User's Manual
SBasic68k Compiler Version 3.4

by
Karl E. Lunt

4 March 2007

Table of Contents

1. Disclaimer

I am releasing the executable for the SBasic68k (SB68k) compiler, all supporting library and include files, assorted test cases, and this document as freeware. Feel free to use SBasic68k for whatever non-commercial application seems appropriate.

The SBasic68k compiler, with or without its attendant support files, is freeware and in the public domain. You may not charge for the sale or distribution of SBasic68k or its distribution files. If you distribute SBasic68k to others, please include this manual in its present form, complete with this disclaimer.

I make no warranty, representation, or guarantee regarding the suitability of SBasic68k for any particular purpose, nor do I assume any liability arising out of the application or use of SBasic68k, and I disclaim any and all liability, including without limitation consequential or incidental damages.

You, as the user, take all responsibility for direct and/or consequential damages of any kind that might arise from using SBasic68k in any fashion.

I do not warrant that SBasic68k is free of bugs. If you find what you think is a bug, kindly let me know what it is IN DETAIL, and I'll certainly consider fixing it in a later release, if there ever is such.

I developed SBasic68k as a tool for working with the Motorola 68xxx family of MCUs. If you use SBasic68k for developing robotics (or other) application code and find it useful, fine. If you don't find it suitable in some fashion, then don't use it.

(4 Mar 2007) I released the first version of this manual in April of 2001. In January 2007, Mike Lozano (mlozano71@satx.rr.com) was kind enough to convert my original DOS text file to Word format and send me the results of his effort. I added further editing and formatting to create what you are reading now. Thanks, Mike, for your work in the conversion and for helping me get back to SB.

Karl Lunt
116 173rd St., SW
Bothell, WA 98012

2. Introduction

This manual describes the use of the SBasic68k (SB68k) compiler. SB68k is a PC-based cross-compiler for a subset of the Basic language. Source files containing SB68k statements are compiled into a source file of assembly language for the target machine. Subsequent assembly of that file yields an executable file for the target machine.

SB68k creates code for a 68xxx target. The compiler's output is compatible with the Motorola as32 assembler, available on the Motorola web site or from my own web site at: **http://www.seanet.com/~karllunt**.

The SBasic68k compiler was written in Borland C (version 4.52), and is compiled as a 32-bit DOS standard app. My SB68k design is loosely based on a small Basic interpreter developed by Herbert Schildt and presented in his excellent book, "The Art of C." (Osborne McGraw-Hill, Berkeley, CA 94710; ISBN 0-07-881691-2)

3. History of SBasic68k

Version 3.4 fixed what I hope is the last bug in the comparison code, again spotted by Doug Kelly of the SRS. The error occurred when comparing the result of a math operation on two variables with a constant. I also noticed that version 3.3 was incorrectly printing out version 3.2 in the header of the listing file; this version now prints out version 3.4.

Version 3.3 fixed a few bugs reported by Doug Kelly of the Seattle Robotics Society. The copy library was not being included when invoked, and the library code had an error in it. Also fixed an obscure error when adding two literals on one side of a test.

Version 3.2 fixed a bug in the 2D array index calculations. It also includes updated library files minmx68k.lib and inkey68k.lib, both contributed by Doug Kelly of the SRS.

Version 3.1 fixed a number of bugs in the 3.0 release. My thanks to Doug Kelly of the SRS for his determined efforts in using the original release, and in helping me weed out these bugs.

Version 3.0 was adapted from the 68hc11 SB68k compiler, version 2.7; this is the baseline release of the 68k version.

Besides reworking the code generator to support 68000 assembler output, I also added a number of features. The biggest change is with variable size; all SB68k variables and arrays use 32-bit integers, rather than the 16-bit integers in the 68hc11 version.

I added word-sized (16-bit) versions of several statements and functions, including POKEW and PEEKW. I also added support for 2-dimension arrays.

I included shift and roll functions similar to RROLL() and LSHFT() that work on bytes, rather than bits. These functions, with names such as RROLLB(), will speed up byte manipulations during I/O.

4. Invocation

You execute SB68k by entering the command:

```
sb68k  infile  <options>
```

where 'infile' contains the path to the source file you wish to compile. SB68k writes its output, the assembly language source for the target, to the standard output, which is normally the screen.

You can redirect the output file to another file by entering the command:

```
sb68k  infile  <options>  >outfile
```

where 'outfile' is the path to an output file to hold the created source lines. This file should carry an .asm extension, as that is expected by the as32 assembler used with SB68k.

If SB68k did not detect any errors in your source file, its output file should assemble correctly with the as32 assembler. If SB68k detected errors, it inserts error notices in the output file. These are almost guaranteed to generate numerous errors if the resulting output file is assembled.

SB68k supports several command-line options, used to control the addresses of key elements in the final program. These options will be explained in detail below.

Upon completion, SB68k returns an errorlevel that can be used to determine success or failure of the compilation. If SB68k successfully compiled the source program, it returns an errorlevel of 0. If SB68k detected one or more errors during compilation, it returns an errorlevel of 1.

5. Command-line options

You can control the placement of variables, code space, and stack space in the target executable by means of SB68k command-line options. If you supply any options in your command line, they must follow the name of the SB68k source
file. Refer to the above section on executing SB68k.

You can control where SB68k places the start of its RAM variables by using the /v option. The format of this option is:

```
/vxxxxxxxx
```

where xxxxxxxx is an eight-digit hexadecimal address that marks the start of the variable space. SB68k assigns this address to the assembler label VARBEG; if you do not use the /v option, SB68k assigns a default value of $3000 to VARBEG.

You can control where SB68k places the beginning of the executable code by using the /c option. The format of this option is:

```
 /cxxxxxxxx
```

where xxxxxxxx is an eight-digit hexadecimal address that marks the start of the code space. SB68k assigns this address to the assembler label CODEBEG; if you do not use the /c option, SB68k assigns a default value of $5000 to CODEBEG.

You can control where SB68k places the top of the target's stack space by using the /s option. The format of this option is:

```
/sxxxxxxxx
```

where xxxxxxxx is an eight-digit hexadecimal address that marks the top of the stack space. SB68k assigns this address to the assembler label STKBEG; if you do not use the /s option, SB68k assigns a default value of $4ff0 to STKBEG.

You can control the type of branch instruction SB68k creates by using the /b option. The format of this option is:

```
/b
```

SB68k normally converts a transfer or jump instruction into two assembly language source lines. The first line is a relative branch around the next line, and the second line is a long jump to the target address. For example:

```
while  n=3
               ' other code goes here
wend
```

contains a branch instruction that tests the value of variable N and branches back to the WHILE statement if N equals 3.

SB68k normally generates code similar to:

```
whl000
    moveq.l   #$3,d0
    cmp.l     var003(a5),d0
    beq       *+8                   if equal, branch
    jmp       whl001                not equal, exit loop
                                    other code goes here
    jmp       whl000                back to top of loop
whl001
```

For short transfers, where the branch target is within the relative addressing limit of the target MCU, this code is larger and will run more slowly than necessary.

Using the /b option forces SB68k to generate relative branches directly to all targets. If the /b option is in effect, SB68k would generate the following code for the above example:

```
whl000
    moveq.l #$3,d0
    cmp.l   var003(a5),d0
    bne     whl001                if not equal, branch
                                  other code goes here
    jmp  whl000                   back to top of loop
whl001
```

Note that the branch has reversed sense and the JMP instruction has disappeared.

WARNING

Branches to addresses beyond the 68000's relative branch limit will result in assembler errors, even though SB68k will not report any compilation errors. SB68k does not maintain an internal program counter and will not detect that a branch target is out of range.

Beginning users should omit the /b option, and accept the slight increase in size and execution times caused by the default branch code generation.

Experienced users may, however, use the /b option to gain improved performance. In this case, however, you must carefully monitor the assembler's output for any errors resulting in out-of-range branches.

If your code generates out-of-range branches using the /b option, recompile without the option. SB68k currently does not support any method for selectively compiling direct branches.

Some 68000 target platforms use on-chip firmware to take over the MCU's interrupt vector table. In this case you need to prevent SB68k from trying to set up a vector table on the target machine.

You can prevent SB68k from creating an interrupt vector table by using the /i option. The format of this option is:

```
/i
```

If you use the /i option and your SB68k program must use interrupts, you will have to add SB68k code to prepare the appropriate RAM-based jump table. Refer to the Motorola literature on your target MCU for details.

Note that the /i option suppresses ALL changes to the vector area, including the reset vector. SB68k programs compiled with the /i option must use some resident firmware to transfer control to the start of the program.

6. Environment variables

SB68k supports the use of two DOS environment variables. These variables can help ease development of multiple projects in SB68k.

When SB68k begins execution, it checks for the existence of two environment variables, SB68K_INCLUDE and SB68K_LIBRARY. SB68k assumes SB68K_INCLUDE contains the path to a directory containing custom INCLUDE files. Similarly, SB68k assumes SB68K_LIBRARY contains the path to a directory containing the standard SB68k library files. If either of these environment variables does not exist, SB68k defaults to the current directory when searching for any corresponding files.

You can assign a path to either of these variables in your AUTOEXEC.BAT file, using DOS' SET command.

Example:

```
set  SB68K_INCLUDE=C:\MYPROJ\INC
set  SB68K_LIBRARY=C:\SBASIC68\LIB
```

These commands assign paths to the SB68K_INCLUDE and SB68K_LIBRARY environment variables.

7. Library files

SB68k normally compiles all operations into in-line assembly language source for the target machine. In some cases, however, a function may translate into so many lines of source code that inserting the code in-line each time the function is used would yield an unacceptably large output file.

In these cases, SB68k automatically appends one or more files of assembly language source code to the output file. These files, called library files, contain pre-written source code for performing the corresponding operation.

For example, most versions of the 68000 require several lines of assembly language code to perform a 32-bit by 32- bit multiplication. Rather than insert this large section of assembler source every time your program uses the * operator, SB68k instead compiles a JSR to a library assembly language subroutine.

