10 SuperBASIC

10.1 Introduction

This chapter is somewhat different to the rest of this book because it is soley devoted to SuperBASIC. There can be considerable advantages in using BASIC, namely that the entire language structure, editor etc. are already there. Creating specialised BASIC functions and procedures for particular applications can therefore be a worthwhile task.

Before we can start looking at how to add utilities to SuperBASIC, it is necessary to have a thorough understanding of the structures for storing variables and other names.

Whenever a name is entered into a BASIC program from the keyboard, an entry is created for it in memory. All that is stored in the program itself is a pointer to the relevant entry. Such a method of variable allocation is very space efficient if a particular variable is used over and over again. There is then virtually no extra storage burden for storing long variable names which make programs much easier to understand.

There are four distinct areas in which information is stored. These are the *name table*, the *name list*, the *variable value area* and the *arithmetic stack*. Unlike many of the less sophisticated computers, the entire working area is liable to move without warning! For this reason, all references in BASIC are carried out relative to register A6 which always points to the base of the area.

10.2 The name table

SuperBASIC handles variables and arrays in a a rather indirect way. As mentioned in section 10.1, every variable reference within the BASIC program file is simply a pointer to an entry in the *name table*. This table itself does not contain the variable values, but holds pointers to a name in the *name list* and a value in the *variables values area*. Each entry in the name table is eight bytes long and conforms to the following:

1st word	code describing name usage
2nd word	pointer to the name in the name list
3rd long word	pointer to the value

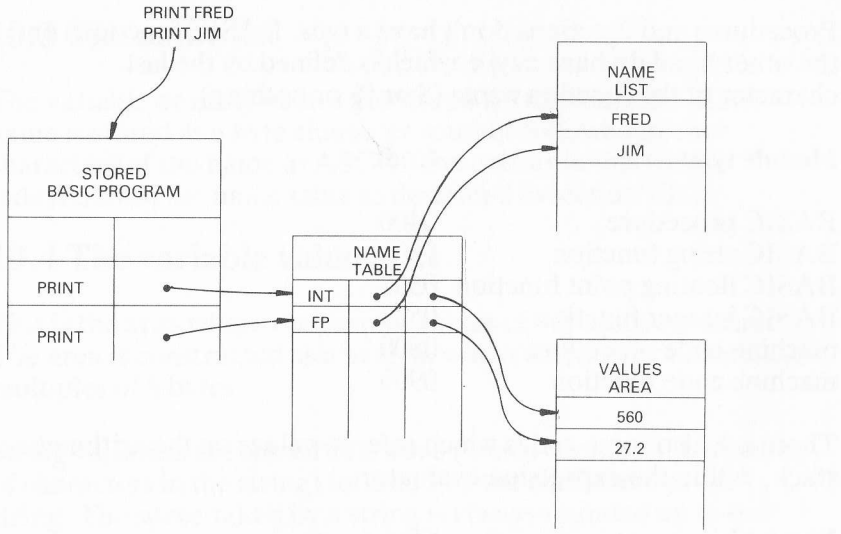

Figure 10.1 – Illustration of BASIC variable storage

Type code

This is the first word in the name table block. There are a variety of different types which are:

Variable	Code
unset string	0001
unset floating point number	0002
unset integer	0003
string variable	0201
floating point number	0202
integer	0203
REPEAT loop index	0602
FOR loop index	0702

Array	Code
substring array	0300 (internal only)
string array	0301
floating point array	0302
integer array	0303

Procedures and functions don't have a *type*. BASIC functions on the other hand do have a *type* which is defined by the last character of the function name ($ or % or nothing).

Module type	Code
BASIC procedure	0400
BASIC string function	0501
BASIC floating point function	0502
BASIC integer function	0503
machine code procedure	0800
machine code function	0900

There are also some codes which refer to values on the arithmetic stack, within the expression evaluator.

Expression	Code
string	0101
floating point	0102
integer	0103

The name pointer

This is the second word in the name table entry. It contains the offset of the name in the name list, unless the entry is the value of an expression. In this case, it contains −1. Alternatively, if the entry is simply a copy of another entry in the table, a pointer to that entry is included.

Value pointer

This is a long word and is the third argument in the name table. If the value of this long word is negative then the *value* is undefined. If positive, this is the offset from the base of the VV (variable value) area in the case of variables, or the offset for the array descriptor in the case of arrays. The most significant word is the line number of the DEF PROC or DEF FN line of a BASIC procedure or function. The pointer is the absolute address of the entry point of a machine code procedure or function. Note that the value pointer is not defined for an internal name entry referring to a value on the arithmetic stack.

