Chapter 9
Sundry Logical and Arithmetic Operations, including Multiplication and Division

```
6110 DEFine FuNction dis8$(pc)
6120 LOCal i,j,k,a$
6130 i=PEEK(pc)MOD 16
6140 j=PEEK(pc+1)DIV 64
6150 k=PEEK(pc+1)MOD 64
6160 pc=pc+2
6170 IF i MOD 2=0 THEN
6180 IF k DIV 8=1 THEN fault=1:RETurn ""
6190 SELect ON j
6200 =3:IF k=60 THEN
6210 pc=pc+2
6220 RETurn "DIVU    #$"&hexcon$(pc-2)&hexcon$(pc-1)&",D"&(i DIV 2)
6230 END IF
6240 RETurn "DIVU    "&adr$(k DIV 8,k MOD 8,pc)&",D"&(i DIV 2)
```

DIVU is divide unsigned. The 32 bit unsigned number in the destination data register is divided by the 16 bit unsigned source data item. If everything works out, the result is a 16 bit unsigned number in the low order word of the destination register, and the high order word contains the remainder. Or, in terms of SuperBASIC's **MOD** and **DIV** operators, the low order word contains the result of **DIV** and the high order word contains the result of **MOD**. Any attempt to divide by zero causes an immediate **TRAP** to vector 5, as described in Chapter 6. The status flags **Z** and **N** are set normally according to the value of the 16 bit **DIV** result, **C** is cleared, **X** is unaffected and **V** indicates overflow; but, as overflow may be detected part way through the division by the 68008, if **V** is set then the register contents and the other condition code flags are left in a relatively random and meaningless state.

If you are using **DIVU** with two 16 bit numbers, remember to clear the high order word of the destination data register otherwise the result will be incorrect.

DIV/MUL U/s source, Dd

1	DIV/MUL	0	0	d	U/s	1	1	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

SBCD/ABCD Ds, Dd

1	SBCD/ABCD	0	0	d	1	0	0	0	0	0	s

SBCD/ABCD –(As), –(Ad)

1	SBCD/ABCD	0	0	d	1	0	0	0	0	1	s

CMP.size source, Dd

1	0	1	1	d	0	SIZE	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

CMP.W/L source, Ad

1	0	1	1	d	W/L	1	1	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

CMPM.size (As)+, (Ad)+

1	0	1	1	d	1	SIZE	0	0	1	s

EXG Ds, Dd

1	1	0	0	s	1	0	1	0	0	0	d

EXG As, Ad

1	1	0	0	s	1	0	1	0	0	1	d

EXG Ds, Ad

1	1	0	0	s	1	1	0	0	0	1	d

TABLE 9.1 SUNDRY ARITHMETIC INSTRUCTIONS


```
6250 =0:a$="OR.B"
6260 =1:a$="OR.W"
6270 =2:a$="OR.L"
6280 END SELect
6290 RETurn a$&"    "&adr$(k DIV 8,k MOD 8,pc)&",D"&(i DIV 2)
6300 END IF
```

OR performs a bitwise OR of the source and destination data items, here the destination is always a data register. Address registers are not valid data sources. The **C** and **V** flags are cleared by the operation, **N** and **Z** are set according to the result, and **X** is unaffected.

```
6310 SELect ON j
6320 =3:IF k=60 THEN
6330 pc=pc+2
6340 RETurn "DIVS    #$"&hexcon$(pc-2)&hexcon$(pc-1)&",D"&(i DIV 2)
6350 END IF
6360 IF k DIV 8=1 THEN fault=1:RETurn ""
6370 RETurn "DIVS    "&adr$(k DIV 8,k MOD 8,pc)&",D"&(i DIV 2)
```

DIVS is identical in operation to **DIVU**, except that it treats the numbers involved as signed numbers rather than unsigned numbers. If you are dealing with 16 bit numbers you need to use **EXT.L** on the destination data register before dividing, to prevent errors due to the random contents of the high order word.

```
6380 =0:IF k<8 THEN
6390 RETurn "SBCD    D"&k&",D"&(i DIV 2)
6400 END IF
```