At the end of your output file, SB68k then includes the library file containing the source code for this multiplication subroutine.

SB68k only includes library files when necessary, based on your source code.

One library file deserves special mention. SB68k always includes the library file START68K.LIB during each compilation. The assembly language source in this file will be executed each time the target machine begins running your SB68k program. In fact, the code in this file is executed immediately following system reset.

If your SB68k application requires changes to the startup library code, you can customize START68K.LIB to include those changes.

Note, however, that you should not change any of the labels provided in the original version of START68K.LIB. Other parts of the SB68k system require that those labels exist, and that they be named exactly as they are.

8. Features

SB68k is a free-form Basic that supports enough control structures, such as IF-ELSE-ENDIF, that line numbers should not be necessary. It does not expect nor support line numbers; if you use them, you will get a syntax error back.

SB68k does not support GOTO.

SB68k generates code that uses the target's largest commonly available accumulator(s). This means that for the 68000, SB68k uses 32-bit variables and 32-bit math operations.

SB68k compiles down to fairly concise assembly language. It does no optimization from source line to source line. That is, it does not maintain a history of register usage and attempt to optimize out redundant operations. Even so, the generated code is quite compact, and will run fast enough to accommodate most projects.

For those projects that demand higher performance, SB68k allows you to embed assembly language source directly in your program. These assembly statements are passed intact to the target assembler.

SB68k is case-insensitive with regard to statements, labels, variables, and constants. The variable FOO may also be referred to as foo, Foo or fOo.

SB68k has built-in maximums for several compilation elements such as variables and labels. These limits are:

- Depth of FOR-NEXT nesting is 25. No one should EVER hit this limit. Compilation parsing stack is limited to 60 atoms. This is an internal limit of the compiler that determines how complex a statement the compiler can parse. Again, no one should ever hit this limit.

- Compilation data stack is limited to 60 cells. This should be sufficient for all programs.

- Maximum number of variables that a program can declare is 400. Note that an array, no matter how large, counts as one variable.

- Maximum number of constants is 400. Maximum number of labels is 500.

SB68k supports the following Basic functions and operators:

```
rem             starts an SB68k comment
'               (single quote) starts an SB68k comment
include         includes other SB68k source files
org             changes location of generated code
data            stores 32-bit values in a ROM table
dataw           stores 16-bit values in a ROM table
datab           stores 8-bit values in a ROM table
copy            copies a block of data between two memory areas
```

=	assignment
+	addition
-	subtraction; unary negation
~	1's complement
*	integer multiply (signed)
/	integer divide (signed)
mod	integer modulus
and	boolean AND
or	boolean OR
xor	boolean XOR
=	test, equal
<	test, less-than
>	test, greater-than
<>, ><	test, not-equal
<*	test, unsigned less-than
>*	test, unsigned greater-than
rshft()	shift argument 1 bit to right
lshft()	shift argument 1 bit to left
rroll()	rotate argument 1 bit to right
lroll()	rotate argument 1 bit to left
rshftb()	shift argument 1 byte to right
lshftb()	shift argument 1 byte to left
rrollb()	rotate argument 1 byte to right
lrollb()	rotate argument 1 byte to left
min()	returns smaller of two values (signed)
max()	returns larger of two values (signed)
minu()	returns smaller of two values (unsigned)
maxu()	returns larger of two values (unsigned)
peek()	read 32-bit contents of an address
peekw()	read 16-bit contents of an address
peekb()	read 8-bit contents of an address
poke	write 32-bit value to an address
pokew	write 16-bit value to an address
pokeb	write 8-bit value to an address
swapw	exchange words (16-bits) within a value
for	starts a FOR-NEXT iterative loop
to	signed test in a FOR-NEXT loop
to*	unsigned test in a FOR-NEXT loop
step	optional part of a FOR-NEXT loop
next	ends a FOR-NEXT loop
if	starts an IF-ELSE-ENDIF structure
else	part of an IF-ELSE-ENDIF structure
elseif	part of an IF-ELSE-ENDIF structure
endif	ends an IF-ELSE-ENDIF structure
while	starts a WHILE-WEND structure
wend	ends a WHILE-WEND structure
do	starts a DO-LOOP structure
while	optional part of a DO-LOOP structure
until	optional part of a DO-LOOP structure
loop	ends a DO-LOOP structure
waitwhile	waits while an I/O condition exists
waituntil	waits until an I/O condition occurs
select	starts a SELECT-CASE structure
case	starts a CASE clause within a SELECT-CASE
endcase	ends a CASE clause
endselect	ends a SELECT-CASE structure
exit	leaves loop structure early

```
print          output text to the console
printu         output text; numbers print as unsigned
printx         output text; numbers print as hexadecimal
inkey()        input a character from the console
outch          output a character to the console
interrupt      marks start of an SB68k ISR
const          creates a named constant
declare        creates a 32-bit variable or array
asm            marks start of inline assembly language source
endasm         marks end of inline assembly language source
addr()         returns address of a label, variable, or start of an array
push           pushes a value onto the SB68k data stack
pop()          pops a value from the SB68k data stack
pull()         synonym for pop()
place          change an element in the SB68k data stack
pick()         copy an element from the SB68k data stack
drop           remove one or more elements from SB68k data stack
interrupts     enables or disables system interrupts; also sets lowest allowed
interrupt level
gosub          invokes an SB68k subroutine
usr()          invokes an SB68k subroutine, returns one value
return         returns from an ISR or subroutine
end            ends an SB68k program or ISR
```

9. Remarks

SB68k provides two comment delimiters, for imbedding remarks in your source files. The traditional REM statement can be used to start a comment at nearly any point in an SB68k program. You can also use the newer ' (single-quote). All text following a remark delimiter is ignored by the SB68k compiler.

You can place a comment at the beginning of any line. You can also place a comment at the end of any complete SB68k statement. You cannot place a comment within an SB68k statement.

Example:

```
rem  This is a legal comment
'    So is this

a = c + 5                ' this is a legal comment, too
a = c +                  ' this is illegal!
```

Note that you can always insert a blank line anywhere in your source; SB68k always ignores blank lines.

10. Include files and the INCLUDE statement

SB68k supports the use of include files to help you organize and maintain your projects. Include files are simply files containing SB68k source code for commonly-used functions.

You can insert any include file into your SB68k program file by using the INCLUDE statement. SB68k will automatically open the named file, compile the code it contains, then resume compiling your original file.

For example, you might keep a single file of SB68k code for controlling servo motors. You can force SB68k to include the code in this file (call it servo.bas) in your current program file, by using the INCLUDE statement:

```
include  "servo.bas"
```

Note the use of double-quotes around the file name.

You can, if you like, supply a full pathname with the file name. For example:

```
include  "c:\sbasic\inc\servo.bas"
```

forces SB68k to search only the supplied path for the file servo.bas. If SB68k cannot find the file using this path, it will report an error.

You can also, if you wish, set the DOS environment variable SB68K_INCLUDE to contain the full pathname of a directory dedicated to holding your include files. If SB68K_INCLUDE exists, SB68k will search that directory for any files named in INCLUDE statements, provided that the file name does not itself contain any path information. If SB68K_INCLUDE does not exist, SB68k defaults to searching the current directory.

To summarize:

1. If the file name does not contain any path information, SB68k checks for the existence of a DOS environment variable, SB68K_INCLUDE. If SB68K_INCLUDE exists, SB68k searches the path in that variable for the named file. If SB68K_INCLUDE does not exist, SB68k searches the current directory.

2. If the file name contains path information, SB68k checks only the given path, regardless of the existence of SB68K_INCLUDE.

Rule 2 above means that you can force SB68k to search the current directory, even if SB68K_INCLUDE exists, by using an INCLUDE statement of the form:

```
include  ".\test.bas"
```

Here, the backslash serves as path information, forcing SB68k to search the full path given in the INCLUDE statement.

11. Labels

SB68k does not support line numbers, but it does support line labels. Line labels consist of a string of up to 20 characters, including a required terminating colon (:)

Labels must begin with an alphabetic character or an underscore ('_'); remaining characters in a label can also include digits. Any text following a line label definition is ignored.

NOTE

Though legal, starting labels with an underscore can cause obscure problems if you embed assembly language in your SB68k source file. See the section below on the ASM statement and the ENDASM statement, regarding references to SB68k variables from within an ASM block.

Example:

```
foo:                ' define the line label foo
     a = 3          ' write a value to A
     return         ' return from this subroutine

main:               ' start of the main program
     gosub foo      ' execute the subroutine foo
```

This example shows several important points. Note that labels require a trailing colon only when they are defined, but not when they are referenced. Thus, the GOSUB to FOO doesn't need a colon at the end of FOO.

Also, every SB68k program MUST contain the line label MAIN, even if it contains no other line labels. The startup code that supports SB68k on the target system always jumps to the label MAIN to begin execution. If your program does not have a MAIN, the compiler will report an error.

Note that the line label MAIN does not mark the first line of your program's code; it only marks the starting point for execution of your program following reset. You are free to place the line label MAIN anywhere in your file you deem appropriate.

12. Numeric constants

SB68k supports decimal, hexadecimal, and binary numeric constants. To enter a hexadecimal number in an SB68k file, use the $ prefix. To enter a binary number in an SB68k file, use the % prefix.

Hexadecimal numbers may contain the characters 0-9, A-F, and a-f. Binary numbers may contain the characters 0 and 1.

The following examples show how to enter different numeric constants:

```
foo = 1234          ' assigns decimal 1234 to FOO
bar = $1234         ' assigns hexadecimal $1234 to BAR
alpha = %10000      ' assigns decimal 16 to alpha
cat = $12 + 34      ' adds hex $12 to decimal 34
```

SB68k also supports ASCII character constants. To enter an ASCII constant, enclose the character in single-quotes. The value used will consist of the binary equivalent of the quoted character. An ASCII constant always consists of eight bits; the upper eight bits of the variable involved will always be 0.

For example:

```
foo = 'a'           '  assigns lowercase-A (97) to foo
```

13. Variables, arrays, and named constants

SB68k requires you to declare the names of all variables used in your program. You declare variables with the DECLARE statement. For example,

```
declare  foo
```

creates the SB68k variable FOO.

