply multiply together all the dimensions to find out the total number of elements in all dimensions.

```
For example: 'stock(60)' has a total of 61 elements.  
'stock(60,52)' has 61*53 elements = 3233 elements.  
'stock%(10,5,12)' has 11*6*13 elements = 858 in all.
```

To find the total memory, multiply the total number of elements by the amount of memory needed by each element.

```
For example: 'stock(60)' takes 61*5 = 305 bytes.  
'stock(60,52)' takes 3233*5 = 16165 bytes.  
'stock%(10,5,12)' takes 858*2 = 1716 bytes in all.
```

The array we are using is 304 bytes long, and starts at @stock(0). In a single bank of RAM we can store 305 bytes about 53 times. The bank address starts at 0 and goes up in steps of 305 bytes:-

```
0 305 610 915 1220 1525 etc.
```

We shall store week 1 at bank address 305, week 2 at address 610 and so on for all 52 weeks.

Data for test purposes could be written onto disc or tape by the program below. Once the test file is written, keep it for use while you are developing your program.

```
10 OPENOUT "stock.dat"
20 FOR week=1 TO 52
30   FOR item=1 TO 60
40     Print#90, INT(RND(1)*3000+100)
40     NEXT item
60 NEXT week
70 CLOSEOUT
80 END
```

Now type 'NEW' and enter the following program:-

```
10 DIM stock(60)
20 INPUT "read file (y/n)";ans$
30 IF LOWER$(ans$)="y" or LOWER$(ans$)="yes" THEN GOSUB 1000
40 REM rest of program ...
1000 REM subroutine to read data from disc.
1010 OPENIN "stock.dat"
1020 FOR week=1 TO 52
1030   FOR item=1 TO 60
1040     INPUT#9, stock(item)
1050     NEXT item
1060     |SAVED,4,@stock(0),61*5,week*305
1070 NEXT week
1080 CLOSEIN
1090 RETURN
```

The above program could be used to read the file from disc or tape. Once the file is in bank RAM, the contents will stay there for use until the computer is switched off, or some other data is put in that bank. This means that data need only be read once from disc, then the program can be rerun without losing the data. This could also be useful too if you wish to write a number of programs to use the same data.

Once the data is in memory, you can access each week's data simply by reloading the stock array. Add the section below to draw a bar graph for a given section.

```
100 MODE 2
110 LOCATE 1,1
120 INPUT "Which item to analyse",itemno
130 IF itemno < 1 OR itemno > 60 THEN 120
```

```

140 CLS : LOCATE 30,1
150 PRINT "Bar Chart For Item"; itemno
160 LOCATE 10,25
170 PRINT"Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec": REM 3 spaces
between each.
180 FOR loop=0 TO 4
190     LOCATE 1,24-loop*5
200     PRINT STR$(loop);"000"
210 NEXT loop
220 MOVE 60,328 : DRAW 60,0 : DRAW 61,0 : DRAW 61,368 :
    MOVE 640,24 : DRAW 48,24
230 FOR loop=1 TO 4
240     MOVE 48,loop*80+24 : DRAW 60,loop*80+24
250 NEXT loop
260 FOR week=1 TO 52
270     If week/2=week/2 THEN n=1 ELSE n=2
280     |LOADD,4,@stock(0),61*5,week*305
290     ycoord=(stock(itemno)/4000*320) AND 4092
300     FOR xcoord=1 TO 11 STEP n
310         MOVE 49 + xcoord + week*11,ycoord+26 :
            DRAW 49 + xcoord + week*11, 26
320     NEXT xcoord
330 NEXT week : GOTO 110

```

#### 1.6.1 MORE ARRAYS, VARIABLES AND STRINGS.

If you have a program that uses all the memory of the computer due to needing a large array, you can use the bank RAM for storing data without even dimensioning an array.

For example if you have a two dimensional array 'sales%(365,30)' to store the amounts of certain types of stock you sell for each day in one year. Even though you are using integers, the array uses over 22K of memory.