SBCD subtracts a binary coded decimal byte from another binary coded decimal byte, also subtracting the value of the **X** condition code from the result. The **C** and **X** flags are set if the operation requires a borrow from the next (more significant) binary coded decimal byte. The values of **V** and **N** are undefined and meaningless, and **Z** is cleared if the result is non-zero but unaffected if the result is zero. Thus **SBCD** is very similar in operation to **SUBX.B**.

```
6410 IF k<16 THEN
6420 RETurn "SBCD    -(A"&(k-8)&"),-(A"&(i DIV 2)&")"
6430 END IF
```

So, again, the most useful form scans two numbers held in memory by starting at the least significant byte. **X** should be cleared and **Z** set before starting an extended binary coded decimal subtraction using **SBCD**.

```
6440 a$="OR.B"
6450 =1:a$="OR.W"
6460 =2:a$="OR.L"
6470 END SELect
6480 IF k<16 OR k>=58 THEN fault=1:RETurn ""
6490 RETurn a$&"▲▲▲▲D"&(i DIV 2)&","&adr$(k DIV 8,k MOD 8,pc)
6500 END DEFine
```

This completes the **OR** command, which can thus operate where its destination is a data register, or its source is a data register, or the immediate data form shown in Chapter 4.

disB$** and **disC$ are quite similar to **dis8$** in organisation but, unfortunately, I could not find enough similarity to write a common function, and ended up writing separate versions for speed, both in the writing and in the operation of the program.

```
6510 DEFine FuNction disB$(pc)
6520 LOCal i,j,k,a$
6530 i=PEEK(pc)MOD 16
6540 j=PEEK(pc+1)DIV 64
6550 k=PEEK(pc+1)MOD 64
6560 pc=pc+2
6570 IF i MOD 2=0 THEN
6580 SELect ON j
6590 =3:IF k=60 THEN
6600 pc=pc+2
6610 RETurn "CMP.W▲▲▲#$"&hexcon$(pc-2)&hexcon$(pc-1)&",A"&(i DIV 2)
6620 END IF
6630 RETurn "CMP.W▲▲▲"&adr$( k DIV 8,k MOD 8,pc)&",A"&(i DIV 2)
```

CMP compares two items by subtracting the source from the destination and setting the **C**, **V**, **N** and **Z** flags according to the result.

X is unaffected, and the result is discarded rather than being placed in the destination.

```
6640 =0:IF k DIV 8=1 THEN fault=1:RETurn ""
6650 a$="CMP.B"
6660 =1:a$="CMP.W"
6670 =2:a$="CMP.L"
6680 END SELect
6690 RETurn a$&"▲▲▲"&adr$(k DIV 8,k MOD 8,pc)&",D"&(i DIV 2)
6700 END IF
6710 SELect ON j
6720 =3:IF k=60 THEN
```



```
6730 pc=pc+4
6740 RETurn "CMP.L▲▲▲#$"&hexcon$(pc-4)&hexcon$(pc-3)&hexcon$(pc-2)&he
     xcon$(pc-1)&",A"&(i DIV 2)
6750 END IF
6760 RETurn "CMP.L▲▲▲"&adr$(k DIV 8,k MOD 8,pc)&",A"&(i DIV 2)
```

You are only allowed to compare things with data or address registers, or using the variation of **CMP** in Chapter 4, you can compare immediate data with any addressable item.

```
6770 =0:a$="B"
6780 =1:a$="W"
6790 =2:a$="L"
6800 END SELect
6810 IF k>=58 THEN fault=1:RETurn ""
6820 IF k DIV 8=1 THEN RETurn "CMPM."&a$&"▲▲(A"&(k-8)&")+,(A"&(i DIV
     2)&")+"
```

