8 LINKING INTO SuperBASIC

There are two major topics within this chapter. One topic is that of SuperBASIC machine code commands (e.g., CALL, SBYTES, etc.), and the second topic is concerned with interfacing machine code procedures and functions into SuperBASIC in order to extend the language. We will, for the present time, look at the theory only. The four chapters in Part 3 present plenty of real examples on how to do things in practice.

8.1 SuperBASIC machine code commands

A total of eight procedures and functions exist within SuperBASIC to enable the assembly language programmer to load, save, and execute machine code routines. Six of the commands are commonly used (i.e., CALL, EXEC, SEXEC, LBYTES, SBYTES, and RESPR). The remaining two (i.e., PEEK and POKE) exist mainly, it is suggested, for completeness. None of the example programs in Part 3 need these last two commands. Peeking and poking, as a discipline on the QL, is discouraged (at least be the author!!). An assembler package should always be used to create, modify, and generate machine code programs.

Although it may be possible to group the commands into sub-groups, and then look at each sub-group, we will deal with the commands in alphabetical order. This is consistent with the format in previous chapters, and will make future references easier tasks.

CALL

This is a procedure which will accept an address followed by a maximum of 13 parameters. The general format of the command is:

CALL addr, p1, p2, ... , pn

The address 'addr' is the start address of the machine code to be executed, and the machine code must exist in the resident procedure area (see RESPR). If any parameters are supplied they will be put into the 68000 data and address registers D1 to D7 and A0 to A5, respectively. For example, if one parameter only is supplied, it will be placed into data register D1. If eight parameters are supplied, the first seven will go into data registers D1 to D7 respectively, and the eighth will go into address register A0.

No parameters can be returned from a CALL statement. SuperBASIC will report the error found in register D0 on return from a called routine. If no errors occurred in the machine code program, it is advisable to set D0.L to zero before returning. The CALL procedure is particularly useful for calling the initialisation routine of a machine code package to extend SuperBASIC.

EXEC

This is a procedure which is used to invoke an executable code file, or a sequence of such files. Each executable program will become a separate QDOS job, and will execute within the transient program area. Two forms of the command exist:

EXEC job1, job2, job3, ... , jobn

EXEC_W job1, job2, job3, ... , jobn

The first form (i.e., EXEC) will invoke the job and return immediately to SuperBASIC. The second form (i.e., EXEC_W) will invoke the jobs, but not return to SuperBASIC until all the jobs have finished execution. All the example programs in Chapters 9 and 10 are designed to be executable code programs and, therefore, invoked by this EXEC command.

LBYTES

Any file can be loaded into memory, starting at a specified address, using this procedure. The general format of the command is:

LBYTES device_file, addr

The procedure is most obviously used to load machine code extensions to SuperBASIC into the resident procedure area, prior to initialisation and subsequent use. The parameter 'addr' is the base address in memory where the code is to be loaded. Note that this procedure will only load the file, it will not attempt to run the bytes loaded.

PEEK

This is a function which will return the contents of the specified memory location. There are three forms, allowing a choice of data size:

value = PEEK addr

value = PEEK_W addr

value = PEEK_L addr

The first of these will return a byte (8-bit) value. The specified address may be any address desired. The last two will return a word (16-bit) and a long-word (32-bit) respectively. Both of these latter two forms require 'addr' to be an even address.

POKE

This is a procedure which will load the specified memory location with the specified data. There are three forms, allowing a choice of data size:

POKE addr, data

POKE_W addr, data

POKE_L addr, data

The first of these will load a byte (8-bit) value. The specified address may be any address desired. The last two will load a word (16-bit) and a long-word (32-bit) respectively. Both of these latter two forms require 'addr' to be an even address. For word operations the memory locations 'addr' and 'addr+1' will be used, with the most significant byte going into 'addr'. Long-word operations will use locations 'addr' to 'addr+3', again with the most significant byte going into 'addr'.