Variable names must begin with an alphabetic character or an underscore ('_'); remaining characters in a variable name can also include digits.

> NOTE
> Though legal, starting variable names with an underscore can cause obscure problems if you embed assembly language in your SB68k source file. See the section below on the ASM statement and ENDASM statement, regarding references to SB68k variables from within an ASM block.

All variables use four bytes of RAM (32 bits). The first variable defined is always located at assembler address VARBEG. Variables are assigned addresses based on the order of their declarations.

You must declare a variable before your code can reference that variable. This means that you will usually place all DECLARE statements in a block at the beginning of your SB68k source file.

Note that, unlike traditional Basics, SB68k does not automatically initialize all variables to zero. The value of any variable following system reset is unknown! Your SB68k program must provide any needed variable initialization.

SB68k also supports single-dimension arrays, or vectors. Each element in a vector array occupies one 32-bit location (four bytes). You use the DECLARE statement to define a vector array in much the same way you use it to define a simple variable. For example:

```
declare  foo(5)
```

defines the vector array FOO, consisting of five sequential 32-bit locations. The first element in any vector array is always element zero. Thus, FOO in the above example consists of the five elements named FOO(0) through FOO(4).

SB68k also supports double-dimension arrays. Each element in a 2D array occupies one 32-bit location (four bytes). You use the DECLARE statement to define a 2D array in much the same way you use it to define a vector array. For example:

```
declare  foo(5,4)
```

defines the 2D array FOO, consisting of 20 sequential 32-bit locations. The first element in a 2D array is always element (0,0). Thus, FOO in the above example consists of the 20 elements named FOO(0,0) through FOO(4,3). When using 2D array, remember that the elements are arranged in memory with the first subscript incrementing faster. Thus, the map for the 2D array FOO would look like:

```
FOO(0,0), FOO(1,0), FOO(2,0), ... FOO(2,3), FOO(3,3),
FOO(4,3)
```

You can use arrays anywhere a variable name would be legal, including the left side of an assignment operator. For example:

```
declare  foo(5,4)

foo(2,0) = 100/n
```

SB68k allows you to create named 32-bit constants. You create constants with the CONST statement. For example:

```
const  bar = 34
```

creates the SB68k constant BAR with a value of 34. CONST statements may refer to previously defined constants, and may include any number of math operations. CONST statements may not, however, refer to variables, as the contents of variables are not known at compile-time. If you refer to a variable inside a CONST statement, the compiler will report an error.

You must create a constant before your code can reference that constant. This means that you will usually place all CONST statements in a block at the beginning of your SB68k source file.

Note that named constants do not consume any space in the final executable file. They only exist as equates in the assembler source file generated by SBasic.

14. Data tables

SB68k allows you to store tables of data in ROM, for access by your program at run-time. This technique is used often for storing pre-defined information, such as lookup tables for motor speeds or mathematical functions.

To store 32-bit values in a data table, use the DATA statement. Follow the DATA statement with a list of values to be written into ROM. For example:

```
data      0, 1, 2, 3                  ' store 4 32-bit values in table
```

SB68k will generate suitable assembly language source to store the values into code memory at the current location. For the 68000, this example would generate assembly language source similar to:

```
dc.l      0,1,2,3
```

To store 16-bit values in a data table, use the DATAW statement. Follow the DATAW statement with a list of values to be written into ROM. For example:

```
dataw     1234, 5678, $1111, $4321    ' store 4 16-bit values in table
```

SB68k will generate suitable assembly language source to store the values into code memory at the current location. For the 68000, this example would generate assembly language source similar to:

```
dc.w      $04d2,$162e,$1111,$4321
```

To store 8-bit values in a data table, use the DATAB statement. Follow the DATAB statement with a list of values to be written into ROM. For example:

```
datab     $ff, 123, 256, 'z'          ' store 4 8-bit values
```

SB68k will generate suitable assembly language source to store the values into code memory at the current location. For the 68000, this example would generate assembly language source similar to:

```
dc.b      $ff,$7b,$00,$7a
```

Note that the third value appears in the SB68k source as 256, but is converted to 0 in the output source file. This happens because SB68k only writes the low eight bits of a DATAB list item to the output file.

In order to access items within a DATA (or DATAB or DATAW) table, you must provide a label at the start of the list. Your program can then use this label to find the first item in the list. For example:

```
declare   n
declare   sum

foo:
data      1,2,3,4
data      5,6,7,8
```

```
main:
sum = 0
for  n = 0 to 7
     sum = sum + peek(addr(foo) + n * 4)
next
end
```

This code uses the ADDR() function to locate the start of the data table at label FOO. The list item of interest is found by adding an offset value (N * 4) to the address of FOO. The PEEK() function then reads the 32-bit value stored at that address in the table.

To use the above technique with a DATAW table, you must change PEEK() to PEEKW() and replace the multiplication by 4 in the offset calculation with a multiplication by 2. To use the above technique with a DATAB table, you must change PEEK() to PEEKB() and remove the multiplication by 4 in the offset calculation.

Note that SB68k writes a DATA table in place. That is, the table appears in exactly the same position in the output file as it appears in your SB68k source file.

This means you cannot place a DATA table inside a block of executable code. If you do, the target MCU will eventually try to execute the DATA table as if it were machine code, and your program will crash.

For that reason, put your DATA statements outside of executable blocks. A good place to write DATA statements is immediately before the MAIN: label in your program.

15. COPY statement

SB68k provides the COPY statement, used to move data from one memory area to another. This proves very handy when you need to initialize a large array. Format of the COPY statement is:

```
copy  from, to, count
```

where FROM is the address of the data block, TO is the address of the destination area, and COUNT is the number of BYTES (not variables) to move.

For example, assume you need to initialize the array FOO with a table of data, already existing in a block of DATA statements. You would use statements similar to:

```
declare  foo(10)

table:
data     $1234, $5678, $1, $2, $3, $4
data     $5, $6, $7, $8

copy  addr(table), addr(foo), 40
```

This code uses the ADDR() function to locate the addresses of the TABLE data block and the FOO array, then moves 40 bytes of data from the table to the array.

Note that COPY cannot be used to move data into an area of memory that overlaps the source area. Should you need to perform this operation, use two COPY statements, first to move the data into an intermediate area, then to move it into the final destination area.

16. Operators

SB68k supports all of the common arithmetic operators:

= (equal-sign) sets the value of a variable to the result of a calculation. This is the traditional assignment operator. All assignments store a 32-bit value into a variable.

+ (plus-sign) performs 32-bit addition.

- (minus-sign) performs 32-bit subtraction. It also acts as the unary negation operator.

~ (tilde) performs 32-bit one's complement. This is logically identical to N XOR $ffffffff.

* (asterisk) multiplies two 32-bit signed values, yielding a 32-bit product.

Run-time support for SBasic68k's multiplication operator on a 68000 MCU relies on a library file, included with the SB68k distribution. This file is automatically added to the assembler source file created by SBasic68k, whenever your program invokes the multiplication operator.

Note that this library file is only included if your code uses a multiply operation. You can create smaller executables by eliminating any use of the multiplication operator, if appropriate.

/ (forward-slash) divides a 32-bit dividend by a 32-bit divisor, yielding a 32-bit quotient; the remainder is lost.

Run-time support for SBasic68k's division operator on a 68000 MCU relies on a library file, included with the SB68k distribution. This file is automatically added to the assembler source file created by SBasic68k, whenever your program invokes the multiplication operator.

Note that this library file is only included if your code uses a divide operation. You can create smaller executables by eliminating any use of the division operator, if appropriate.

MOD (modulus) divides a 32-bit dividend by a 32-bit divisor, yielding a 32-bit remainder; the quotient is lost.

Run-time support for SBasic68k's division and MOD operators on a 68000 MCU relies on a library file, included with the SB68k distribution. This file is automatically added to the assembler source file created by SBasic68k, whenever your program invokes either operator.

Note that this library file is only included if your code uses a division or MOD operation. You can create smaller executables by eliminating any use of these operators, if appropriate.

Examples:

```
alpha = foo + bar          ' adds two variables
beta = gamma - $1234       ' subtracts a hex constant
```

```
var1 = 3 * var2           ' multiplies a variable
c = delta / 88            ' divides a variable by a constant
m = gamma MOD 10          ' takes the modulus function
```

SB68k also supports most of the common Boolean operators:

AND performs the logical AND of two 32-bit values.

OR performs the logical inclusive-OR of two 32-bit values.

XOR performs the logical exclusive-OR of two 32-bit values.

Examples:

```
alpha = foo AND $7f       ' leaves only low 7 bits of foo
beta = alpha OR 255       ' sets all low 8 bits of alpha
gamma = beta XOR $ffffffff      ' inverts all bits in beta
```

17. Comparisons

SB68k supports a wide range of comparisons, for use with control structures such as IF-ELSE-ENDIF and DO-LOOP. All comparisons test two 32-bit values and return TRUE if the values meet the comparison test.

= (equal-to) yields TRUE if the two 32-bit values are equal.

< (less-than) yields TRUE if the first 32-bit value is less than the second 32-bit value. This is a signed comparison; $80000000 is less than 0.

> (greater-than) yields TRUE if the first 32-bit value is greater than the second 32-bit value. This is a signed comparison; 0 is greater than $80000000.

<> (not-equal-to) yields TRUE if the two 32-bit values are not equal.

>< (not-equal-to) yields TRUE if the two 32-bit values are not equal.

<= (less-than-or-equal-to) yields TRUE if the first 32-bit value is less than or equal to the second 32-bit value. This is a signed comparison.

>= (greater-than-or-equal-to) yields TRUE if the first 32- bit value is greater than or equal to the second 32-bit value. This is a signed comparison.

<* (less-than unsigned) yields TRUE if the first 32-bit value is less than the second 32-bit value. This is an unsigned comparison; 0 is less than $80000000 unsigned.

>* (greater-than unsigned) yields TRUE if the first 32-bit value is greater than the second 32-bit value. This is an unsigned comparison; $80000000 is greater than 0 unsigned.