CMPM stands for compare multiple or company memory and it allows two strings, or binary coded decimal numbers, or extended numbers to be compared with each other starting at the most significant end. Flags are set as for **CMP** rather than as for an extended arithmetic command; so, **X** is unaffected, and **Z** should be tested after every comparison, rather than at the end of the string or number.

```
6830 RETurn "EOR."&a$&"▲▲▲D"&(i DIV 2)&","&adr$(k DIV 8,k MOD 8,pc)
6840 END DEFine
```

EOR performs an Exclusive OR with a data register as source and another data register or memory item as destination. The **C** and **V** flags are cleared, **N** and **Z** are set according to the result, and **X** is unaffected.

```
6850 DEFine FuNction disC$(pc)
6860 LOCal i,j,k,a$
6870 i=PEEK(pc)MOD 16
6880 j=PEEK(pc+1) DIV 64
6890 k=PEEK(pc+1) MOD 64
6900 pc=pc+2
6910 IF i MOD 2=0 THEN
6920 IF k DIV 8=1 THEN fault=1:RETurn ""
6930 SELect ON j
6940 =3:IF k=60 THEN
6950 pc=pc+2
6960 RETurn "MULU▲▲▲#$"&hexcon$(pc-2)&hexcon$(pc-1)&",D"&(i DIV 2)
6970 END IF
6980 RETurn "MULU▲▲▲"&adr$(k DIV 8,k MOD 8,pc)&",D"&(i DIV 2)
```

MULU performs an unsigned multiplication of the 16 bit source data with the low order word of the destination data register, placing the 32 bit result in the destination data register. Overflow is not possible, so the **V** and **C** flags are cleared. **N** and **Z** are set normally according to the 32 bit result, and **X** is unaffected.

```
6990 =0:a$="AND.B"
7000 =1:a$="AND.W"
7010 =2:a$="AND.L"
7020 END SELect
7030 RETurn a$&"▲▲▲"&adr$(k DIV 8,k MOD 8,pc)&",D"&(i DIV 2)
7040 END IF
```

AND performs a bitwise AND of source and destination data items, clearing the **C** and **V** flags, setting **N** and **Z** according to the result and leaving **X** unaffected. The various forms of **AND** mirror the available forms of **OR** which we have already seen.

```
7050 SELect ON j
7060 =0:IF k<8 THEN RETurn "ABCD▲▲▲▲D"&k&",D"&(i DIV 2)
7070 IF k<16 THEN RETurn "ABCD▲▲▲▲-(A"&(k-8)&"),-(A"&(i DIV 2)&")"
```

OR.size source, Dd

1	0	0	0	d	0	SIZE	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

OR.size Ds, destination

1	0	0	0	s	1	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

EOR.size Ds, destination

1	0	1	1	s	1	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

AND.size source, Dd

1	1	0	0	d	0	SIZE	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

AND.size Ds, destination

1	1	0	0	s	1	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

TABLE 9.2 SUNDRY LOGICAL INSTRUCTIONS

ABCD, add binary coded decimal, mirrors **SBCD**, adding together two binary coded decimal bytes and the value of the **X** flag, setting the **C** and **X** flags according to the result, leaving **N** and **V** in an undefined state, clearing **Z** if the result is non-zero, but leaving it unaffected otherwise.

```
7080 a$="AND.B"
7090 =1:IF k<8 THEN RETurn "EXG▲▲▲▲▲D"&(i DIV 2)&",D"&k
7100 IF k<16 THEN RETurn "EXG▲▲▲▲▲A"&(i DIV 2)&",A"&(k-8)
```

EXG exchanges the full 32 bit contents of two registers without affecting any condition code flags.

```
7110 a$="AND.W"
7120 =2:IF k<8 THEN fault=1:RETurn ""
7130 IF k<16 THEN RETurn "EXG▲▲▲▲▲D"&(i DIV 2)&",A"&(k-8)
```

