Answering the call to the bar

Auntie John spends some time RSXercising

HELLO to everyone, and especially to Fiona's father who sneaks a look at this magazine every month, possibly in the hope that I'll mention him. He's got no chance. The firmware call we are going to look at this month is the one that sets up what we in the machine code trade call an RSX.

Remember those three letters – mentioning them at a party will gain you vital street cred points. But remember, if you do mention them you might have to explain what they stand for, which can be a problem. Until now, that is, because I'm going to explain what they are and what to do with them.

RSX stands for Resident System eXtension. Aha! It all becomes clear, doesn't it. No, seriously, RSXs are no problem. In the simplest sense they are just machine code subroutines that you can call from Basic without having to remember nasty hex addresses. Every CPC has at least one RSX built in – try typing |BASIC. Oops. Did I mention that it resets the computer?

Other RSXs that are supplied with disc-based machines are |TAPE, |DISC and everybody's favourite |CPM. (Bar Completely Pretentious Mnemonics).

Using our own RSXs we can tack extra commands on to Basic. You have probably come across utility programs that add commands like |GPEN and |FRAME, well now you can add some of your own.

To set up the RSXs you must first give the computer a list of the commands you are going to use. This list must be set out in a particular way else nasty things will happen. (Crash! Oh bother did I save that first? What? I didn't? Oh dear. I think I'm going to cry. Boo-hoo-hoo).

The best way to explain how the command table is set up is to give an example, so let's look at what is needed to create an RSX called |GIBBLE.

```
; Sets up simple an RSX called |GIBBLE.

txt_output equ &bb5a ;Routine to print character.
log_ext    equ &bcd1 ;The firmware call that introduces
                     ;an RSX to the system.
org &4000

ld bc,command_table  ;The details required by the
ld hl,buffer         ;firmware call.
call log_ext
ret                  ;Return to Basic

.buffer ds 4         ;Four bytes needed by system.

.command_table
dw name_table        ;The addresses of the name table.
jp gibblecode

.name_table
db "GIBBL","E"+&80   ;The RSX name.
db 0                 ;A special value to indicate the
                     ;the end of the name table.
.gibblecode
ld a,7
call txt_output      ;print chr$(7)
ld a,42
call txt_output      ;print chr$(42)
ret

end
```

Listing I: The code for |GIBBLE

Void filled

And what's wrong with |GIBBLE? |GIBBLE is going to fill that desperate need that every CPC owner has to make his or her computer go "Bleep" and print an asterisk. Yes, yes, I know, how did you manage without it. But be patient my little artichokes and remember: Large machine code programs from tiny minds do grow. For the moment, though, take a look at Listing I.

You can see that there are two tables. The first is the command table, which starts with the address of the name table, and continues with jump instructions to the machine code subroutines. In this case we only have the one subroutine, and so only one jump.

The name table is where the RSX names themselves are stored. The last character in each name must have its most significant bit (msb) set – that means that bit seven must be a 1 – and the easiest way to do this is to add 128 to the byte (or &80 in hexadecimal). All RSX names must be in capitals because Basic will automatically convert keyboard commands to upper case before processing them. The end of the name table is indicated by a single zero.

Four bytes are needed by the operating system for its own evil purposes, and so we give it a "buffer" to play with.

Recipe for success

Now it was at about this time in the article that I was going to give you a listing of Fiona's recipe for Sweet and Sour Pork. But she wouldn't tell me it, so I'll just have to make it up. Here goes...

```
org the_kitchen

.start ld bowl,sugar
       ld bowl,vinegar
       call prepare_pork
       add bowl,pork
       ld oven,(bowl)

.cook  peek (oven)
       cp cooked
       jp nz,cook

.eat   ld mouth,(pork)
       hmmm...

.prepare_pork ld (grill),pork
              add salt_and_pepper
              ret
```

You could try using rabbits instead of pork. If you want some rabbits, I know just where to get some... Ow! Stop hitting me, Fiona. I'll tell everyone you read Jeffrey Archer books... OK, I apologise for saying nasty things about your rabbits. Please remind me to buy you some socks sometime.