RESPR

This is a function which is used to reserve space in the resident procedure area of memory. It has the general form:

base = RESPR (space)

The function requires one parameter, specifying the amount of memory required. If there is insufficient room in memory to perform the required allocation, the message 'out of memory' will be displayed and the function will abort. The function will also abort, with the error message 'not complete', if any executable programs are in the process of running. On a successful completion, the function will return the base address of the memory area allocated.

Space allocated by RESPR, in the resident procedure area, cannot be reclaimed (normally) without re-booting the machine. It follows that, if two or more RESPR functions are executed without intermediate re-boots, the function will keep extending the resident procedure area until memory is exhausted. This enables more than one block of RAM to be allocated, with each block having its own base address. The blocks will not overlap.

SBYTES

This procedure is the inverse of LBYTES, discussed earlier. The general form of the command is:

SBYTES device_file, addr, length

The area of memory from 'addr' to 'addr+length' will be saved on the specified device, and in the specified file.

SEXEC

This is a procedure which will save an area of memory, on a specified device, in a form suitable for use with the SuperBASIC EXEC (or EXEC_W) procedure. The general form of the statement is:

SEXEC device_file, addr, length, dataspace

It is assumed that the area of memory from 'addr' to 'addr+length' holds a machine code program. The parameter 'dataspace' should specify the size of the data memory (including stack areas) that will be required by the program when it is executed. Note that you do not save, therefore, run-time workspace. You save the code only, and specify the size of run-time workspace required.

8.2 Interfacing to SuperBASIC

When we talk of interfacing to SuperBASIC, we are really talking about extending the SuperBASIC language by the addition of suitably written machine code procedures and functions. Actually making SuperBASIC realise that extra routines (commands or statements) are available is an almost trivial task (see Sec.8.5). Collecting parameters, manipulating SuperBASIC variables and channels, and returning results may not be so easy (though they could not be called onerous).

In order to interface to SuperBASIC we need to know about such things as the run-time context of our routine (i.e., what pointers exist in which address registers when entry is made to our routine by SuperBASIC), and the structure of the SuperBASIC work area. The rest of this chapter is dedicated to the theory of interfacing to SuperBASIC. Chapters 11 and 12 contain some real examples.

8.3 The SuperBASIC environment

It was shown in Chapter 3 that the whole of the SuperBASIC area can shift dynamically. This being the case, there must be a pointer somewhere that informs SuperBASIC routines where the start of the area is. All references to the actual program, the variable tables, and so on

will then be made relative to that (running) pointer. In practice, the pointer is address register A6.

On entry to your machine code extension, register A6 will point to the base of the SuperBASIC work area pointer table (see Fig.8.1). This pointer table holds vital information about the relative positions of distinctly separate SuperBASIC work areas. For example, one of these areas is the program itself! Each pointer is a long-word and its value is itself relative to register A6. In other words, the pointers do not hold absolute location values for the work areas. They contain, instead, relative offsets to the work areas. Let us look at each of these areas in more detail.

BUFFER AREA

This is exactly what the name suggests; a buffer area. A minimum of 128 bytes exist, and routines are free to use the area as they wish. The true length of this area can be determined by subtracting the buffer area base pointer from the pointer held in '$08(A6)', e.g.:

```
:
move.l    bv_bfbas(a6),d1     ;get base pointer
move.l    $08(a6),d2          ;get pointer past top
subq.l    #1,d2               ;(top)
sub.l     d1,d2               ;length now in D2
:
```

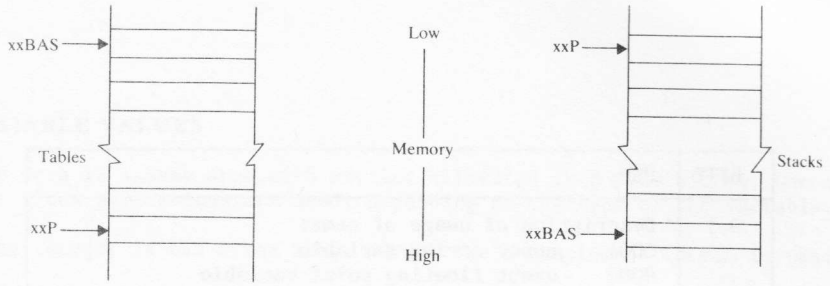
A running pointer (BV_BFP) exists for use as required, within this buffer area. It will usually point to the first free location.