This is, perhaps, the most useful version of **EXG**, allowing the exchanging of a data register and an address register. Assemblers often allow the address register to be named as the source, but such commands are translated to the code above where the data register is nominally the source. As the operation works symmetrically on source and destination, this does not matter.

```
7140 a$="AND.L"
7150 =3:IF k=60 THEN
7160 pc=pc+2
7170 RETurn "MULS▲▲▲▲#$"&hexcon$(pc-2)&hexcon$(pc-1)&",D"&(i DIV 2)
7180 END IF
7190 IF k DIV 8=1 THEN fault=1:RETurn ""
7200 RETurn "MULS▲▲▲▲"&adr$(k DIV 8,k MOD 8,pc)&",D"&(i DIV 2)
7210 END SELect
```

MULS is identical to **MULU**, except that it treats the source data items as being signed numbers.

```
7220 IF k>=58 THEN fault=1:RETurn ""
7230 RETurn a$&"▲▲▲D"&(i DIV 2)&","&adr$(k DIV 8,k MOD 8,pc)
7240 END DEFine
```

These lines complete the decoding of **AND** instructions where the source is a data register and the destination is a memory item, whose various different sized versions were scattered through the last **SELec**tion.

Chapter 10
Shifts and Rotates

A complicated collection of shifting and rotating operations are fitted into **disE$**. The operations either work directly on a 16 bit word in memory, when the data is moved by just one bit, or they work on the contents of a data register. When the operations work on data registers, a byte, word or longword can be affected and the distance the item is moved can be specified as immediate data in the range 1 to 8, or by the contents of a data register, when the value is taken **MOD 64**.

```
7250 DEFine FuNction disE$(pc)
7260 LOCal i,j,k,a$
7270 i=PEEK(pc)MOD 16
7280 j=PEEK(pc+1)DIV 64
7290 k=PEEK(pc+1)MOD 64
7300 pc=pc+2
7310 SELect ON j
7320 =3:IF k<16 OR k>=58 THEN fault=1:RETurn ""
7330 RETurn shift$("  ",i)&adr$(k DIV 8,k MOD 8,pc)
```

When a word in memory is shifted, program counter relative addressing cannot be used.

```
7340 =0:a$=".B"
7350 =1:a$=".W"
7360 =2:a$=".L"
7370 END SELect
7380 j=k DIV 8 MOD 4*2+i MOD 2
```

In the data register movement versions of the shift commands, the type of shift is specified by different bits in the instruction to those used when shifting memory words. The above calculation picks up the separate bits and recombines them to form a bit pattern which correctly reflects the types of shift to be decoded by **shift$**.

Shift R/L.size #q, Dd

1	1	1	0	q	MOD 8	R/L	SIZE	0	SHIFT	d

Shift R/L.size Dn, Dd

1	1	1	0	n	R/L	SIZE	1	SHIFT	d

Shift R/L destination

1	1	1	0	0	SHIFT	R/L	1	1	DESTINATION ADDRESSING MODE	DESTINATON REGISTER NUMBER

SHIFT:				
	0	0	=	AS
	0	1	=	LS
	1	0	=	ROX
	1	1	=	RO

TABLE 10.1 SHIFTS AND ROTATES

```
7390 IF k<32 THEN
7400 IF i<2 THEN i=16
7410 RETurn shift$(a$,j)&"#"&(i DIV 2)&",D"&(k MOD 8)
7420 END IF
7430 RETurn shift$(a$,j)&"D"&(i DIV 2)&",D"&(k MOD 8)
7440 END DEFine
```

A zero bit pattern for the three bits of immediate data indicates a shift by 8 bits in the same way that **ADDQ** and **SUBQ** can add and subtract a value of 8.

```
7450 DEFine FuNction shift$(a$,n)
7460 SELect ON n
7470 =0:RETurn "ASR"&a$&"   "
```