10.3 The name list

The variable or array names are stored in this name list. Each name is stored as a byte character counter followed by the characters of the name in ASCII. The entries in this list are indexed from the name table as described in section 10.2.

10.4 The variable value area

This is the area where the current values of variables are stored. The area is constructed as a heap in which entries are allocated as multiples of 8 bytes.

String variables are stored as a word (which contains the number of characters in the string) followed by the characters of the string. The space taken by a string is always rounded up to the nearest even number so that the string doesn't end in the middle of a word boundary.

Floating point numbers are stored as a two byte exponent and a four byte mantissa. See section 8.5.4 for more details of floating point number storage.

An integer is stored as a word.

Array storage

The array descriptor which is indexed from the name table is in the following format (see illustration in figure 10.2); it commences with a long word offset for the array values from the base of the variable values area. This is followed by a word which contains the number of dimensions. A pair of words come after this for each dimension. The first word is the maximum index, and the second word is the index multiplier for this dimension.

Floating point and integer arrays are stored in an entirely regular manner. A floating point array takes up 6 bytes per element and an integer array occupies 2 bytes per element.

String arrays are inherently not so regular because of the variable lengths of entries. The string array is stored as an array of characters. The zeroth element of the final dimension is actually a

word containing the maximum length of the string. The zeroth dimension is always rounded up to the nearest even value (so that each string starts on a word boundary).

Internal string slicing operations produce *substrings*. These are regular arrays of characters. The base to which indexing is referred is one rather than zero. It should be noted that the descriptor of a substring of a substring of a string array is incorrect in current release versions of the Basic.

10.5 Examples of variable storage

Floating point variables (in hex)

stored as:	representing:
0000 0000 0000	0.0
0801 4000 0000	1.0
0800 8000 0000	−1.0
0804 5000 0000	10.0

see also section 8.5.4

Floating point arrays

base, 2, 3, 3, 2, 1	DIM A(3,2)

String storage (numbers in decimal)

4; 65, 66, 67, 68	'ABCD'

String arrays

base, 2, 3, 12, 10, 1	DIM A$(3,10)

```
4;65,66,67,68, x, x, x, x, x, x  "ABCD"
9;49,50,51,52,53,54,55,56,57, x  "123456789"
0; x, x, x, x, x, x, x, x, x, x  ""
1;32, x, x, x, x, x, x, x, x, x  " "
```

Substring array

base, 1, 3, 1	A$(0,1 TO 3) as above
65, 66, 67	'ABC'

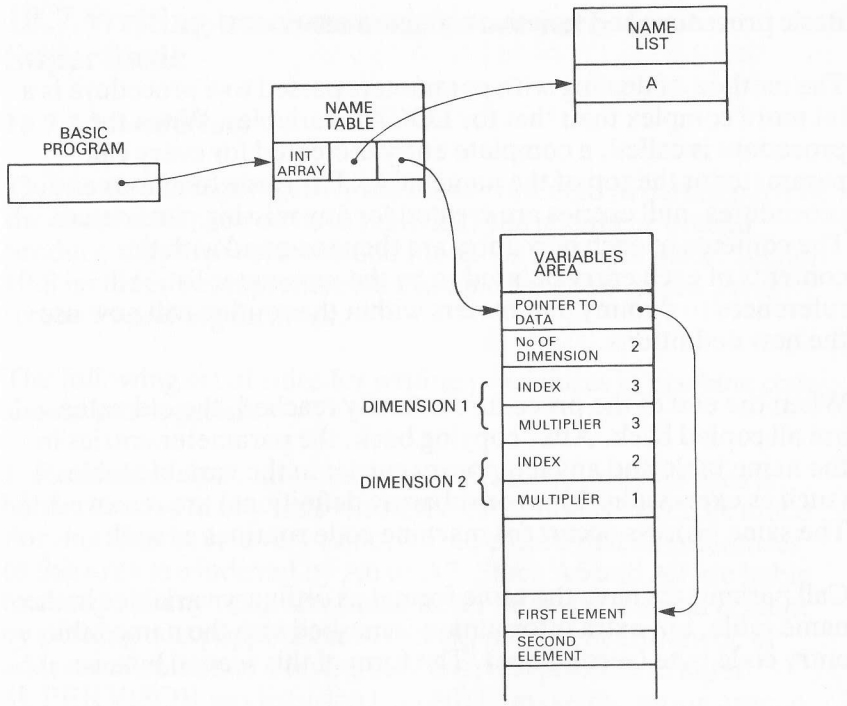

Figure 10.2 – Array storage in BASIC

10.6 Sending variables between procedures, functions and Basic

Before we go into more details on how to write procedures, it is necessary to understand how parameters are transferred for different types of operations.