Of prunes and parameters

Back to RSXs, and you may be interested to know that in their infinite wisdoms the designers of Arnold's interior decided to allow the user to pass parameters to and from RSXs and Basic. In English, this means you can pass and receive numbers and strings to and from the machine code subroutines. Thus you could write a subroutine called I GIBBLETWO which requires a parameter after it to determine the number of asterisks to be printed out. To pass parameters you eat them with some prunes beforehand. Oops, sorry. No. To pass parameters with RSXs you put commas before them like this: I GIBBLETWO,42.

The number of parameters passed is stored in the A register, and the parameters themselves are stored around an address given in the IX register. If you want to be exact about it, IX points to the address of the last entry, so you have to read them backwards.

If you supply the address of Basic variables, by using the @ symbol before the variable name, you can get the machine code to return a value to Basic.

Listing II is an example of this. It takes two integers, adds them and places the result in a predefined Basic integer variable. Integer variables take up two bytes of ram, hence the four instructions needed to transfer their contents to machine registers – first the least significant byte, then the most significant byte.

```
10 p%=10
20 |GIBBLETHREE,@p%,100,200
30 PRINT p%
40 END
```

Listing III: Using I GIBBLETHREE from Basic

Listing III is a small Basic program that shows I GIBBLETHREE in operation, assuming it has been set up by assembling the code with the machine code assembler you bought. You did buy one, didn't you?

The integer variable *must* have previously been assigned a value (as in line 10), and if all goes to plan, rather amazingly the number 300 will be printed.

Green stuff

So anyway, there was my mate Green in the pub, and this very pretty girl came along and sat beside him.

"Hello", says the girl. "Hello", says Green, sipping his pint.

"I'm thirsty", says the girl. "Oh", says Green, "why don't you buy a drink then?"

"I'd like to", says the girl, "but I haven't any money".

"What a pity", says Green, "why don't you get your boyfriend to buy you one?"

"I haven't got a boyfriend either", she simpers. "What? A pretty girl like you doesn't have a boyfriend?"

"No. And I'm not that pretty really".

"Hmm. No, I suppose you're not". says Green, downs his pint and leaves.

There is a moral to this story. Pity I can't remember it.

Passing strings

String variables can be passed to the machine code subroutine in a similar way to integer variables. However, because strings tend not to be fixed in length, a third byte is needed to tell us how many characters are involved. As we use only one byte for this purpose, strings are limited to a maximum length of 255 characters – 255 being the largest number that one byte can hold. If you attempted to alter this length byte from within the subroutine, Basic would collapse in around you as it loses track of all its variables – a

```
; Machine code to set up an RSX called |GIBBLETHREE
; It adds integers sent to it. Wow.

txt_output equ &bb5a ;Routine to print character.
log_ext equ    &bcd1 ;Firmware call that introduces an
                     ;RSX the system.
org &4000

ld bc,command_table  ;The details required by the
ld hl,buffer         ;firmware call.
call log_ext
ret                  ;Return to Basic.

 buffer ds 4          ;Four bytes needed by system.

 .command_table
 dw name_table       ;The addresses of the name table.
 jp gibblemc

 .name_table
 db "GIBBLETHRE"
 db "E"+&80          ;The RSX name.
 db 0                 ;A special value to indicate the
                      ;the end of the name table.
 .gibblemc
 ;
 ; RSX that takes (only) two integers
 ; adds them and returns them into a
 ; Basic variable. Heaven knows why.
 ;
 cp 3                ;Return to Basic if not three
 jr nz,errortrap     ;parameters passed.
 ;
 ld a,(ix+0)         ;Read the
 ld l,a              ;second
 ld a,(ix+1)         ;integer
 ld h,a              ;parameter.
 ;
 ld a,(ix+2)         ;Read the
 ld e,a              ;first
 ld a,(ix+3)         ;integer
 ld d,a              ;parameter.
 ;
 add hl,de           ;Do the addition.
 ;
 ld a,(ix+4)         ;Get the address of
 ld e,a              ;Basic p% variable
 ld a,(ix+5)         ;into DE.
 ld d,a
 ;
 ld a,l              ;Poke the
 ld (de),a           ;result into
 inc de              ;the address
 ld a,h              ;held in DE.
 ld (de),a
 ret

 .errortrap
 ld a,42             ;Print a * using my very
 call txt_output     ;favourite firmware call.
 ret

 end
```

Listing II: The code for I GIBBLETHREE

nasty thing to happen to anyone, rather like being very drunk, I imagine.