Instead of having the whole array in BASIC memory, each element can be accessed by using a subroutine to read out a value, and one to store a value.

```

10000 REM load 'store%' from bank memory using 'year' & 'type'.
10010 p= (year*31 + type)*2
10020 bank=1 : IF p >=16000 THEN p=p-16000 : bank=2
10030 |LOADD, bank, @store%, 2, p
10040 RETURN
11000 rem copy 'store%' to bank using 'year' & 'type'
11010 p= (year*31 + type)*2
11020 bank=1 : IF p >=16000 THEN p=p-16000 : bank=2
11030 |SAVED, bank, @store%,2, p
11040 RETURN

```

Two banks are used, 1 and 2, and the variables 'year' and 'type' are used to reference which element is required. On line 10030 and 11030, there are just 2 bytes moved to and from the bank RAM because we are using integers. The '\*2' in lines 10010 and 11010 reflect the fact that an integer is stored in two bytes. If real variables were used, 5 bytes would need to be used instead. Lines 10020 and 11020 decide whether the element is in the first bank or the second.

If the array is to be filled with data from tape or disc, there is no need to initially clear the values to nil. If you want all elements preset to zero then the easiest way is to save a blank screen into each bank at the start of the program:-

```
10 MODE 1 : PAPER 0 : CLS
20 |SAVES,1
30 |SAVES,2
```

### 1.6.2 STRING STORAGE.

The major obstacle in storing strings is that they can vary in length and can be stored anywhere in memory, including in a BASIC program. One method of storing string arrays is outlined below. However you may find an easier way to store strings than the one described below when you consider exactly what you want to do.

Suppose that you wanted to store 500 names, up to 20 characters long each. A bank is separated into units of memory 21 bytes each so that strings can be randomly accessed. In each 21 byte segment there is one string, and one byte to say how many letters there are in that string. That means that we will use a total of just over 10K. If we use the variable 'name' to specify the string we want then we can enter two subroutines; one to put a string from bank 1 into 'name\$' and one to store the contents of 'name\$' into RAM bank number 1:-

```
20000 REM assign 'name$' to string number 'name'
20010 b$=" " : REM 21 spaces
20020 |LOADD, 1, PEEK(@b$+1) + PEEK(@b$+2)*256, 21, name*21
20030 name$=MID$(b$, 2, ASC(b$)) : RETURN
21000 REM Store 'name$' in bank as element 'name'
21010 b$=" " : REM 21 spaces
21020 MID$(b$,1,21) = CHR$(LEN(name$)) + name$
21030 |SAVED, 1, PEEK(@b$+1) + PEEK(@b$+2)*256, 21, name*21
21040 RETURN
```

A dummy string b\$ is used to form the element before it is saved into RAM. The first character is set to the length of 'name\$'. The latter 20 characters are where the contents of 'name\$' are stored. Then 21 characters are copied into bank RAM. When the string is retrieved the characters are copied out and 'name\$' is set to the right length by looking at the first character.

String storage would come into its own if all the words were of the same length because there would be no wastage. For example a word quiz program using five, six and seven letter words. A bank of RAM could be used for each length of word. A loader program would set up the data into the RAM, then another could be CHAINED and use up to 36K of RAM for program.

A number array could also be stored in bank RAM to index the first letters and so aid the speed of access to a particular word.

### 1.7 ANIMATION AND PICTURE SHOWS.

We have seen how screens and windows can be stored and retrieved. Animation is the act of putting pictures on the screen quickly enough so that the eyes see something move. With 64K or 256K of memory whole screens can be stored away, then put on the screen to produce animation.

You may have noticed in section 4 that when a screen loads onto the screen, you can see each line appear. To illustrate, type in the following program:-

```
10 MODE 1
20 BORDER 0
30 FOR col=0 TO 3
40   INK col,0
50 NEXT col
60 FOR col=0 TO 3
70   PAPER col : CLS
80   |SAVES,col+1
90 NEXT col
100 INK 0,1 : INK 1,6 : INK 2,21 : INK 3,13
110 PEN 1 : PAPER 0
120 WHILE INKEY$=""
130   FOR screen=1 TO 4
140     |LOADS, screen
150     NEXT screen
160 WEND
170 END
```

The program saves four coloured screens into bank RAM, then loads then up in sequence. Unfortunately, the effect is a striped pattern.