Local variables

Local variables within procedures or functions are the most easy to understand because virtually nothing happens. When a LOCAL statement is encountered, the entry for the given variable is copied to the top of the name table. The contents of the original entry are replaced by a new empty entry. This new empty entry is then used within the procedure. At the end of the procedure, the LOCAL variable is no longer required. The old entry is therefore copied back from the top of the stack, deleting the LOCAL entry in the process.

Basic procedure and function call parameters

The method of dealing with parameters passed to a procedure is a bit more complex than that for LOCAL variables. When the procedure is called, a complete entry is created for every call parameter at the top of the name table. For Basic functions and procedures, null entries are created for any missing parameters. The contents of each new entry are then swapped with the contents of each entry pointed to by the parameter list. All references to dummy parameters within the routine will now use the new definitions.

When the end of the procedure is finally reached, the old values are all copied back. After copying back, the parameter entries in the name table and any temporary entries in the variable table (such as expression values or subarray definitions) are removed. The same process occurs for machine code routines as well.

Call parameters have the same format as ordinary variables in the name table, but extra information is masked into the name table entry code byte (second one). The form of this second byte is:

hsssvvvv, where h is set if the parameter is preceded by #

sss is the following separator

000	for no separator
001	for ,
010	for ;
011	for \
100	for !
101	for TO

vvvv is the *type*

0000	null
0001	string
0010	floating point
0011	integer

10.7 Writing machine code procedures for SuperBasic

10.7.1 Introduction

This section brings together the ideas presented in earlier parts of the chapter. Once you have read this, you should be able to produce simple procedures and functions for SuperBasic. Section 10.8 includes a comprehensive example to set up screen 0 and screen 1 handling facilities.

The following set of rules for writing procedures in machine code should prove useful:

1. Remember at all times that the SuperBasic program area is liable to move at any time whilst the execution is in USER mode. For this reason, it is very important to ensure that all references to this area are indexed by A6 or A7. Since A6 and A7 are liable to change at any time, they must never be saved, altered (except by pushing and popping the A7 stack) or used in arithmetic or address calculations. Having said this, it is possible to enter SUPERVISOR mode (using trap#0) to make the action atomic. If this is done, ensure that neither A6 nor USP (A7) are altered before entering supervisor mode, and that they are fully restored before going back to user mode and returning.

2. Do not use more than 128 bytes on the USER stack.

3. D0 must be returned as a long word error code.

4. On entry to the routine, A3 points to the first entry in the name table and A5 points to the last entry. The number of parameters in the name table can be calculated as (A5-A3) / 8.

5. D1 to D7 and A0 to A5 can all be changed at will.

10.7.2 Creating name table entries

Before a procedure can become fully accessible from Basic, it must first of all be correctly linked into the name table. Basic procedures in ROM are automatically linked in (see appendix C). It is as simple to link in procedures which are in RAM. Generally, these procedures are in the resident procedure area. This is so

that they cannot be deleted except by re-booting the QL. If a procedure did get removed once it has been put into the name table, any attempt to access it would probably lead to a system crash.

To link in the procedure in RAM, A1 should be set to point to the start of the procedure definition list, and a call should be made to BP.INIT (vectored utility $110). A suitable piece of code would be:

```
LEA     PROCDEF (PC),A1
MOVE.W  $110,A2
JSR     (A2)
```

PROCDEF is a procedure definition list in the standard format:

word number of procedures

then for each procedure:

word pointer to routine
byte length of name of procedure
characters of procedure name

word 0

word number of functions

then for each function:

word pointer to routine
byte length of name of function
characters of function name

word 0 for end of definition list

The pointers to the routines are all relative to the address at which the pointer is located.

Internal table space is reserved by the number of functions or procedures. If the average length of the procedure or function names exceeds 7 then this should be (total number of characters + number of functions or procedures + 7)/8.

BP.INIT preserves all registers except for A1, and uses a maximum of 48 bytes on the USER stack.