SUPERBASIC PROGRAM AREA

This is simply the area where the current SuperBASIC program is held. Pointers BV_PFBAS and BV_PFP point to the base and the byte beyond the end of this area respectively.

NAME TABLE

This table is vitally important. Each reference to a variable in a SuperBASIC program is, internally, a pointer to an entry in the name table. Each entry is eight bytes long, as shown in Fig.8.1.



Super BASIC variable pointer table

Table pointers	$00 (A6)	Buffer base	BV_BFBAS
	$04 (A6)	Buffer running pointer	BV_BFP
	$08 (A6)		
	$10 (A6)	Super BASIC program base	BV_PFBAS
	$14 (A6)	Top of Super BASIC program	BV_PFP
	$18 (A6)	Name table base	BV_NTBAS
	$1C (A6)	Top of name table	BV_NTP
	$20 (A6)	Name list base	BV_NLBAS
	$24 (A6)	Top of name list	BV_NLP
	$28 (A6)	Variable values base	BV_VVBAS
	$2C (A6)	Top of variable values	BV_VVP
	$30 (A6)	Channel table base	BV_CHBAS
	$34 (A6)	Top of channel table	BV_CHP
Stack pointers	$58 (A6)	Top of arithmetic stack	BV_RIP
	$5C (A6)	Base of arithmetic stack	BV_RIBAS
	$60 (A6)	Top of system stack	BV_SSP
	$64 (A6)	Base of system stack	BV_SSBAS

★★ Each pointer is itself relative to A6

Figure 8.1 SuperBASIC pointer table and work areas

BYTE	USE
0,1	Description of usage of name: 0001 unset string variable 0002 unset floating point variable 0003 unset integer variable 0201 string variable 0202 floating point variable 0203 integer variable 0300 array substring (internal only) 0301 string array 0302 floating point array 0303 integer array 0400 SuperBASIC procedure 0501 SuperBASIC string function 0502 SuperBASIC floating point function 0503 SuperBASIC integer function 0602 REPEAT loop index variable 0702 FOR loop index variable 0800 Machine code procedure 0900 Machine code function
2,3	Name pointer. This is an offset to the name in the name list. A value of -1 signifies that the entry is an expression value for the expression evaluator (in such cases the bytes 0,1 of the name table entry will be 01xx). If the entry is a copy of another entry, the name pointer will be a pointer to that entry.
4-7	Value pointer (long-word). This is an offset to the value of the entry (in the case of variables) or to the descriptor (in the case of arrays), from the base of the variable values area. If the pointer is negative, a value has not been assigned yet. It is also used for the absolute pointer to a procedure or function, or the line number of a SuperBASIC procedure or function.

Note that procedures (SuperBASIC or machine code), and machine code functions, have no 'type'. SuperBASIC functions do have a type, which is defined by the last character of the function name (i.e., none, $, or %).

NAME LIST

This is a list of the actual names themselves. Each name is stored as a byte of data holding the character count, followed by the characters of the name.

VARIABLE VALUES

This area is a heap area with entries allocated in 8-byte blocks. One or more block allocations are used, depending on the type of the variable.