SB68k does not allow you to store the result of a comparison in a variable. SB68k also does not allow multiple comparisons in a single operation. Control structures such as IF-ELSE-ENDIF can use only one comparison in the IF- clause.

18. Control structures

SB68k supports several structures for controlling the flow of your program. Used properly, these control structures can improve the quality of your program design, making your source file easier to read, understand, and debug.

Some of the following control structures allow or require a comparison clause. Such clauses consist of an expression, a comparison operator, and a second expression. The comparison clause is evaluated as your program runs and, depending on the evaluation, control transfers within the control structure.

Example:

```
while  a < 5
     a = a + 1
wend
```

Here, the comparison clause "a < 5" determines whether control remains in the WHILE-WEND loop or transfers to the line following the WEND statement.

All comparison clauses may contain one and only one comparison operator. Multiple comparisons, such as:

```
while a < 5 and c-1 = 3
```

are illegal and will generate compilation errors.

Note also that you do not enclose comparisons within parentheses. Doing so will result in a compilation error.

18.1. WHILE-WEND

WHILE-WEND is a conditional loop structure. Control executes all statements between the WHILE statement and the WEND statement for so long as the comparison in the WHILE clause is TRUE. When the comparison becomes FALSE, control exits the loop by transferring to the line following the WEND statement.

Example:

```
while  a < 500            ' loop while a is less than 500
     a = a + 1            ' increment a
wend                      ' end of loop
```

This example loops for so long as the value in A is less than (signed) 500.

Note that WHILE-WEND is obsolete, even though SB68k still supports it. WHILE-WEND has been replaced by the more flexible DO-LOOP structure.

18.2. DO-LOOP

DO-LOOP is a loop structure that may be either conditional or unconditional. Control executes all statements between the DO statement and the LOOP statement. You may include an optional comparison clause in either the DO statement or in the LOOP statement.

Examples:

```
do                          ' start of a do-loop
     gosub process          ' do something useful
loop                        ' end of loop
```

The above example will loop forever, since it has no comparison clause. An endless loop is often used as the core of a large program.

```
do while a < 500            ' loop while a is less than 500
     a = a + 1              ' increment a
loop                        ' end of loop
```

This example mimics the WHILE-WEND loop above.

```
do until a > b              ' loop until a is greater than b
     a = a + c/2            ' change value of a
loop                        ' end of loop
```

The above example loops until the value in A is greater (signed) than the value in B.

```
do                          ' start of a do-loop
     a = peekb($1000)       ' read a value from an I/O port
loop while a = 0            ' loop while a equals 0
```

The above example loops for so long as the value read from the I/O port equals 0.

```
do  ' start of a do-loop
     a = peekb($1000)     ' read a value from an I/O port
loop until a = $ff        ' loop until a equals $ff
```

The above example loops until the value read from the I/O port equals $ff.

The DO-LOOP provides great flexibility for loop control, because you can control when the comparison occurs in the loop. If you place the comparison in the DO statement, the test is performed before the body of the loop is executed. By placing the comparison in the LOOP statement, you can force the body of the loop to execute before the comparison is performed.

18.3. FOR-NEXT

The FOR-NEXT structure creates an iterated loop. This means that a selected variable, called the index variable, controls exactly how many times the loop is executed.

Control executes all statements between the FOR statement and the NEXT statement until the value in the index variable EXCEEDS a specified limit. The index variable is always tested at the top of

the loop. The comparison is signed for the usual FOR-NEXT loop, although you can use an unsigned comparison if necessary.

Example:

```
for n = 1 to 10
     a = a + n
next
```

Here, the variable N starts with a value of one. The statement inside the loop is executed ten times, with N incrementing each time the NEXT statement executes.

Eventually, N holds the value 11 when the FOR statement executes. At this point, the statement inside the loop is not executed. Instead, control passes directly to the statement following the NEXT statement.

Sometimes you must use a limit larger than $7fffffff. Since SB68k uses 32-bit math, numbers larger than $7fffffff are treated as negative in signed comparisons. Therefore, the following example:

```
for n = 1 to $90000000
     a = a + 1
next
```

will exit immediately, as SB68k treats $90000000 as a negative number, and 1 is already greater than a negative number.

To change the above example to use an unsigned comparison, use the TO* operator. The above example becomes:

```
for n = 1 to* $90000000
     a = a + 1
next
```

This loop will execute the expected $90000000 times.

Remember to leave room on your limit value so that the index variable can actually exceed the limit. For example:

```
for n = 1 to* $ffffffff
     a = a + 1
next
```

This loop will never end, since the value of N can never exceed the limit of $ffffffff.

Note that it is legal (but not good practice) to modify the index variable inside the loop.

Normally, the index variable increments by one each time control reaches the bottom of the loop. You can change the value by which the index variable increments with the STEP operator.

Example:

```
for n = 1 to 10 step 2
     a = a + n
next
```

Here, the variable N starts with a value of one, and increments by two each time control reaches the NEXT statement. Thus, N takes on the values 1, 3, 5, 7, 9, and 11. When N becomes 11 at the top of the loop, control immediately passes to the statement following the NEXT statement.

18.4. SELECT-CASE

The SELECT statement marks the beginning of a SELECT-CASE control structure. The SELECT-CASE structure allows your code to select one option out of a list, based on the value of an argument, called the selector. Only the CASE statement associated with the matching selector value, if any, is executed.

The general format of a SELECT-CASE structure is:

```
select  foo          ' use value of FOO as selector
     case  123       ' if FOO = 123...
     .               ' execute this code
     .               '
     endcase         ' end of case 1

     case  456       ' if FOO = 456...
     .               ' execute this code
     .               '
     endcase         ' end of case 2
endselect            ' end of select structure
```

This structure replaces the older ON-GOTO construct, and allows creation of code that is easier to debug and maintain.

The SELECT-CASE structure supports variations for handling special cases. For example:

```
select  foo + 3*j         ' use expression for selector
     case  1              ' if selector = 1...
     case  n              ' or N...
     case  'X'            ' or ASCII character X...
     foo = n + 3          ' change FOO
     endcase              ' end of first case

     case  2              ' if selector = 2...
     case  3              ' or 3...
     foo = n + 2          ' change FOO
     endcase              ' end of second case

     foo = 4              ' default action...
endselect
```

This example shows the use of multiple CASE statements within a CASE clause. This feature comes in handy if your code must perform the same function for a group of selector values.

This example also shows the method for performing a default action. The line FOO = 4 executes if no CASE clause matches the selector value. Note that the default clause does not require an initial CASE statement or a terminating ENDCASE statement.

Also, this example shows that the selector value for this example is an expression. You can use any valid algebraic expression as the selector value in a SELECT statement.

Note, however, that CASE statements only accept a numeric value, a constant, or a variable as an argument. You cannot use an algebraic expression as the argument to a CASE statement! Doing so will generate a compiler error.

Finally, this example shows that you can change the selector value within a CASE clause, without affecting the actions of the SELECT statement. This is because control passes to the ENDSELECT statement immediately after executing the code in any CASE clause. Thus, any following CASE clauses do not evaluate the changed selector value.

18.5. EXIT

The EXIT statement allows you to leave a looping structure before the terminating condition, if any, is reached.

Control automatically jumps to the end of the currently active looping structure.

You can also use EXIT to leave a SELECT-CASE structure. In this case, control will jump to the corresponding ENDSELECT statement.

Example:

```
for n = 1 to 10
     if n = 5
          exit
     endif
next
```

Here, control automatically leaves the FOR-NEXT loop when N equals 5.

Note that you can use EXIT inside DO-LOOP, WHILE-WEND, FOR- NEXT, and SELECT-CASE structures. You cannot use EXIT outside of these looping structures.

18.6. IF-ELSE-ELSEIF-ENDIF

The IF-ENDIF structure selectively executes a block of statements, depending on a comparison at the beginning of the structure. If the comparison is TRUE, the statements are executed, otherwise they are not.

Example:

```
if a = $12
     b = b * 2
endif
```

Here, the middle statement is executed only if the value of A is $12.

You can use the ELSE modifier to provide greater flexibility to the IF-ENDIF structure. Statements between the ELSE statement and the ENDIF statement are only executed if the comparison in the IF statement is FALSE.

Example:

```
if a = $12
     b = b * 2
else
     b = -b
endif
```

Here, B is set to -B only if A is not $12.

Note that you do not include parentheses around the comparison clause of any structure. Doing so will cause SB68k to report an error during compilation.

You can also use the ELSEIF modifier to simplify nested IF-ENDIF structures.

Consider the following example, in which three different conditions must be tested:

```
if a = $12
     n = 1
else
     if foo = w+2
          n = 3
     else
          if bar = a+w
n = 5
          endif
     endif
endif
```

This type of test sequence can be difficult to decipher. Using the ELSEIF modifier simplifies the structure:

```
if a = $12
     n = 1
elseif  foo = w+2
     n = 3
elseif  bar = a+w
     n = 5
endif
```

Note that the ELSEIF modifier requires the same type of comparison clause used by the IF statement.

19. Functions and statements

SB68k supports several functions and statements useful for embedded control applications.

Generally speaking, a function returns a value, while a statement does not. All functions contain parentheses, though not all functions actually need an argument inside the parentheses.

All statements, however, appear without parentheses.

19.1. SWAPW()

The SWAPW() function exchanges the two 16-bit words of the 32-bit argument. This operation is useful for sending both halves of a variable to a word-wide I/O port.

Example:

```
j = $12345678            ' prepare j
pokew ioport, swapw(j)   ' send $1234 to I/O port
```

19.2. RSHFT() and LSHFT()

The RSHFT() function and LSHFT() function shift the 32-bit argument one bit position, either right or left. This operation can be used as a fast multiply or divide by 2.