ASR stands for arithmetic shift right. Arithmetic shifts are designed to work on signed numbers, trying to maintain the sign of the operand; so, the sign bit is copied back into its old position as the data is shifted right. The last bit shifted out of the least significant bit of the item is copied into both **C** and **X**. The flags **N** and **Z** are set according to the value of the result. As overflow cannot occur, **V** is cleared by the operation.


```
7480 =1:RETurn "ASL"&a$&"   "
```

ASL is arithmetic shift left. Bits introduced at the least significant end of the operand are zero. **C** and **X** keep a copy of the last bit shifted out of the operand, **N** and **Z** are set according to the value of the result, and **V** is set if the sign changes, recording any signed arithmetic overflow.

```
7490 =2:RETurn "LSR"&a$&"   "
```

LSR is logical shift right. This is similar to **ASR**, but the new bits introduced at the left of the item are always zero, rather than copies of the sign bit. This means that the **N** flag is always cleared by the operation. All other flags are set as for **ASR**.

```
7500 =3:RETurn "LSL"&a$&"   "
```

LSL, logical shift left, is almost identical with **ASL** with zero bits being introduced from the right, except that **V** is always cleared by the operation, as it is not considered to be dealing with numeric quantities.

```
7510 =4:RETurn "ROXR"&a$&"  "
```

ROXR is rotate extended right. The word extended in the name implies that the **X** bit forms an extension of the item being rotated, linking the most and least significant bits of the item. Bits introduced at the left of the item come from **X**, and bits leaving the right go into **X**. The value of **C** copies **X**, **N** and **Z** are set normally, and **V** is always cleared.

A **ROXR** by 9 bits would be needed to return a byte item to its original position, 17 for a word, 33 for a longword.

```
7520 =5:RETurn "ROXL"&a$&"  "
```

ROXL, rotate extended left, is the mirror image of **ROXR**, and its condition code flags are set according to the same rules as **ROXR**.

```
7530 =6:RETurn "ROR"&a$&"   "
```

ROR, rotate right, does not affect the **X** bit, nor is it affected by the **X** bit. Instead, bits leaving the right end of the item immediately reappear at the left end, forming the bit that is introduced there. **C** records the value of the last bit transferred from the extreme right to the extreme left. **N** and **Z** are set according to the value of the result, and **V** is cleared.

```
7540 =7:RETurn "ROL"&a$&"   "
```

ROL, rotate left, is the mirror image of **ROR**, and it sets the condition code flags in the same manner, **X** is unaffected, **C** records the last bit transferred from one end to the other, this time from extreme left to extreme right, **N** and **Z** are set according to the value, and **V** is cleared.