1. An integer is two bytes long. Normal two's complement format is used.

2. A floating point number is stored as a 2-byte offset exponent followed by a 4-byte mantissa. Examples of floating byte codes are:

Exponent	Mantissa		Value
0000	0000	0000	0.0
0801	4000	0000	1.0
0800	8000	0000	-1.0
0804	5000	0000	10.0

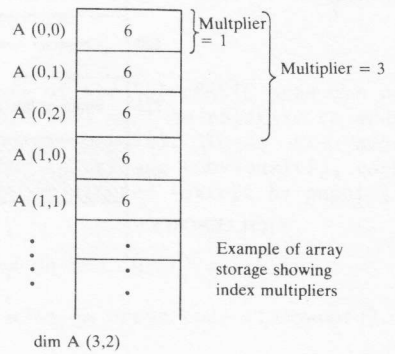
3. A string will be stored as a word of data, containing the byte count of the string, followed by the string itself. The actual space taken by a string will be rounded to the nearest even number.

4. Array descriptors have a long-word header that is the offset of the array values from the base of the variable values area. Next, there are the number of dimensions of the array (stored as a word of data), and then there are pairs of data words for each dimension. The first 'dimension word' will specify the maximum index for that dimension, and the second word will be the index multiplier for that dimension. The figure opposite shows the layout of a floating point array. Note that if the dimensioning statement in SuperBASIC was, for example,

    DIM A(3,2)

    the descriptor would have the format:

    base, 2, 3, 3, 2, 1



The storage of floating point arrays and integer arrays is entirely regular. Floating point array elements are six bytes long, and integer array elements are two bytes long.

A string array is regular (i.e., it is an array of standard strings) except for element zero of the last dimension. The last dimension of a string array defines the maximum length of the string. It will always be rounded up to the nearest even number.

CHANNEL TABLE

SuperBASIC channel numbers (#n) are pointers to the channel table. This table is a set of sub-tables, one sub-table for each open channel (see Fig.8.2). A sub-table is 40 ($28) bytes long. The sub-table entry for #n would therefore be located at:

BV_CHBAS(A6) + (n x CH.LENCH)

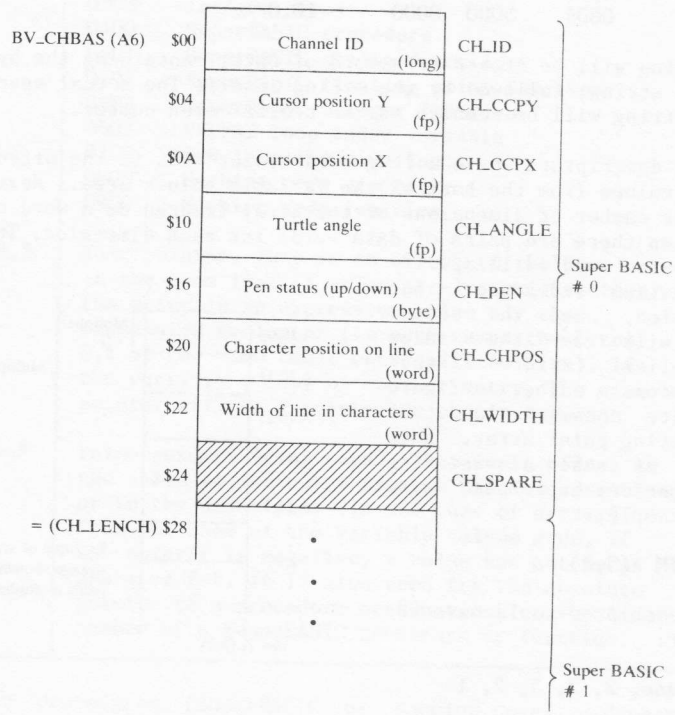


Figure 8.2 SuperBASIC channel definition table

ARITHMETIC STACK

The arithmetic stack is the working area for expression evaluation. It is also used in evaluating call, and return, parameters. It may also be used as a general working area. Remember that stacks grow downwards (i.e., from high memory to lower memory).

The SuperBASIC interface mechanism automatically tidies up the arithmetic stack after procedure calls, and after errors in functions. On the other hand, a good return from a function must be made with a tidy stack. The return argument must be on the top of the stack (i.e., at the low stack memory end), and no other data must be left below the argument (i.e., at a physically higher stack memory address). See Sec.8.11 also.