Example:

```
n = $c0000003       ' initial value of n
n = rshft(n)        ' n now holds $60000001
n = lshft(n)        ' n now holds $c0000002
```

Note that the shift functions move the argument one bit in the specified direction, moving a 0 bit into the vacated position. Thus:

```
x <- xxxxxxx (32 bits) xxxxxxxx <- 0 = LSHFT()
0 -> xxxxxxx (32 bits) xxxxxxxx -> x = RSHFT()
```

19.3. RSHFTB() and LSHFTB()

The RSHFTB() function and LSHFTB() function shift the 32-bit argument one byte position, either right or left. This operation can be used as a fast multiply or divide by 256.

Example:

```
n = $123456         ' initial value of n
n = rshftb(n)       ' n now holds $00001234
n = lshftb(n)       ' n now holds $00123400
```

Note that the shift functions move the argument one byte in the specified direction, moving a $00 byte into the vacated position. Thus:

```
$xx <- xxxx (4 bytes) xxxx <- $00 = LSHFTB()
$00 -> xxxx (4 bytes) xxxx -> $xx = RSHFTB()
```

19.4. RROLL() and LROLL()

The RROLL() function and LROLL() function rotate the 32-bit argument one bit position, either right or left. This operation can be used as part of a pulse-width modulation (PWM) function.

Example:

```
n = %1111           ' initial value of n
n = rroll(n)        ' n now holds $80000007
n = lroll(n)        ' n now holds $0000000f
```

Note that the roll functions move the argument one bit in the specified direction, placing the rotated bit into the position at the opposite end of the word. Thus:

```
+-------<-------+              +-------->-------+
|               | = LROLL()    |               | = RROLL()
xxxx(32 bits)xxxx              xxxx(32 bits)xxxx
```

19.5. RROLLB() and LROLLB()

The RROLLB() function and LROLLB() function rotate the 32- bit argument one byte position (eight bits), either right or left. This operation can be used with byte-wide I/O functions.

Example:

```
n = $123456         ' initial value of n
n = rrollb(n)       ' n now holds $56001234
n = lrollb(n)       ' n now holds $00123456
```

Note that the roll functions move the argument one byte in the specified direction, placing the rotated byte into the position at the opposite end of the word. Thus:

```
+-------<-------+              +-------->-------+
|               | = LROLLB()   |               | = RROLLB()
xxxx(4 bytes)xxxx              xxxx(4 bytes)xxxx
```

19.6. PEEK()

The PEEK() function returns the 32-bit value stored in a specific address. This is the usual function for reading 32-bit I/O ports.

Example:

```
a = peek($1000)     ' get 32-bit value from address $1000
```

You can make your use of PEEK easier to understand if you combine it with named constants defined with the CONST statement.

Example:

```
const porta = $1000 ' define address of port A
a = peek(porta)     ' get 32-bit value at port A
```

19.7. PEEKW()

The PEEKW() function returns the 16-bit value stored in a specific address. This is the usual function for reading 16-bit I/O ports.

Example:

```
a = peekw($1000)    ' get 16-bit value from address $1000
```

You can make your use of PEEKW easier to understand if you combine it with named constants defined with the CONST statement.

Example:

```
const porta = $1000 ' define address of port A
a = peekw(porta)    ' get 16-bit value at port A
```

19.8. PEEKB()

The PEEKB() function returns the 8-bit value stored in a specific address. This is the usual function for reading 8- bit I/O ports.

Example:

```
a = peekb($1020)    ' get 8-bit value from address $1020
```

You can make your use of PEEKB easier to understand if you combine it with named constants defined with the CONST statement.

Example:

```
const portb = $1020 ' define address of port B
a = peekb(portb)    ' get 8-bit value at port B
```

Recall that the = (assignment) operator writes a 32-bit value to a variable. In the case of the PEEKB and PEEKW functions, the upper bits of the variable will always be written as 0. Thus, using PEEKB to read a value of $45 in the above example results in storing $00000045 in variable A.

19.9. POKE statement

The POKE statement writes a 32-bit value to a specific address. This is the usual statement for writing data to 32-bit I/O ports.

Example:

```
poke $1000, a       ' write value in A to address $1000
```

Note the difference in syntax between the PEEK() function and the POKE statement; PEEK uses parentheses around the address argument, while POKE does not use parentheses.

You can make your use of POKE easier to understand if you combine it with named constants defined with the CONST statement.

Example:

```
const porta = $1000 ' define address of port A
poke porta, a       ' write value in A to port A
```

19.10. POKEW statement

The POKEW statement writes a 16-bit value to a specific address. This is the usual statement for writing data to 16-bit I/O ports.

Example:

```
pokew $1000, a      ' write low 16 bits in A to addr $1000
```

Note the difference in syntax between the PEEKW() function and the POKEW statement; PEEKW uses parentheses around the address argument, while POKEW does not use parentheses.

You can make your use of POKEW easier to understand if you combine it with named constants defined with the CONST statement.

Example:

```
const porta = $1000 ' define address of port A
pokew porta, a      ' write low 16 bits in A to port A
```

19.11. POKEB statement

The POKEB statement writes an 8-bit value to a specific address. This is the usual statement for writing data to 8-bit I/O ports.

Example:

```
pokeb $1000, a      ' write low 8 bits in A to addr $1000
```

Note the difference in syntax between the PEEKB() function and the POKEB statement; PEEKB uses parentheses around the address argument, while POKEB does not use parentheses.

You can make your use of POKEB easier to understand if you combine it with named constants defined with the CONST statement.

Example:

```
const porta = $1000 ' define address of port A
pokeb porta, a      ' write low 8 bits in A to port A
```

19.12. WAITWHILE and WAITUNTIL statements

The WAITWHILE statement and WAITUNTIL statement provide high-speed testing loops, for use with 8-bit I/O ports. You can use these single statements to replace larger, less efficient wait-loops built from the PEEKB() function.

The WAITWHILE statement loops while the contents of an 8-bit port, ANDed with an 8-bit mask, yields a non-zero result.

The WAITUNTIL statement does the opposite; it loops UNTIL the contents of an 8-bit value, ANDed with an 8-bit mask, yields a non-zero result.

Example:

```
waitwhile $1000, $40     ' wait while bit 6 of $1000 is high
```

This statement repeatedly reads the 8-bit value at address $1000 and ANDs that value with $40, for so long as the result is not zero. When the result equals zero, the loop terminates.

Example:

```
waituntil j, n      ' wait until mask of value at j is not 0
```

This statement repeatedly reads the 8-bit value at the address contained in variable J and ANDs that value with the low eight bits in variable N, until the result is not zero. When the result is not zero, the loop terminates.

You must be aware of a few characteristics of the WAIT loops. The first argument is always treated as an address, even if you use a variable. In the second example above, the loop does not test the value in J, it uses the value in J as the address to test.

Second, the WAIT statements always test an 8-bit address; you cannot test a 16-bit I/O port with these statements.

Also, the WAIT statements always use the low eight bits of the mask argument. Thus, if you specify a variable as the mask value, the WAIT statements will automatically use just the low eight bits in the loop test.

Finally, you can improve the speed of the generated code by using only constants, numbers, or variables as arguments to these statements. Using an argument that contains math or logical operations will generate larger, slower test loops.

Example:

```
waitwhile j+4, n/q  ' this runs slowly
```

will run slowly, since the two math operations will be performed inside each test loop. A better way to write this is:

```
adr = j+4                 ' calc the address
mask = n/q                ' calc the mask
waitwhile  adr, mask      ' now do the loop
```

19.13. ADDR()

The ADDR() function returns the address of a specified label or variable.

Example:

```
a = addr(MyLabel)    ' put address of MyLabel in A
```

You can use the ADDR() function to locate the first element in an array. To do this, simply leave off the parentheses when supplying the array's name. For example:

```
declare  foo(5)

a = addr(foo)
```

causes A to contain the address of FOO(0).

You cannot use ADDR() to calculate the address of a selected array element; if you include a subscript after the array name, the compiler will report an error in the ADDR() function.

This isn't really a problem, though, since all array elements occupy four bytes. For example:

```
a = addr(foo) + n * 4
```

causes A to contain the address of FOO(N). It does this by finding the address of FOO(0), then adding four to thataddress for each element named in N. Thus, if N = 2, A will hold the address of FOO(2).

See the discussion below on the INTERRUPT statement for a detailed example of using the ADDR() function.

19.14. PRINT, PRINTU, and PRINTX statements

SB68k supports a limited PRINT statement. SBasic68k's PRINT statement sends characters to a default output device, based on the target system. This output device is platform-dependent, and you will need to modify the outch68k.lib file so it handles the hardware on your system properly.

SB68k supports the following variations of the PRINT statement:

```
print "a constant string followed by a CR"
print "a string followed by a space";
print "a string followed by a TAB character",
print                     ' prints a blank line
print foo                 ' prints the value of foo
print "FOO ="; foo        ' prints a string, then a value
print a; b; c             ' prints three values
```

Additionally, SB68k supports two statements similar to PRINT that print values in slightly different formats. The PRINTU statement prints any values in unsigned format and the PRINTX statement

prints any values in hexadecimal characters. The PRINTU and PRINTX statements behave exactly the same as the PRINT statement with regard to spacing, tabs, and quoted strings.

For example:

```
print   "-1 = "; -1 ' prints -1 = -1
printu  "-1 = "; -1 ' prints -1 =  4294967295
printx  "-1 = "; -1 ' prints -1 = FFFFFFFF
```

SB68k also supports the C language's escape character for embedding special characters within a PRINT string. You can embed any of the following special characters inside a PRINT string:

```
"\n" inserts a newline ($0a)
"\r" inserts a carriage-return ($0d)
"\f" inserts a form-feed ($0c)
"\a" inserts an alert ($07)
"\b" inserts a backspace ($08)
"\t" inserts a horizontal tab ($09)
"\v" inserts a vertical tab ($0b)
"\\" inserts a backslash
```

For example:

```
print "Hello, world!\n\r\a";
```

prints the string "Hello, world!" followed by a line-feed, a carriage-return, and a bell.

Run-time support for SBasic's PRINT statements relies on several library files, included with the SB68k distribution. These files are automatically added to the assembler source file created by SBasic68k, whenever your program invokes a variation of the PRINT statement.