In order to create animation which is easy on the eye the computer needs to create the screen display, then instantly display it.

Three new instructions that allow this to be done are:-

|LOW, |HIGH and |SWAP

Before the commands can be understood it is necessary to know how the Amstrad's screen can be used. The normal screen is located at 49152. However the Amstrad is capable of viewing a screen anywhere in memory in 16K blocks. The first block at 0 and the third block at 32768 are difficult to use for screen as the computer uses these as part of the BASIC interpreter. The block of memory at 16384 is free for use as long as BASICs HIMEM is lowered to below 16384. Using this, we have called the original screen the high screen and the new screen at 16384 is called the low screen. To go from one to the other just use:-

|LOW to set the low screen in action  
|HIGH to reset the high screen  
|SWAP to swap from low to high and vice-versa

Whenever the swap is made, the computer is told, and all further text and graphics appear on the selected screen.

To use this facility of swapping from one screen to another instantly, the screen and window commands can have an added parameter which tells the computer to load or save the data to and from the alternate screen.

The new forms can be written:-

```
|SAVES, [ bank ], [ swap ]
|LOADS, [ bank ], [ swap ]
|SAVEW, [ window number ], [ bank ], [ bank address ], [ swap ]
|LOADW, [ window number ], [ bank ], [ bank address ], [ swap ]
```

If the swap value is zero, by default, then the command will act on the screen that is presently being displayed. Alternatively, if the value is one the computer will load and save data from the screen which is not being displayed. When the work is done, the computer can swap screens and the effect is that the screen appears to change instantly.

In the above program type these lines:-

```
5 MEMORY 16383 : |HIGH
135 IF screen\2=screen/2 THEN t=TIME : WHILE TIME <t+20 : WEND
140 |LOADS, screen, 1 : |SWAP
```

Now that the computer can build up the screen while another is being displayed there is no pattern. The coloured screen appears to change instantly.

Due to the fact that the bank memory moves into the address space at 16K it takes longer for the transfer of screens to be made to the low screen than to the high screen. Hence line 135 delays the computer as it is to load the high screen. This means the time each screen is on the screen remains the same. Try removing line 135 to see the difference.

If a longer delay were to be put between line 140 and 150 you would get a picture show effect. Alternatively, you could select screens when a key is pressed.

On a small scale, a window could be defined and graphics could be rapidly displayed without resorting to swapping screens.

Note, that of less use is the fact that the contents of screens and windows can be saved from a screen which is not on display simply by adding a one for the swap parameter. For example if you want to load a series of screens from tape or disc, load them into the low (16K) screen. Messages generated by the tape system need not be switched off as the screen's contents will not be changed in the low memory screen.

```
10 LOAD "screen1",16384 : |SAVES, 3, 1
```

The above will load a screen then save it to bank 3. The screen the user sees can have something else on it.

### 1.8 ADVANCED PROGRAMMING.

This section introduces one new command and some other programming aspects which you may find useful.

The new command is:-

```
|ASKRAM, [ enquiry ], ( variable ]
```

The command allows certain constants to be found by the program you are writing. For example it can return the number of banks available to the program as this will change depending on whether you are using the 64K RAM pack or the 256K RAM pack. The 'enquiry' value is a number 1 to 3 which selects what you want to know. The answer is placed in an INTEGER variable defined by the second parameter.

```
1000 a%=0 : |ASKRAM, 1, @a% .. will assign a% to the amount of RAM
1100 a%=0 : |ASKRAM, 2, @a% .. will assign a% to the number of banks
1200 a%=0 : |ASKRAM, 3, @a% .. will set a% to 0 or 1 depending on
whether there is a problem with the RAM
```

The last command can be used to make sure the RAM is there and ready to use if in your programs you do not want to have to load the RSX loader first. It is possible to load just the RSX machine code on its own:-

```
20 MODE 1 : PRINT "Program Loading!"
30 I=HIMEM
40 MEMORY 9999
50 LOAD "rsx", 10000
60 I=I-( PEEK(10004) + PEEK(10005)*256+1)
70 POKE 10002, I-INT(I/256)*256
80 POKE 10003, INT(I/256)
90 PRINT CHR$(30);CHR$(21);
100 CALL 10000
110 PRINT CHR$(30);CHR$(6);
120 a%=0 : |ASKRAM, 3, @a%
130 IF a% THEN PRINT "RAM is faulty" : END
140 CLEAR : MEMORY PEEK(10002) + PEEK(10003)*256-1
160 CHAIN "part2"
```