SYSTEM STACK

This is the area used when any reference is made (implied or otherwise) to the address register A7. It is, for example, the stack area that will be used to store the return address for a 68000 'BSR' instruction.

8.4 Implementing machine code procedures

There is a simple set of rules that must be obeyed when writing machine code extensions to the SuperBASIC language.

1. It must be remembered that the whole of the SuperBASIC area can move, and therefore all references to this area must be relative to address register A6 (or A7 in the case of stacks). These two address registers should never be saved for future use (obviously!), used in arithmetic or address calculations, or altered (except by pushing and popping on the A7 stack).

2. Not more than 128 bytes may be used on the 'user' stack.

3. Data register D0 must be returned with an error code (long-word).

On entry to the routine, SuperBASIC will have set up, in addition to the above registers, address registers A3 and A5. Any parameters passed over to the routine will have entries created for them in the name table (see Sections 8.3 and 8.6). Register A3 will point to the first parameter entry, and register A5 will point to the end of the last entry (remember that A3 and A5 are relative to A6). The number of parameters passed over will, therefore, be equal to '(A5-A3)/8'. Clearly if A5 equals A3, no parameters were supplied.

Registers D1 to D7, and A0 to A5 may be treated as volatile within the routine itself (though it would be very unwise to destroy A3 or A5 too early!).

8.5 Creating name table entries

A simple mechanism exists for the initialization of RAM based extensions to SuperBASIC. The extensions should be loaded into the resident procedure area by using the SuperBASIC commands RESPR and LBYTES. For simplicity, and the sake of clean source documentation, it is convenient to have the initialization code at the very beginning of the machine code (though this is not essential).

INITIALIZATION CODE

The code, and its corresponding table, for the initialization of extension routines is very simple. Address register A1 should be set to the start of the procedure definition table, and a call made to the utility routine BP.INIT (vector $110):

```
        :
        lea     proc_def(pc),a1     ;get table address.
        move.w  $110,a2             ;prepare for BP.INIT and
        jsr     (a2)                ;call it.
        moveq   #0,d0               ;no error.
        rts                         ;finish.
```

More than one extension can exist, and the format of the table is therfore:

Data size	Use
word	number of procedures
	(for each procedure):
word	- pointer to routine
byte	- length of procedure name
characters	- name of procedure
word	0
word	number of functions
	(for each function):
word	- pointer to routine
byte	- length of function name
characters	- name of function
word	0

The number of procedures and/or functions is used purely to reserve internal table space. If the average length of the names exceeds seven, this number needs to be:

(total no. of characters + number of routines + 7)/8

The pointers to the routines are relative to the address of the pointer. All registers (except A1) are preserved by the BP.INIT utility. No more than 48 bytes are used on the 'user' stack.

8.6 Parameter initialization

When a machine code procedure or function is called, an entry will exist in the name table for each parameter passed over. At the end of execution, the parameter entries in the name table will be removed, together with any temporary entries made in the various tables (e.g., the variable values table).

Name table entries, for call parameters, have the various separators (e.g., hash, comma, semi-colon, and so on) masked into the least significant byte of the description code (i.e., byte 1, see Sec.8.3). The full form of this byte is given by the bit pattern:

bit	7	6	5	4	3	2	1	0
	h	s	s	s	v	v	v	v

Bit 7 ('h') will be set if the parameter was preceded by a hash (#). Bits 4 to 6 specify the separator that follows the parameter:

bit 654	separator
000	no separator
001	comma (,)
010	semi-colon (;)
011	back-slash
100	exclamation (!)
101	the keyword TO

Bits 0 to 3 specify the 'type' of the parameter, as follows:

bit 3210	type
0000	null
0001	string
0010	floating point
0011	integer

Note that if an expression was passed over as an actual parameter to the

call, the name pointer in the name table (bytes 2,3 - see Sec.8.3) will be set negative.