10.7.3 The arithmetic stack

The arithmetic package routines require some working storage area to hold the arguments and results from expression evaluation. The arithmetic stack provides this storage area. This stack is normally referenced by A1. It can be used by machine code to evaluate call or return parameters, or simply as general working space.

The routines to be discussed in section 10.7.4 which get arguments will reserve enough space on the arithmetic stack for their own purposes. If a user designed procedure wishes to use space on the arithmetic stack, this space should be reserved by a call to BV.CHRIX (vectored utility $11A). The required number of bytes are passed in D1.L. Note that the arithmetic stack may move when this call is made. The arithmetic stack pointer must therefore be saved (in BV.RIP(A6)) if there is anything on the stack. BV.RIP should then be moved back to the stack pointer afterwards.

The arithmetic stack is automatically tidied up by the function and procedure interface routines after procedures and after errors in functions.

10.7.4 Getting procedure arguments

An indeterminate number of procedure arguments can be obtained by using one of the routines CA.GTINT, CA.GTFP, CA.GTSTR and CA.GTLIN which will get integer words, floating point numbers, strings and long integers respectively.

Arguments are pointed to by A3 (the first parameter in the name table) and A5 (the last parameter in the name table). The routines will convert parameters between these two limits into the required form, putting the results onto the arithmetic stack. The first argument is at the lowest address pointed to by (A6,A1.L). D3 contains the number of arguments which were fetched on return. Note that the # and separator flags in the name table entries are destroyed. Arguments can be processed one at a time by extracting the # and separator flags. A5 should then be set to 8 bytes greater than A3 and the appropriate routine should be called to get one argument.

The routines can be found in the vectored utilities section:

$112	CA.GTINT	gets word integers
$114	CA.GTFP	get floating point numbers
$116	CA.GTSTR	get strings
$118	CA.GTLIN	get long word integers

10.7.5 Returning function values

Values can be returned from functions by putting the value on the arithmetic stack with (A6,A1.L) pointing to it. The value of A1 must be stored in the Basic variables area in BV.RIP (A6) which is located at ($58(A6)). D4 should be set to contain the type of the return argument. Possible return arguments are:

D4=1	string argument
D4=2	floating point argument
D4=3	integer (word) argument

Note that long word integers must be converted to floating point format. A good return from a function must only be made if the stack is in a *tidy* state. This means that the return argument must be at the top of the stack, and the function must not leave anything below it.

10.7.6 Returning parameter values

There is a special utility routine called BP.LET which will let you return values through the parameter list of a procedure or function. In order to do this, the value to be returned should be on the arithmetic stack (BV.RIP (A6) should be set to this). The value must be in the correct form required by the calling parameter, ie. string, integer or floating point format. The appropriate parameter entry in the name table should be pointed to by A3. Routine BP.LET (vector $120) should then be called. See section 8.5.3 for details of the call.

Note that the value will be lost on return if the passed parameter was an expression and not a named variable. If strings are being returned, care must be taken to ensure that the byte counter for the string comes on a word boundary. This can be done by *padding* out odd length strings by one byte at the end.

10.8 Toggling between screen 0 and 1

On the QL, there are two different areas of memory which can be used to generate the screen display. The *normal* screen is called screen 0 and sits in the bottom 32K bytes of RAM below the system variables. The second screen is located in the 32K bytes of RAM above the first. Unfortunately, the system variables have been defined to sit on top of screen 0. This means that they are actually inside of screen 1!

However, all is not lost. There is still the possibility of using most of screen 1 (apart from the 5K allocated to system variables). To do this, screen 0 alone can be used to display the top fifth of the screen. Screen 1 is then flipped into the display under interrupt control every frame. The lower half of the screen can then be used for fast graphics. Plotting occurs on one screen whilst being displayed on the other, and vice versa.

You may now be wondering what screen toggling has to do with adding Procdures and Functions to Basic. In this section, a complete screen handling utility package is presented. Three new Procedures are added to Basic. SCR0 turns on screen 0, SCR1 turns on screen 1 (the top half being masked) and SCRA sets the *auto-toggle* mode so that pressing CTRL-F5 will flip from one screen to another. So that programs can tell which screen is being displayed at any time, a Function called SCRNUM is provided.