The program above will load the RSX machine code and put it into memory. Nothing will be printed on the screen unless the RAM proves to be faulty or not even there! The program 'part2' would be the bulk of the program. Loading the program in two parts saves reloading the RSX code every time the program is run.

The code has to be loaded in at 10000 in memory before it is relocated for use. The 16 bit value in locations 10002 and 10003 is the place you want the code to be located at. Another 16 bit value in locations 10004 and 10005 contains the length of the code which is moved higher in memory. Nearly 1K of the program is only needed once - the relocation and the RAM test programs, and hence this part is not moved higher in RAM.

If you want to use user defined graphics then add the following lines:-

```
10 SYMBOL AFTER 256
150 SYMBOL AFTER 0
```

The value in line 150 will be different depending on how many user defined graphics you want.

In your program you may want to have a number of different styles of character set. After you issue a SYMBOL AFTER command, HIMEM is set just below the user defined graphics. Hence it is possible to use the |LOADD and |SAVED commands to move graphics to and from the graphics characters.

If you have a program that defines the character set, the definitions can be saved and loaded into bank RAM so that a program may have multiple character sets.

```
10 SYMBOL AFTER 0
20 chars=HIMEM+1
30 REM define symbols here
1000 SAVE "set1.grp", B, chars, 2048
```

This program will save your character set onto disc or tape.

On your final program you may wish to load a number of sets:-

```
10 SYMBOL AFTER 0
20 chars=HIMEM+1
30 LOAD "set1.grp", chars : |SAVED, 1, chars, 2048, 0
40 LOAD "set2.grp", chars : |SAVED, 1, chars, 2048, 2048
50 LOAD "set4.grp", chars : |SAVED, 1, chars, 2048, 4096
```

The reason the variable 'chars' is set up is because the value of HIMEM alters when the disc or tape is accessed.

During the program, a subroutine could be used to select a character set:-

```
1000 REM given the variable 'set', load the characters
1010 |LOADD, 1, chars, 2048,(set-1)*2048
1020 RETURN
```

Note that the variable 'set' is used. In the above loading sequence sets 1 to 3 will be valid. More or less could be added as it suits you.

All of this setting up can be done on the loader program, just once. When the program is subsequently run, there is no need to reload the bank RAM. The setting of chars can be found whenever needed by:-

```
200 CLEAR : SYMBOL AFTER 0 : chars = HIMEM+1
```

This will remove any disc buffers that have been set up and 'chars' will indeed point to the characters.

### 1.9 PEEKING AND POKING.

There are two commands which allow the memory in the banks to be viewed and changed byte by byte.

```
|POKE, [ bank ], [ bank address ], [ value ]  
|PEEK, [ bank ], [ bank address ], [ variable ]
```

|POKE works in a similar way to the original POKE. You need to supply a bank number in addition to the normal address and value. The bank address is in the range 0 to 16383 or 0 to 16K.

|PEEK is a command rather than the normal function. The bank and bank address are the same as for |POKE. To find out the value, you need to supply an integer variable in a similar way to the |ASKRAM command.

For example:-

```
10 value%=0  
20 |PEEK, 3, 12345, @value%  
30 PRINT value%
```

The above will read the byte from location 12345 in bank number 3. The @ character tells the RSX extension where the variable is in memory so that its contents can be changed to the byte required.

|PEEK and |POKE are not really commands for the beginner, in fact they have only been included for the more advanced programmer who wishes to use the bank RAM in his own way.