8.7 Obtaining arguments

A set of four SuperBASIC utility routines exist which will read an indeterminate number of identical 'type' parameters. They are accessed in the same way as other utilities (i.e., through vectors), and they have the following vector addresses:

$0112	CA.GTINT	Get integers (16-bit)
$0114	CA.GTFP	Get floating point numbers (6 byte)
$0116	CA.GTSTR	Get strings (2+n bytes if even, 3+n bytes if odd)
$0118	CA.GTLIN	Get long integers (32-bit)

If a parameter list contains different 'types', it will be necessary to make multiple calls to appropriate routines in order to collect all of the parameters.

On entry, the utilties require A3 and A5 to be set to the base and the top of the name table parameter entry list respectively. The results will be placed on the arithmetic stack, with the first argument at the lowest physical address pointed to by 'A6,A1.L'. The number of arguments fetched will be returned in register D3 as a word. Register D0 will contain the error code (the status flags will be set also, according to the error return), and registers D1, D2, D4, D6, A0, and A2 are affected. Registers A3 and A5 will be preserved and, therefore, the address register A3 will need to be updated if a further call is required to one of these vectors.

Parameter arguments may, of course, be processed one at a time under the programmer's own control. To do this, you would extract the hash (#) and separator codes, set A5 to be eight bytes above A3, and then call the appropriate utility. It is clearly important, if you adopt this method, to be careful how you manipulate registers A3 and A5 so as not to miss any parameters, nor overrun.

8.8 Returning function values

A function value is returned simply by putting the value on the arithmetic stack (pointed to by 'A6,A1.L'). The value of register A1 must also be in BV_RIP(A6); see Fig.8.1. The 'type' of the return must be placed into register D4 (1=string, 2=floating point, and 3=integer). A long integer (32-bit) must be converted and returned as a floating point value.

8.9 Returning parameter values

Values may be returned through the parameter list of a procedure (or function!) call. As with function value returns, the parameter value should be on the arithmetic stack with BV_RIP(A6) set accordingly. The return value must be integer, string, or floating point, to match the calling parameter. Register A3 must be set to the corresponding parameter entry in the name table, and finally, the utility BP.LET (vector $0120) called.

On returning from BP.LET, register D0 will be set to the error code, and registers D1 to D3 and A0 to A2 will be affected. If the actual parameter initially passed over was in the form of an expression, the return assignment will be made but the value lost.

8.10 Returning strings

The word addressing limitations of the 68000 processor cause some problems when returning strings. Care must be exercised to ensure that the byte counter for the string comes on a word boundary (i.e., an even memory address). In practice, this is achieved by padding out odd length strings by a blank at the end of the string. Note that, for example, a string of length 3, and a string of length 4, will both occupy six bytes on the stack.

8.11 Special note on arithmetic stack handling

The built-in utility routines to fetch arguments will reserve enough space on the arithmetic stack for their own purposes. If a machine code extension requires autonomous use of the arithmetic stack, it also should reserve space by calling the utility BV.CHRIX (vector $011A). The number of bytes required should be in register D1 (as a long-word) on entry. The utility will affect registers D0 and D3.

It is possible that the arithmetic stack will move when this operation is performed. If the procedure has anything on the arithmetic stack before BV.CHRIX is called, the stack pointer (usually register A1) should be saved in BV_RIP(A6), and then retrieved from BV_RIP(A6) afterwards.

8.12 TRAP #4

There is a special TRAP for the SuperBASIC command interpreter, that may be required for use also by machine code procedures. The particular trap call is TRAP #4, and it has the effect of making the addresses passed to the I/O traps (see Chapters 5 and 6) relative to register A6. The call should be made before each and every TRAP #2, or TRAP #3 call, because its effect is cancelled by the latter calls.

For TRAP #2, register A6 is added to A0 on entry. For TRAP #3,

register A6 is added to A1 on entry, but removed from it on exit. Note that the TRAP #4 call will not be cancelled by a TRAP #3 call which fails under the error 'not open'.