You may modify the PRINT statement library routines, if desired, to create support for other output devices. However, you must keep the names of all subroutines defined inside a library file unchanged.

This is because the assembler source created by SB68k uses fixed names for library functions. If you change the names of the library routines, the assembler will report an undefined label error when it tries to find the library subroutines.

Note that these library files are only included if your code uses a PRINT statement. You can create smaller executables by eliminating any use of the PRINT statement, if appropriate.

NOTE

Your code must issue any setup instructions necessary to prepare the default I/O port for use with the PRINT statements. These are generally platform-dependent, and require that you have a good working knowledge of your target 68000 system.

19.15. INKEY()

SB68k supports a version of the INKEY() function. You can use INKEY() to receive characters from a default input device, based on the target system.

Unlike the INKEY() function in traditional Basics, SBasic's version does not expect or allow an argument.

If no character was received from the console, INKEY() returns 0. If a character was received, INKEY() returns a value containing the character in the low eight bits; additionally, INKEY() sets bit 8 of the returned value. Setting bit 8 allows your program to distinguish between receiving a NULL (returned value = $0100) and not receiving any character (returned value = $0).

For example:

```
do
     n = inkey()
loop while n = 0
n = n and $ff
```

This code loops until INKEY() returns a valid character from the console. It then strips off the upper byte, leaving only the received character in N.

INKEY() sets bit 8 to permit receiving any character, including NULLs, from the console input device.

The actual code that supports the INKEY() function appears in the library file INKEY68K.LIB. You can edit the supplied INKEY68K.LIB source file to support using INKEY() with other devices. Take care, however, to ensure your new version of INKEY() preserves the register usage of the original.

NOTE
Your code must issue any setup instructions necessary to prepare the default I/O port for use with the INKEY() function.

19.16. OUTCH statement

SB68k provides the OUTCH statement, used to send an 8-bit character directly to the console. This statement provides the same functionality as the usual Basic phrase:

```
PRINT CHR$(N);
```

only the OUTCH statement takes much less space, both in the source file and in the final executable.

For example:

```
outch  n+2
```

This example adds 2 to the current value in variable N, then sends the low eight bits of the sum directly to the console device as an ASCII character.

The actual code that supports OUTCH appears in the library file OUTCH68K.LIB. You can edit the supplied OUTCH68K.LIB source file to support using OUTCH with other devices. Take care, however, to ensure your new version of OUTCH preserves the register usage of the original.

NOTE
The routine contained in OUTCH68K.LIB is used by all console output statements, including all variations of PRINT. Changing the routine in OUTCH68K.LIB will also change the behavior of all variations of PRINT.

OUTCH only sends the low eight bits of its argument to the console. Therefore, OUTCH can be used directly with the value returned by INKEY(), as shown:

```
do
     n = inkey()
loop while  n = 0
outch  n
```

This code loops until INKEY() returns a non-zero value for N, indicating that a character was entered at the console. The value in N is then sent back to the console as an echo. Since OUTCH ignores the upper bytes in N, the character echoes properly; your code does not need to alter the upper bytes of N before calling OUTCH.

NOTE
Your code must issue any setup instructions necessary to prepare the default I/O port for use with the OUTCH statement.

19.17. MIN() and MAX()

The MIN() function and MAX() function return the smaller or greater of two arguments, respectively. Both functions use signed comparisons. You must supply two arguments for either function, with a comma separator between arguments. You may use either constants or algebraic expressions for either or both arguments. For example:

```
print min(-4, 3); "is smaller than"; max(-4, 3)
```

prints the correct result, as a signed comparison between -4 and 3 will properly show that -4 is smaller than 3.

19.18. MINU() and MAXU()

The MINU() function and MAXU() function perform the same operations as MIN() and MAX, except these functions use unsigned comparisons.

For example:

```
print minu(-4, 3); "is smaller than"; maxu(-4, 3)
```

will print apparent nonsense, as -4, taken as an unsigned number, is larger than 3.

20. Subroutines, GOSUB, and USR()

SB68k supports the traditional Basic concept of subroutines. A subroutine is a block of SB68k statements that can be invoked, or called, from elsewhere in the SB68k program. After these statements complete execution, control returns to the calling section.

A subroutine contains a line label marking the start of the subroutine, and at least one RETURN statement, which transfers control back to the calling section.

20.1. GOSUB statement

The calling section of SB68k code invokes, or calls, a subroutine by means of the GOSUB statement.

Example:

```
main:
do  ' start of an endless loop
     gosub foo      ' call subroutine FOO
loop                ' loop forever

foo:                ' start of subroutine FOO
a = a + 1           ' increment A
return              ' return to caller
```

This example, while not very useful, shows how a subroutine is invoked and how it is defined. Note that you can use a GOSUB to a subroutine before that subroutine is defined in your code.

Note that code inside a subroutine has full access to all variables defined with the DECLARE statement. Changes made to a variable from inside a subroutine remain in effect when control returns from that subroutine.

All GOSUBs push a return address onto the target's return stack. SB68k's data stack resides a fixed distance below the return stack. Excessive nesting of GOSUBs could clobber values on the data stack.

The address of the subroutine invoked must be either a label or a variable; you may not use algebraic expressions or functions as addresses for a GOSUB statement.

GOSUBs may pass one or more arguments to the called subroutine. Your code should include the arguments after the name of the subroutine invoked. For example:

```
main:
do                        ' start an endless loop
     gosub foo, 3, j      ' call FOO with two arguments
loop                      ' loop forever
```

SB68k automatically pushes all arguments onto the data stack, then calls the named subroutine as above. Code in the subroutine can use the data stack operators, such as the PICK() function, to test or change the arguments.

Pay careful attention to the order in which SB68k pushes the arguments; they are pushed in the order given. For example:

```
gosub foo, 3, j
          ^  ^
          |  +------- This argument is on top of data stack
          +---------- This argument is next on data stack
```

This method of passing arguments means that a subroutine may be passed different numbers of arguments at any time. SB68k performs no checks to see if you have passed the correct number of arguments.

SB68k similarly does not "clean up" the data stack before returning control to the calling routine. Your code must update the data stack, if necessary, to remove any arguments. You can choose to do this within the called routine, or in the calling routine after the subroutine invocation. Regardless of where you perform this cleanup, it must be done or your program will eventually corrupt the data stack, likely crashing.

Note that GOSUBs make no allowance for the called routine returning a value. Thus, code that uses a GOSUB to invoke a subroutine must rely on global variables to check the results of the GOSUB. This can result in awkward code that can prove difficult to maintain.

20.2. USR()

To return a value to the calling routine, use SBasic's USR() function. The USR() function is similar to GOSUB, in that it calls an SB68k subroutine. It can also include one or more arguments.

Unlike GOSUB, however, USR() returns a value from the called routine for later use. For example:

```
    n = usr(foo, 3)           ' call FOO, put returned value in N
    j = usr(bar)              ' call BAR with no arguments
```

Note the difference between USR() and GOSUB. The USR() function, like any other SB68k function, requires parentheses around its list of arguments.

The address of the subroutine invoked must be either a label or a variable; you may not use algebraic expressions or functions as addresses for a USR() function.

USR() functions use the data stack in exactly the same manner described above for the GOSUB statement. Also, the first argument in the USR() list is always the address of the called subroutine.

21. RETURN statement

SBasic's RETURN statement serves two functions. You can use it to return control from a subroutine, and you can use it to return control from an interrupt. For details on processing interrupts, see the Interrupts section below.

When used in main-line code (code not in an interrupt service routine), RETURN generates the proper code to return control from a subroutine. For the 68000 CPU, this is an RTS instruction.

Note that SB68k does not check to see if you are using a RETURN statement after a label. A RETURN statement always generates a Return-from-Subroutine instruction in main-line code, regardless of where it occurs.

When used in an interrupt section, RETURN generates the proper code to return control from an interrupt or exception. For the 68000 CPU, this is an RTE instruction.

Additionally, a RETURN statement in main-line code may optionally include a value that will be returned to the calling routine. For example:

```
return              ' this statement just returns
return  j+3         ' this statement returns a value
```

You may include a RETURN statement with a returned value at any point in your code, even inside an interrupt section. However, the returned value will only have meaning if it occurs in main-line code, and even then only if control is returning to a USR() invocation. Returned values to a GOSUB are ignored, as are returned values from an interrupt section.

Note that returning a value from inside an interrupt section is guaranteed to create obscure bugs, since the values created by the main-line code will change randomly when the interrupt completes. Do not try to return a value in the interrupt section unless you are a very experienced programmer and know exactly what the effects will be.

22. Interrupts

SB68k provides support for processing interrupts on the target system.

You can write interrupt service routines (ISRs) directly in SBasic68k, rather than having to drop down into assembly language for the target machine. However, you must declare a block of SB68k code as an ISR by using the INTERRUPT statement.

22.1. INTERRUPT statement

The INTERRUPT statement can accept a single argument, which is the address on the target system to use as an interrupt vector. SB68k will determine the address of the ISR code, then write that address to the vector address you specify.

Later, when your program is running on the target system, an interrupt will cause control to transfer to the address stored in the vector address. This in turn starts execution of your SB68k routine.

Example:

```
interrupt $80              ' use TRAP 0 as the interrupt vector
a = peekb(porta)           ' read 8 bits from port A
end                        ' return from the interrupt
```

Given the above example, an interrupt that uses $80 as its vector will cause control to jump to the statement containing the PEEKB() function. This small program will read a value from port A, then return from the interrupt.

Note that your SB68k routine does not get written to address $80; only the address of your routine gets written there. If necessary, examine the code created by the compiler to help understand how SB68k interrupts work.

22.2. END and RETURN statements

You must use an END statement to terminate all ISR code following an INTERRUPT statement. When it processes this END statement, SB68k compiles the proper Return from Exception instruction for the target system.

You sometimes need more than one exit from an ISR. If so, simply use the RETURN statement to exit the ISR. SB68k will automatically compile the proper instruction for leaving the interrupt section.