Another advanced command which has been included for the experienced programmer is |BANK.

```
|BANK, [ bank number ]
```

The command is followed by one parameter. If this parameter is not present, a zero is assumed. The bank referenced is mapped into the address space at 16K to 32K. A bank number of zero will map the original RAM back in, numbers 1 to the maximum bank number will map that bank into the address space. If a bank is mapped in, the computer will use the bank memory instead of the normal RAM. However, the screen will still be taken from the original RAM if |LOW was issued. The advantage of this is that the whole memory can be used for programming instead of having to set the top of memory to 16383. The disadvantage is that if the program is halted while the low screen is being displayed, the computer will write screen data into the BASIC program - causing chaos.

Make sure you are accustomed to using |BANK, |POKE and |PEEK before you risk creating a large program using them. Save the program frequently in case you make a mistake and lose your work.

1.10 PROGRAMMING WITHOUT RSX's.

With no RSX software the programmer can still access the memory from the RAM banks. To use the RAM yourself, some degree of understanding of the memory map of the Amstrad is necessary.

From both BASIC and machine code, the original block of memory from 16384 to 32767 CANNOT be used for program. Hence in BASIC, you need to set the top of memory to 16383. Machine code is free to use any memory that it can normally except the block mentioned.

The extra RAM is mapped into the addresses 16384 to 32767 in 16 banks. Once the bank is mapped in, you can do anything with the RAM you normally would. It is inadvisable to use the bank RAM for machine code because if you subsequently change the bank, the program disappears! Nevertheless, programs can be written to run in banks and indeed in the original 16K block that is banked out, but it is necessary to do the bank changing outside of this memory range. In BASIC it would be extremely difficult to use the banked RAM for extra programs, but not impossible, but we shall leave that possibility up to you!

The way that banks are selected is defined below:-

```
IN BASIC: Where 'bank' is the number of the bank to map in
OUT &7F00, 196+ (bank AND 3) + (bank AND 28)*2
NOTE: the bank numbers in this case START AT 0
```

For 64K expansions the banks are 0 to 3. On the 256K, bank numbers are 0 to 15.(bank 0-3 = 196-199:bank 4-7 = 204-207 etc.)

To reset the original bank:-

```
OUT &7F00, 192
IN MACHINE CODE: Where the bank number is in the accumulator. (A)

SELECT: PUSH BC          ; select bank A (save all registers except A
LD C,A                  ; and flags)
AND 3                    ; (bank AND 3) +
LD B,A
LD A,C
AND 28                   ;(bank AND 28)*2
ADD A,A
OR B
OR 196                   ;+ 196
LD BC,07F00H            ;BC=&7F00
OUT (C),A
POP BC
RET
```

Again the bank number in the accumulator starts at 0. To reset the original bank:-

```
RESET: PUSH BC          ;reset original memory
LD BC,07F00H           ;BC=&7F00
LD A,192
OUT (C),A
POP BC
RET
```

## 1.11 TECHNICAL DETAILS.

### 1.11.1 The Load Address.

The software which loads from tape is relocatable. However the areas of memory in which the program can go is limited to between 32768 and the top of memory. This is because the banked RAM appears in the block 16384 to 32767. (See previous chapter for explanation of why!) Below the 16K boundary, the RSX command table will no longer function. Hence, during relocation, the code is loaded at 10000 in memory and moves to a place higher in memory. Pressing <ENTER> while loading, will automatically select the highest location available. Alternatively you may wish to load the code to a lower address and reserve some space for your own programs.

### 1.11.2 Saving to Disc.

The software on the cassette is NOT protected. Hence to save it onto disc or even onto another tape at speed write 1 is a matter of loading the data into memory, then saving it.

- 1) Type '|TAPE' and press <ENTER> (for disc systems)
- 2) LOAD "bank"
- 3) MEMORY 9999
- 4) LOAD "rsx", 10000
- 5) Type '|DISC' or set SPEED WRITE as desired
- 6) SAVE "bank"
- 7) SAVE "rsx", B, 10000, 4000

### 1.11.3 INCREASING CP/M 2.2 TPA.

Boot up CP/M 2.2 that has CLOAD.COM on it. Copy the two programs NEWCPM.COM and OLDCPM.COM from cassette to disc by typing:-

```
CLOAD "NEWCPM.COM <enter>.(Repeat for OLDCPM.COM)
```

Then create a new CP/M system file by typing:-

```
A> MOVCPM 255 * <enter>
A> SAVE 34 NEWCPM.SYS <enter>
```

The new working disc now contains your increased TPA CP/M, invoked at any time by typing:-

```
A> NEWCPM
```

You can return to the original CP/M by typing:-

```
A> OLDCPM
```

This must be done before using some of the DFS utilities such as format etc. as these will only work with OLDCPM.