10.8.1 Two screen support program

```
*
* Two screen support utility
*
MT.ALCHP EQU      $18
MT.LPOLL EQU      $1C
BP.INIT  EQU      $110
SV_SCRST EQU      $33
SV_MCSTA EQU      $34
SV_SER1C EQU      $98
SV_SER2C EQU      $9C
SV_MDRUN EQU      $EE
MC_STAT  EQU      $18063
MC..SCRN EQU      $7
BV_RIP   EQU      $58
```

```
*
* First initialisE the extra BASIC procedures
*
        MOVE.W   BP.INIT,A2
        LEA      PROC_DEF(PC),A1
        JSR      (A2)
*
* Now reserve the second screen
*
        MOVEQ    #MT.ALCHP,D0
        MOVE.L   #$6BE0,D1
        TRAP     #1
        TST.L    D0
        BNE.S    EXIT
*
* Now link in the interrupt handler
*
        LEA      INT_SERVE(PC),A1  entry address
        MOVEQ    #MT.LPOLL,D0
        LEA      INT_LINK(PC),A0   link address
        MOVE.L   A1,4(A0)          put entry in link
        TRAP     #1                link in
EXIT
        RTS
*
* BASIC procedure definitions
*
PROC_DEF
        DC.W     3
        DC.W     SCRA-*
        DC.B     4,'SCRA'
        DC.W     SCR0-*
        DC.B     4,'SCR0'
        DC.W     SCR1-*
        DC.B     4,'SCR1'
        DC.W     0
        DC.W     1
        DC.W     SCRNUM-*
        DC.B     6,'SCRNUM'
        DC.W     0
*
* Set screen auto toggle
*
SCRA
        LEA      SCR_AUTO(PC),A1
        ST       (A1)              set auto toggle
        BRA.S    EXIT_OK
```



```
*
* Set to screen 0
*
SCR0
        LEA      SCR_AUTO(PC),A1
        CLR.W    (A1)              screen 0, no auto toggle
        BRA.S    EXIT_OK
*
* Set to screen 1
*
SCR1
        LEA      SCR_AUTO(PC),A1
        MOVE.W   #$00FF,(A1)       screen 1, no auto toggle
        BRA.S    EXIT_OK
*
* Find which screen is displayed
*
SCRNUM
        MOVE.L   BV_RIP(A6),A1     must be room for 2 bytes
        SUBQ     #2,A1
        MOVE.L   A1,BV_RIP(A6)
        MOVEQ    #1,D1             return 1
        AND.B    SCR_1(PC),D1      ... or 0
        MOVE.W   D1,(A6,A1.L)
        MOVEQ    #3,D4             type is integer
EXIT_OK
        MOVEQ    #0,D0
        RTS
*
*
* Interrupt service
*
SCR_AUTO DS.B    1
SCR_1    DS.B    1
INT_LINK DS.L    2                 link and entry address
INT_BASE EQU     INT_LINK-8        link table (A3 on entry)
INT_SERVE
        MOVE.B   SV_MCSTA(A6),D1 reset to screen 0
        MOVE.B   D1,MC_STAT
*
        TST.B    SCR_1-INT_BASE(A3) is screen 1 required
        BEQ.S    INT_TOGGLE        ... no, give up
        TST.L    SV_SER1C(A6)      is serial 1 open?
        BNE.S    EXIT_INT          ... yes, timings wrong
        TST.L    SV_SER2C(A6)      is serial 2 open?
        BNE.S    EXIT_INT          ... yes, timings wrong
        TST.B    SV_MDRUN(A6)      is a microdrive running?
        BNE.S    EXIT_INT          ... yes, timings wrong
*
        MOVE.W   #1360,D0          wait for system variables
        DBRA     D0,*              ... to pass
```

```
*
        BSET    #MC..SCRN,D1
        MOVE.B  D1,MC_STAT
*
INT_TOGGLE
        TST.B   SCR_AUTO-INT_BASE(A3) CTRL F5 toggling?
        BEQ.S   EXIT_INT        ... no
        TST.B   SV_SCRST(A6)    has CTRL F5 been pressed
        BEQ.S   EXIT_INT        ... no
        SF      SV_SCRST(A6)
        NOT.B   SCR_1-INT_BASE(A3) switch screen
EXIT_INT
        RTS
```

10.8.2 How the program works

The first part of the program is concerned with initialisation. BP.INIT is used to set up the name table entries for the procedures and functions (see section 10.7.2 for more information). Manager trap MT.ALCHP is then called to reserve memory in the common heap area ($6BE0 bytes). This is so that no other allocations in the common heap area will overwrite screen 1. Finally, the interrupt handler is linked into the 50/60 Hz interrupt linked list. It is this service routine which toggles between screen 0 and screen 1, carefully controlling the position at which the swithover occurs.