Example:

```
interrupt $fff0
if n = $66
     return
endif
n = $10
end
```

In the above example, control leaves the ISR immediately if N contains the value $66. If not, then N is changed to $10 and control leaves the ISR through the normal END statement.

In rare cases, you may need to combine a line label with the INTERRUPT statement. This can happen if your target system already has a monitor in ROM that has taken control of the interrupt vector area. In this case, you will likely need to prepare a jump table for your ISR, so the monitor can pass control to your ISR by jumping through the appropriate address in your table. To do this, you should combine a line label with use of the ADDR() function.

Example:

```
interrupt                ' note that no address is used!
timerisr:
if u <> 0                ' if u is not yet 0
     u = u - 1           ' decrement u
endif
pokeb timerreg, $40      ' rearm interrupt
end

main:
pokew memtable, $4ef9                ' write a JMP instruction to memory
poke memtable+2, addr(timerisr)      ' write addr of ISR
```

Here, the code in MAIN modifies the RAM addresses at MEMTABLE. It stores a JMP instruction ($4ef9) followed by the address of the jump target.

When the RTI interrupt occurs, the monitor will pass control to address MEMTABLE. The code left there by MAIN will in turn pass control to the ISR at RTIISR, where the actual interrupt processing occurs.

Note that the above example does not require an argument to the INTERRUPT statement. This means SB68k will not create an entry in the target's vector area. The above code, following the label MAIN, must be used to provide the target processor with access to the ISR.

If necessary, you can use the /i option on SB68k's command line to suppress generation of all interrupt vectors, including the reset vector. You can use this option if the target MCU already contains firmware for activating your program following reset.

23. INTERRUPTS statement

You can enable or disable system-wide interrupts by using the INTERRUPTS statement. This statement takes a single argument that is ON to enable interrupts or OFF to disable them. This statement only affects system-wide interrupts, and its exact implementation varies, based on the target system.

For the 68000 CPU, INTERRUPTS ON is compiled into an ANDI #$f8ff,SR instruction and INTERRUPTS OFF is compiled into a ORI #$0700,SR instruction.

Example:

```
INTERRUPTS ON       ' turn on system-wide interrupts
```

Additionally, you can follow the INTERRUPTS statement with a literal argument in the range of 0 to 7. The argument is used to set the value of the interrupt mask (IRM field) in the status register (SR), effectively setting the interrupt priority level.

Example:

```
INTERRUPTS  5       ' allow only interrupts at level 6 or higher
```

Note that using an argument of 0 to the INTERRUPTS statement permits all interrupts, while using an argument of 7 prevents all interrupts EXCEPT level 7 interrupts, which are non-maskable.

24. ORG statement

Normally, SB68k generates all code so it occupies sequential addresses on the target machine, starting at the address named CODEBEG. You can think of this range of addresses as SB68k's original code section.

If necessary, you can force SB68k to compile code at other addresses, by using the ORG statement. The ORG statement takes one of three forms:

```
    org  <addr>              or
    org  code                or
    org  <addr>  code
```

where <addr> is the address where you want subsequent SB68k code to compile. You can think of these other address ranges as alternate code sections. For example:

```
    org  $200
```

causes subsequent SB68k code to compile in an alternate section, starting at address $200. SB68k will continue to compile all code into sequential addresses, until you end the program or change the compile origin with another ORG statement. Note that any address you specify in an ORG statement must be word-aligned; that is, it must be an even address.

If you use the keyword CODE as the argument to an ORG statement, SB68k resumes compiling at the last address in the original code section.

Perhaps a larger example will clarify this. Assume that the following program was compiled with a CODEBEG address of $8000:

```
main:
    n = 14                  ' this code compiles at $8000

    org  $400               ' change the origin
table1:                     ' use a label at new origin
    datab  0,1,2,3          ' this code compiles at $400

    org  $500               ' change the origin
    interrupt $80           ' TRAP 0 ISR compiles at $500
    end                     ' bogus ISR, just for example

    org  code               ' return to original code section
    j = addr(table1)        ' sets j to $400
    end
```

You may use as many ORG statements, and change between alternate code sections and the original code section, as often as you want.

In rare cases, you might need to change the address of SBasic68k's code section inside your program. For example, you may need to force the compiler to generate a JMP instruction to a

vector area; this can happen if the JMP instruction must be in EPROM and thus cannot be modified at run-time. In such situations, you can use the third variation of the ORG statement above.

For example, the following code was compiled using a /c option of $f700:

```
    org  $8000  code          ' redefine code section here
main:
    n = 14                    ' this is the mainline code
    ...                       ' rest of mainline code goes here

    org  $14000               ' need to vector to an ISR
    asm                       ' switch to assembly language
    jmp  _rtiisr              ' use an assembly JMP instruction
    endasm                    ' back to SBasic

    org  code                 ' return to code section
    end                       ' end of program
```

The /cf700 option started compilation with the code section at $f700. This caused SB68k to write the startup library code at $f700. The ORG $8000 statement then moved MAIN down to $8000 and also caused the code section to move to that address. The rest of the mainline code (not shown here) compiled from there.

Next, the ORG $14000 writes a JMP instruction into the vector area for use by an ISR. Finally, the ORG CODE statement switched back to the code section, now somewhere above $8000.

This last step is important. SB68k automatically switches to the code section before adding any library files at the end of the compilation. If the code section had not moved to $8000, SB68k would have added the library files at $f700, which was the original code section. The resulting executable file would have failed.

25. The data stack

SB68k supports a data stack, for temporary storage of data. For the 68000 CPU, the data stack resides about 256 bytes below the return stack. Both stacks grow downward (towards lower addresses) as items are pushed onto them.

Do not confuse the use of these two stacks. The return stack holds all data used by the target MCU in servicing interrupts and subroutine calls. Items placed on the return stack are not currently accessible by your SB68k code.

The data stack, however, contains items explicitly placed by your program. You are free to use the data stack in any manner you like, and items placed on the stack will remain until your code specifically removes or modifies them.

25.1. PUSH statement

You can push items onto the data stack by using the PUSH statement. The PUSH statement takes a single argument; the 32-bit value of that argument will be pushed onto the data stack.

Example:

```
push n+5
```

adds 5 to the current value of N and pushes the sum onto the data stack. The value in N does not change.

The size of the data stack depends on where you place the return stack, using the /s option. The data stack will grow downward in memory until it runs into whatever values, if any, lie below it. There are no runtime checks for pushing too many items onto the data stack.

25.2. POP() and PULL()

You can pull (or pop) items from the data stack by using the POP() function. POP() returns the top (most recent) item pushed onto the data stack.

Example:

```
n = pop() + 5
```

pops the top item off the data stack, adds 5 to that value, and stores the sum in variable N.

There are no runtime checks for popping too many items from the data stack. The data stack resides below the return stack, and popping items causes the data stack pointer to move upwards in memory. If you pop more items from the data stack than you pushed onto it, you risk corrupting the return stack. This in turn will cause your program to crash on a later RETURN statement.

You can also use the PULL() function to remove items from the data stack. PULL() works exactly the same was as the POP() function; it is simply a synonym for POP().

25.3. PICK()

You can copy a value from within the data stack by using the PICK() function. PICK() locates a specific item within the data stack and returns that value.

Example:

```
n = pick(2)
```

copies the third item in the data stack into N. The size of the data stack does not change, and the value in the third item does not change.

Note that the item on the top of the stack is item 0; the second item is item 1. SB68k does not check to see how many items are actually on the data stack. If you supply an argument to PICK() that is larger than the current data stack, SB68k will return a bogus but legal value.

25.4. PLACE statement

You can alter a value within the data stack by using the PLACE statement. PLACE stores a 32-bit value into a specified item in the data stack.

Example:

```
place  2, n
```

stores the value in N into the third item in the data stack. The value in N does not change, and the size of the data stack does not change.

SB68k does not test the actual size of the data stack before executing the PLACE statement. Using PLACE to modify an item beyond the actual data stack will corrupt that location in memory and could crash your program.

You can combine PICK() and PLACE to create 32-bit variables local to a section of code, such as a subroutine. For example:

```
foo:
do
     place  0, pick(0) + 1     ' increment the item
loop while pick(0) < 5         ' loop until it hits 5
```

This code uses an item, already stored on the data stack by previous code, as a local variable. Changes to this variable occur only in the data stack, not in a DECLAREd variable.

25.5. DROP statement

In some cases, you want to remove items from the data stack without actually using the removed values. The DROP statement serves this purpose. It takes a single argument, which gives the number of items (NOT BYTES) to remove from the data stack. For the 68000, DROP removes four bytes for each value. For example:

```
drop  3              ' remove 12 bytes on a 68000
```

SB68k does not check that you are DROPping a valid number of items from the stack. If you DROP too many items, you risk corrupting the return stack, located immediately above the data stack on a 68hc11. If this happens, your program will likely crash.

25.6. SWAP statement

The SWAP statement makes it easier to manipulate items on the data stack. It exchanges the values in the topmost and second cells on the data stack.

Example:

```
push  2             ' first push a 2
push  3             ' stack has 3, then 2
swap                ' now stack has 2, then 3
```

Note that SWAP does not alter the size of the data stack, only the contents of the top two cells on the stack. SWAP always uses 32-bit cells.

Do not confuse the SWAPW() function, which exchange bytes within a variable, and SWAP, which exchanges cells on the data stack.

26. ASM and ENDASM statements

The ASM statement and ENDASM statement allow you to imbed assembly language source inside your SB68k program. Assembly language source lines between these two statements (with one exception) are not processed by SB68k; instead, they are passed directly to the output file for subsequent assembly.

This feature allows you to drop down into assembly language when necessary, should you need to write code that must run faster or take up less space. Additionally, imbedded assembly language gives you direct access to registers on the target system not currently supported by SBasic. For example, you can gain access to the 68000's hardware stack register (a7) by using imbedded assembly language.