### 1.11.4 Commercial Program Compatibility.

The RAM expansion is compatible with the banked RAM supplied with the 6128. This means that a number of programs written for the 6128 will now work on the CPC464. The RSX software provided will work on the 6128 where the 256K pack will give 320K of banked RAM.

The bank switching software in its supplied state will only access 256K or 16 banks of memory. If you add more memory or have the CPC6128 with a 256K memory pack, the RSX software can be told to access a full 512K of banked memory (32 banks) by poking location 10006 with 1. See section 8 for explanation of how to load the RSX software on its own.

```
55 POKE 10006, 1 This line will do the trick!
```

If a commercial program fails to work on your CPC464 or CPC664 then try the suggestions below.

- 1) The software may be using the new firmware vector at &BD58. If this is the case, try running the RSX program before running your application program.

Some programs which will function correctly after the RSX software tape has been loaded in are Tasman's Tasword(R) word processor, Tasspell and Tasprint for the CPC6128. In conjunction with these, Campbell Systems' Masterfile 128 will provide a 64K filespace and interfaces with Tasman's software.

- 2) Some software, whether loaded from disc/tape or booted from a background ROM will check the ROM identity by using the firmware call &B915. There is one more command included in the RSX software on tape which will cause a CPC464 or 664 to emulate the ROM identity of the CPC6128:

Type |EMULATE and press <ENTER>

Any programs that call the ROM identity routine will now be informed that the computer is a CPC6128 and may now work correctly.

- 3) The software may use some features of the CPC6128 ROM which are unavailable on the CPC464 and CPC664 machines. In this instance, you may be able to get information on how the program can be altered to work on the CPC464 or 664 from the manufacturers of the program in question.

#### 1.11.5 Using CP/M 2.2.

CP/M 2.2 as supplied with all Amstrad computers will function as normal with the extra memory fitted. However if you create and use the NEWCPM program the TPA on CP/M 2.2 will be increased to 61K.

Programs of your own devising written under this operating system are free to use the extra memory. See section 1.10 for details of how to use the extra memory from machine code.

#### 1.12 ERROR MESSAGES.

While you are using the RSX software, there will be some occasions when the computer does not understand, or cannot carry out what you have instructed. The software may issue some error messages in addition to the normal messages that the computer will give. The errors and why they are likely to occur are outlined below:-

- |                     |   |
|---------------------|---|
| 1) Bad bank command | Given if you have given the wrong number of parameters or if a variable is not present where there should be one. |
| 2) Bank unavailable | You have tried to access a bank which is not present on your system.  |

- |                          |   |
|--------------------------|---|
| 3) Bad bank parameter    | You have referenced a bank which can never be fitted to the computer.   |
| 4) Bad bank address      | The address you have given is out of range: bank addresses range from 0 to 16383.   |
| 5) Value invalid         | The bank address may be too large for the block of data defined. The parameter for  ASKRAM is other than 1, 2 or 3. The size of a block to be saved is larger than 16K. |
| 6) Bad window definition | The window referenced in  SAVEW or  LOADW is above 7.   |

1.13 REFERENCE OF RSX COMMANDS.

All the additional commands are listed below as a reminder to their functions and syntax.

SCREENS.

|SAVES, [ bank ], [ swap ]  
|LOADS, [ bank ], [ swap ]

WINDOWS.

|SAVEW, [ window number], [ bank ], [ bank address ], [ swap ]  
|LOADW, [ window number], [ bank ], [ bank address ], [ swap ]

DATA BLOCKS.

|SAVED, [ bank ], [ start location ], [ length ], [ bank address ]  
|LOADD, [ bank ], [ start location ], [ length ], [ bank address ]

ANIMATION.

|LOW (Low screen)  
|HIGH (High screen)  
|SWAP (Alternate between High and Low screens)

OTHER.

|POKE, [ bank ], [ bank address ], [ value ]  
|PEEK, [ bank ], [ bank address ], [ variable ]  
|BANK, [ bank ]  
|ASKRAM, [ enquiry ], [ variable ]  
( [ enquiry ]: 1 = RAM, 2 = banks, 3 = error occurred?)