The code for each of the procedures is very simple indeed. Each sets the contents of the byte at SCR_AUTO. The interrupt server then reads from this location to determine whether it should toggle the screen or set screen 0 or 1.

SCRNUM is a function which returns the current screen number as 1 or 0. It returns this value to Basic using the mechanism as described in section 10.7.5. Note that the value returned is set as an integer type.

The Interrupt service routine checks the contents of SCR_AUTO to determine what action (if any) should be taken. Screen 1 can only be switched in after the system variables have be passed (otherwise the system variable display would appear on the screen). The timing for this period is done in software using DBRA D0,* which puts the 68008 into an internal timing loop. This will only work if the high priority interrupts from the serial ports or microdrive are not being used. A check is therefore made. If any of these interrupts are likely to occur, screen 1 is not paged in.

10.9 Implementing BPUT# as a procedure

This example sets up a BASIC procedure which allows individual bytes to be sent to a channel (such as a microdrive).

```
*
* Single byte put utility
*
MT.LPOLL EQU    $1C
CA.GTINT EQU    $112
IO.SBYTE EQU    $05
ERR.BP   EQU    -15
ERR.NO   EQU    -6
BP.INIT  EQU    $110
BV_CHBAS EQU    $30
BV_CHP   EQU    $34
*
* First initialise the BASIC procedure BPUT#n,data
*
        MOVE.W  BP.INIT,A2
        LEA     PROC_DEF(PC),A1
        JSR     (A2)
        MOVEQ   #0,D0
        RTS
*
* BASIC procedure definition
*
PROC_DEF
        DC.W    1
        DC.W    BPUT-*
        DC.B    4,'BPUT'
        DC.W    0
        DC.W    0
        DC.W    0
*
* BPUT routine
*
BPUT
        BSR.S   CHANNEL
        MOVE.W  CA.GTINIT,A2
        JSR     (A2)
        SUBQ.W  #1,D3           just one argument?
        BNE.S   ERR_BP
        MOVE.B  1(A6,A1.L),D1
        MOVEQ   #IO.SBYTE,D0    output byte to channel
        MOVEQ   #-1,D3
        TRAP    #3
        RTS
```

```
*
ERR_BP
        MOVEQ   #ERR.BP,D0      bad parameters
        RTS
*
* Set default or given channel
* Call parameters   :
*
* A3 and A5 standard pointers to name table for parameters
*
* Return parameters :
*
* D6 pointer to channel table
* A0 channel ID
*
CHANNEL
        MOVEQ   #1,D6           default is channel #1
        CMPA.L  A3,A5           any parameters?
        BEQ.S   CHAN_LOOK       ... no
*
        BTST    #7,1(A6,A3.L)   has 1st parameter a hash?
        BEQ.S   CHAN_LOOK       ... no
*
        MOVE.L  A5,-(A7)        save top parameter pointer
        MOVE.L  A3,A5           set new top
        ADDQ    #8,A5           to 8 bytes above bottom
        MOVE.L  A5,-(A7)        (when done, it's new base)
        MOVE.W  CA.GTINT,A2     get an integer
        JSR     (A2)
        MOVE.L  (A7)+,A3        restore parameter pointers
        MOVE.L  (A7)+,A5        (doesn't alter cond codes)
        BNE.S   CHAN_EXIT       was it OK?
        MOVE.W  0(A6,A1.L),D6   replace default with D6
*
CHAN_LOOK
        MULU    #$28,D6         point D6 to channel table
        ADD.L   BV_CHBAS(A6),D6
        CMP.L   BV_CHP(A6),D6   is it within the table?
        BHI.S   ERR_NO          ... no
        MOVE.L  0(A6,D6.L),A0   set channel ID
        MOVEQ   #0,D0           no error
CHAN_EXIT
        RTS
ERR_NO
        MOVEQ   #ERR.NO,D0      channel not open
        RTS
```


10.9.1 How the program works

The first part of the program is concerned with initialisation. BP.INIT is used to set up the name table entry for the procedure BPUT.

Routine CHANNEL then looks for a channel number preceded by a '#'. If found, the channel ID is returned in A0 with a pointer to the channel table in D6. The byte to be sent to this channel is then obtained using CA.GTINT. Finally, the byte is output using IO.SBYTE.

Another example of linking a function into Basic can be found in 8.5.5.