The following example shows how to imbed 68000 assembly language:

```
foo:
asm                                 ' switch to assembly language
    movea.l    #IOREGS,a3           point A3 at i/o regs
    move.l     timer0(a3),d0        get 32-bit timer value
    move.l     d0,_time             save in variable TIME
endasm                              ' switch back to SBasic
return                              ' and return
```

This example shows an SB68k routine named FOO that uses imbedded assembly language to access some kind of timer. The value read from offset timer0 is stored in the SB68k variable TIME. The example then uses the ENDASM statement to switch back to SBasic68k, where the RETURN statement returns control to the calling routine.

Note that you will generally use an SB68k label at the start of an assembly section; other SB68k routines can use this label to pass control to your assembly section. Note also that imbedded assembly language lines can (and should) have comments appended to them.

With one exception, all source lines between an ASM statement and an ENDASM statement are passed unaltered to the output file for processing by the assembler. Thus, you cannot use SB68k statements or functions within an assembly section. Such statements or functions would be processed by the assembler, not by SBasic68k, and will result in assembler errors.

The sole exception to the above involves an underscore character. As shown in the example above, you can refer to SB68k variables, constants, or labels by prepending an underscore to the name. Before SB68k passes each assembly source line to the output file, it scans the line for any underscores. If SB68k detects an underscore, the following characters are parsed and tested against SBasic's list of known variables, constants, and labels. If found, the underscore and following characters are replaced with SBasic's internal name. Since this internal name is what the assembler will use to resolve operands, the SB68k name will be understood by the assembler. In the above example, SB68k would convert the string "_time" to something like "var009."

SB68k can handle multiple occurrences of underscores within a source line. For example, it will properly resolve a line such as:

```
move.l   #_main+2*_cons0         uses a label and a constant
```

If SB68k cannot resolve the character string following an underscore into the name of a variable, constant, or label, the line is passed unchanged to the output file. This will usually result in an assembler error message, but it will not cause an SB68k error! This means that if you use inline assembly language, you must check not only for SB68k errors but for assembler errors as well. SB68k will not know that the assembler could not resolve the label.

It is easy to lose track of which addresses are known to SB68k and which are known to the assembler. Remember that labels known to SB68k are automatically converted to an internal label before SB68k writes them to the output file. Thus, your SB68k source may refer to a label WAMPUM, but it will be converted to something like lbl010 before the assembler sees it.

To refer to standard assembler labels such as I/O registers, make sure that you make them known to the assembler by including them within an ASM section. The above example could have been written:

```
foo:
asm                                  ' switch to assembly language
timer0   equ         $20             offset to timer0
IOREGS   equ         $200000         address of I/O regs

         movea.l     #IOREGS,a3      point A3 at i/o regs
         move.l      timer0(a3),d0   get 32-bit timer value
         move.l      d0,_time        save in variable TIME
endasm                               ' switch back to SBasic
return                               ' and return
```

27. ASMFUNC statement

The ASMFUNC statement gives SB68k access to labels and routines within a block of assembly language code. See the section above on the ASM statement. The format for the ASMFUNC statement is:

```
asmfunc  foo
```

This statement tells the SB68k compiler that subsequent references to the label FOO must be passed to the output file as FOO, not as a converted label.

ASMFUNC adds tremendous power to SB68k, allowing you to write your own SB68k extensions in assembly language, then use them as if they were an integral part of SB68k. For example:

```
declare  stack              ' declare a variable
asmfunc  getstk             ' define an asm entry point

main:                       ' enter here
stack = getstk(0)           ' get addr of return stack
do  loop                    ' silly loop

asm                         ' switch to assembly language
getstk                      entry point to getstk()
   move.l      a7,d0        copy SP to D0
   rts                      return addr of stack
endasm                      ' back to SB68k
end
```

Here, the ASMFUNC statement tells SB68k that references to the label GETSTK are to be passed unchanged to the output file. Thus, when the SB68k code invokes the function GETSTK() to get the current hardware stack address, SB68k generates a JSR to GETSTK, not a JSR to an address with an internal SB68k label.

The actual code for subroutine GETSTK exists in the ASM block. GETSTK moves the stack pointer into the 68000's d0 and returns. The code generated by SB68k then stores the d0 into variable STACK and falls into the silly loop at the end of the program.

This example shows how to set up an ASMFUNC statement and its associated assembly language. It also shows how to use an ASMFUNC label as a function. In this case, you must adhere to SB68k's general rules regarding functions.

Functions, which return a result in the D0 register, must be called with an argument. Since the GETSTK routine doesn't need an argument, anything will work, but you must include an argument of some kind. That's why I show an argument of 0 for GETSTK.

You can also use ASMFUNC to create statements. Remember that an SB68k statement doesn't return a value, but simply performs an operation. For example:

```
asmfunc  setstk                 ' define an asm entry point
```

```
main:                                ' enter here
setstk  $006000                      ' change hardware stack addr
do  loop                             ' silly loop

asm                                  ' switch to assembly language
setstk                               entry point to setstk
   move.l      (a7)+,a3              get return addr from stack
   move.l      (a4)+,a7              change stack pointer
   jmp         (a3)                  return to SB68k
endasm                               ' back to SBasic
end
```

This example is more advanced and shows how to change the return stack pointer from inside SBasic. The SB68k program uses the SETSTK statement to set the return stack pointer to $6000, essentially moving the hardware stack. The tricky part here is that SB68k will execute this statement via a JSR to SETSTK. If the assembly language code simply moved the new stack pointer into the A7 register and returned, the program would crash since the return address would be undefined. The code above changes the A7 register, but saves the return address in the A3 register. It finally returns by jumping through the A3 register.

Here you can see how SB68k processes the arguments to a statement. The argument $006000 for the SETSTK statement is pushed onto SB68k's data stack; it is NOT passed in the d0 register. Thus, before the assembly language code in the SETSTK routine can do anything with the argument, it must first move the top item of the data stack into a spare address register. Refer to the GOSUB statement above for details on how SB68k parses arguments to statements.

Note that regardless of how you access the arguments passed, your assembly language routine MUST remove all arguments from the data stack before returning! If not, repeated invocations of your routine will eventually crash the target system.

This brings up another element of the ASMFUNC usage. SB68k does no error checking to make sure your program uses an ASMFUNC label consistently. Thus, you could use the same assembly language routine as both a function and a statement, if you so choose. What's more, you could pass varying numbers of arguments to the same ASMFUNC label as a statement. You are responsible for ensuring your assembly language routine behaves properly in all cases. SB68k will blindly load up the arguments and perform the JSR; your code has to deal properly with any variations in argument count.

Note that the second example is not completely general, since it can be called only from the top level (main) of your SB68k program. If, for example, you tried to do a SETSTK from within an SB68k routine, that routine would crash when it executed a RETURN statement, since its return address had moved when the stack pointer changed.

When writing assembly language code invoked with ASMFUNC labels, remember to preserve SB68k's registers. For the 68000, this includes a4 through a7. Additionally, any argument returned to the calling routine is passed back in register d0.

To summarize, you can use an ASMFUNC label as either a statement or a function. If used as a statement, you can supply zero, one, or more arguments; any arguments will be pushed left to right onto the data stack before the JSR to your label is executed. If used as a function, you MUST supply one and only one argument to the function. This argument will be passed to the called routine in register d0, though your routine can ignore it. Upon returning from your routine, the contents of register d0 will be used as the returned value of the function.

Additionally, remember that ASMFUNC labels do not exist as SB68k labels. You cannot do a GOSUB to an ASMFUNC label, since that label only exists in the assembly language module.

Since the ASMFUNC statement only affects subsequent references, the statement must appear in your source file before any references to the target label appear. Thus, it's best if you put all of your ASMFUNC statements near the beginning of your source file.

28. ASMFUNC, _INKEY, and _OUTCH

Generally, you should not use ASMFUNC to define labels used internally by SB68k run-time routines. Doing so will cause the assembler that processes the resulting output file to report an error. This happens because SB68k will create two identical labels in its output assembly language file, one label that you defined in your ASM section and a matching label in an SB68k library file.

The exceptions to this rule involve the labels _INKEY and _OUTCH. SB68k reserves these labels for internal routines that handle character I/O. By default, the _OUTCH routine sends the character in the low byte of register d0 to the hardware's serial port, and the _INKEY routine returns a character from the serial port in register d0. SB68k automatically appends a library file containing the assembly language source for these routines whenever it processes a statement that requires them.

In these two cases, SB68k permits your source file to override the normal inclusion of a library file. For example, if SB68k detects an ASMFUNC statement defining the label _OUTCH:

```
   asmfunc  _outch
```

SB68k does not append the _OUTCH library file. Instead, SB68k relies on whatever _OUTCH assembly language routine you define in your SB68k source file to handle all character output.

This feature allows you to redirect the output from all SB68k PRINT and OUTCH statements to an alternate device. Similarly, you can redirect the input for the SB68k INKEY() function from an alternate device. This makes it easy to add SB68k support for formatted output to devices such as LCDs, or character input from custom keyboards. For example:

```
asmfunc  _outch
asm
outch
    move.b  d0,$10040         send char in d0 to I/O port
    rts
endasm

main:
    print "2 + 2 ="; 2+3
    end
```

This sample program sends a mathematical statement to the I/O port that should convince anyone your computer is loony.

Note that the labels _OUTCH and _INKEY must appear within an ASM-ENDASM block. Also note that the argument to these ASMFUNC statements includes the assembly-language (underscored) version of the routine name, NOT the normal SB68k representation.

29. Character I/O on the 68000

The PRINT and OUTCH statements generate code for sending characters and text to some type of host, using library routines. For the 68000, this is usually hardware-dependent, based on the serial port device on your board.

Note, however, that SB68k does not actually set up the serial port for serial transfers. Thus, it is not usually enough to simply PRINT a string; your code must first (as a minimum) enable the serial port transmitter and set the port's baud rate.

This same requirement exists for the INKEY() function. Before your code can successfully invoke the INKEY() function, it must first enable the serial port's receiver and set the port's baud rate.

Refer to your target system's documentation for details on setting up the system's serial port. The OUTCH68K.LIB and INKEY68K.LIB files included in the standard SB68k distribution work with the Motorola M68332EVS evaluation system, and can serve as a starting point for your own code.