DEFINITIONS.

- |                  |  |
|------------------|--|
| [ bank ]         | Bank number 1-4 or 1-16 for 64K and 256K expansions. |
| [ bank address ] | Address within bank, 0 to 16383.                     |



Notes.

1. On power-up, code 0 is selected.
2. The VDU circuitry always reads from the original 64K (block 1), independently of the code.
3. If code 3 is used, reads from &4000 to &7FFF, on CPC 464 and 664 machines, will only return the correct data if the upper ROM is disabled. This is at variance with CPC 6128 operation, but is unlikely to be a significant difference.
4. If code 3 is used, addresses &4000 to &7FFF must not be used to run programs, they are intended for VDU or data access only.

Bits D5-D3, 64K Block Select Codes.

D5	D4	D3	BLOCK.
0	0	0	2 (ie, 'new' memory sub blocks came from block 2, as in CPC 6128.
0	0	1	3
0	1	0	4
0	1	1	5

Which of the above codes are relevant to your machine depends on total memory (see previous remarks).

Notes.

1. On power-up, code 0, 0, 0 is selected.
2. Bits D5, D4, D3 above 'count up' as blocks are selected. This may assist the programmer.
3. If 2x256K memory expanders are fitted to the machine, and option links are set appropriately, all patterns on D5-D3 can be used giving a maximum of 512K extra memory. (The memory that is used for a 256K silicon disc is correctly mapped to provide the extra 256K to give 512K total!)

1.15 CUSTOMIZING YOUR CP/M+ SYSTEM DISC.

Converting A 464 Keyboard Scan To That Of A 6128.

Some CP/M+ programs will not run correctly on the 464 computer because of the way the 6128 scans the keyboard.

The following program and instructions convert (fool) CP/M+ into thinking that it is running on a 6128.

1. Make a working copy of both sides of your system disc (sides 1 and 2).
  - a) Use your standard CP/M 2.2 System disc and type '|CPM'
  - b) For Single drive systems use DISCCOPY. For dual drives use COPYDISC. (Remember to do both sides).
  - c) Put your original system disc away in a safe place to keep as a backup in case your working disc is damaged.
2. Put 'BANK' and 'RSX' onto side 1 of the working disc.
  - a) Follow the instructions in the manual to get a copy onto disc. Alternatively, use 'FILECOPY BANK.BAS' and 'FILECOPY RSX.BIN' if you have transferred the software to disc already.
3. Reset the machine, then enter CP/M+. Only one drive is necessary but if you have two you may wish to disconnect the second drive because all the changes are to be made on the working disc and with a single drive the computer can use both sides.
  - a) RUN "BANK" <enter> in response to 'LOAD ADDRESS?'
  - b) Type in |EMULATE:|CPM <enter>
4. Type in the following, pressing <enter> after each line except where shown. Turn the disc over when the computer asks.

```
ED PATCH.ASM
i
ORG 100H
XRA A
STA 0FDEFH
JMP 0000
END
<control z>      DO NOT PRESS ENTER!
e
ED PROFILE.SUB
i
PATCH
<control z>      DO NOT PRESS ENTER!
e
(now insert the disc containing MAC.COM)
```

(continued over page...)

```
B:MAC PATCH
B:HEXCOM PATCH
ERA PATCH.HEX
ERA PATCH.SYM
ERA PATCH.PRN
ERA PATCH.ASM
```

5. It is possible to alter the CP/M+ disc to boot up without loading the BANK program first. Type the following if you want your system disc altered in this way:-

```
B:SAVE
B:SID C10CPM3.EMS
S1E0
C9
<control c> DO NOT PRESS ENTER!
C10CPM3.EMS
Y
100
6500
```

THE DISC WILL NOW BOOT UP WITHOUT BANK BEING RUN.