A slightly useful subroutine would be one that encoded any Ascii string passed to it. The easiest way to do this would be to XOR each part of the string with a special code-letter or, better still, a series of code-letters. The way XOR works means that to decode the string you encode it again.

Listing IV sets up an RSX that will encode the string with a code word, here converted into its Ascii equivalent – the five numbers following the XORs. Once the string has been encoded it can be sent to disc or tape just like any other string – but if PRINTed to screen or printer will cause a terrible mess.

To decode the string, pass it through the routine once more and all will be back to normal. Listing V shows you how to use |ENCODE from Basic assuming, once again, that you have previously assembled and CALLed the machine code (use CALL &4000 to log the RSX on).

Such fun. You now have a genuinely useful RSX that will protect your Ascii files from prying pryers.

And so we come to the last programming example. The Basic function TIME returns the time in yonkettes of a second – useful for timing things like eggs or contractions – or rabbits (Ow!). However, Basic doesn't allow you to reset this timer, so it is necessary to use another Basic variable to keep track. But wait! Listing VI will set up an RSX called |ZEROTIME. I'll leave you to study it in peace...

Well, by now you should be getting to grips with RSXs. Adding them to your computer is nowhere near as painful as it may seem at first, and as you build up a library of the more useful routines, you will find your Basic programs become more and more adventurous.

An obvious thing for you to do would be to combine all the RSXs presented above into one program. All you need to do is to add the commands and names to the various tables, and then supply the code for each routine. Easy.

Happy birthday to me, Happy birthday to me, Happy birthday dear Auntie, Happy birthday to me. I'm old.

See you next time. Bye.

```
; Sets up simple an RSX called |ENCODE that
; when given a string returns a coded version.

log_ext equ &bcd1    ;Firmware call that introduces
                     ;an RSX to the system.
org &4000

ld bc,command_table  ;The details required by the
ld hl,buffer         ;firmware call.
call log_ext
ret                  ;Return to Basic.

buffer ds 4          ;Four bytes needed by system.

.command_table
dw name_table        ;The addresses of the name table.
jp encode

.name_table
db "ENCOD","E"+&80   ;The RSX name.
db 0                 ;A special value to indicate the
                     ;the end of the name table.
.encode
cp 1                 ;if not just one string present
ret nz               ;we return to Basic immediately.
ld a,(ix+0)
ld l,a               ;get the address of
ld a,(ix+1)          ;the string's "descriptor"
ld h,a               ;into HL.
ld a,(hl)            ;The length of the string.
cp 0                 ;If the length is zero
ret z                ;return to Basic
ld b,a               ;Store length in B.
inc hl
ld a,(hl)            ;Low byte of string address.
ld e,a
inc hl
ld a,(hl)
ld d,a               ;High byte of string address.
;
; Now the subroutine to encode/decode a string.
; DE contains the address of the string.
; B contains the length.
;
.loop
ld a,(de)
xor 70
xor 105
xor 111
xor 110              ;Extremely secret code word
xor 97               ;(use your own, obviously).
ld (de),a
inc de
djnz loop            ;Repeat for entire length.
ret

end
```

Listing IV: Routine to encode a string

```
10 INPUT"Enter a string";a$
20 |ENCODE,@a$
30 PRINT"Encoded string = ";a$
40 PRINT"Press a key to decode..."
50 CALL &BB18 'wait for a keypress
60 |ENCODE,@a$
70 PRINT"Decoded string = ";a$
80 END
```

Listing V: Using |ENCODE from Basic

```
; Program to set up an RSX called |ZEROTIME
; that resets the internal timer.

set_timer equ &bd10  ;Firmware call that sets the timer.
log_ext   equ &bcd1  ;Firmware call that introduces
                     ;an RSX to the system.
org &4000

ld bc,command_table  ;The details required by the
ld hl,buffer         ;firmware call.
call log_ext
ret

buffer ds 4          ;Four bytes needed by system.

.command_table
dw name_table        ;The addresses of the name table.
jp timercode

.name_table
db "ZEROTIM","E"+&80 ;The RSX name.
db 0                 ;A special value to indicate the
                     ;the end of the name table.
.timercode
;
; A routine to reset the internal timer
; to zero. Did I mention my middle name
; is Terence?
;
ld hl,0
ld de,0
call set_timer
ret

end
```

Listing VI: Routine to reset the internal timer