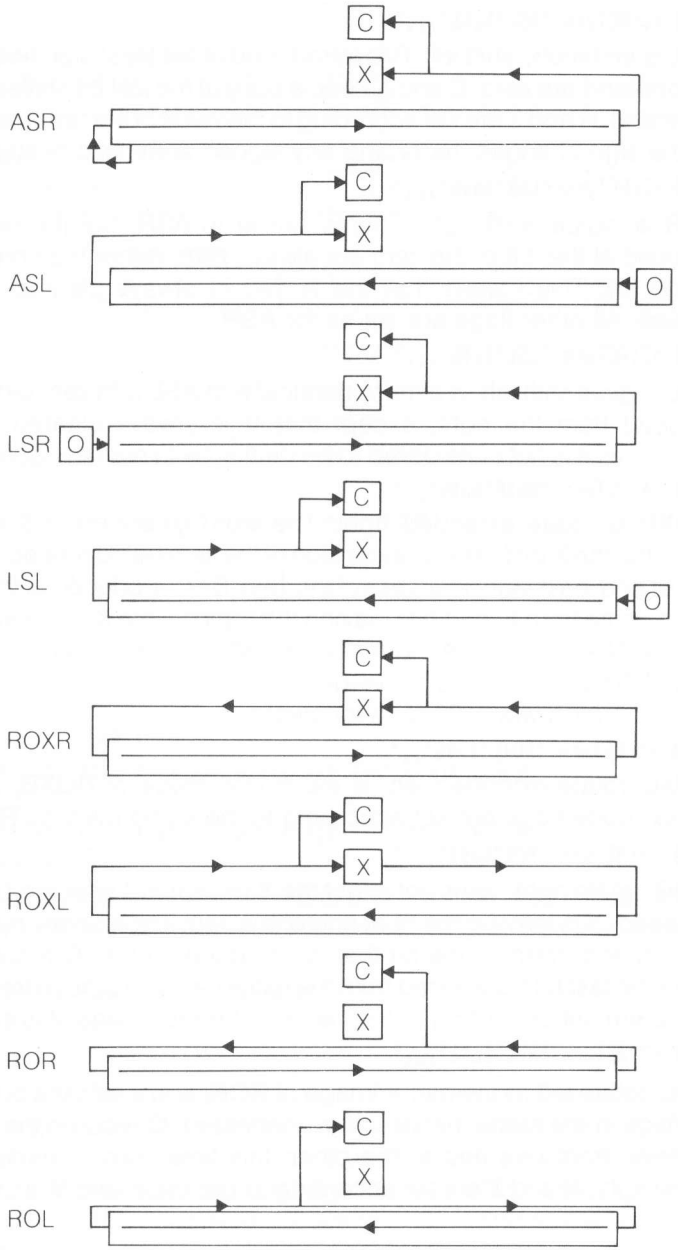

TABLE 10.2 CARRY AND EXTENDED ARITHMETIC BITS SHIFTED AND ROTATED


```
7550 =REMAINDER :fault=1:RETurn ""
7560 END SELect
7570 END DEFine
7580 REMark - End of Disassembler
```

If you have not been trying out the disassembler, or its individual functions, as you have typed it in, now is the time to try it out. Perhaps the easiest place to start is with the first 8 bytes of the ROM, which define the initial stack pointer and program counter contents. With this information you can follow the initialisation routines, such as the QL's extensive memory testing process that it goes through before any other action. Whatever you decide to disassemble, I wish you good luck in deciphering what the code does.

Chapter 11
Practical Use of the Assembler

When you run the assembler listed in Appendix D it shows a status page, and in the channel 0 area asks you to key a letter to select "Assemble", "Edit" or "Load code". Figure 11.1 shows the layout. As you have not yet written any 68000 programs, the only sensible choice is to press <E> for "Edit". The other choices can work from files on microdrive so, if a friend had given you copies of programs he had written with this assembler, you could make the other choices.

Again, the editor shows a menu in the channel 0 window. This time it is usual to press <A> for "Append", to add program lines to the end of any you typed in before. **getline** asks you for the "label", "command" and "parameters" in turn for each line of program. To leave the label blank, simply press <ENTER>. To leave this line entry mode, enter a blank command by pressing <ENTER> when asked for a "command". Try typing a few lines of program that might come to mind from the

```
68000 Assembler
© 1984 Alan Giles - Version AH

Current program size 0 lines
Code start 0
Space reserved 0 bytes
Space used 0 bytes
Spare space 21144 bytes


A=Assemble E=Edit L=Load code
```

FIGURE 11.1

disassembler discussion. Perhaps you might type something like:

```
start   movem.l d0-d1/a0,-(sp)
        move.l  #$20000,a0
        move    #8191,d0
        moveq   #0,d1
loop    move.l  d1,(a0)+
        dbra    d0,loop
        movem.l (a7)+,d0-d1/a0
        rts
```

which, according to the QL memory map, ought to clear the screen memory by storing 8 192 longword zeroes in the appropriate locations. Note that the names "sp" and "a7" are interchangeable, and that you can use lower case letters rather than capital letters if you prefer to do so. Figure 11.2 shows the screen at this point.

Press <ENTER>, <ENTER> to leave line entry mode, then press <X> to leave **editor** and choose <A> to assemble the current program. For many of the questions the program asks, just pressing <ENTER> will be taken as the normally expected answer. The program should assemble without any trouble. However, do not run it yet, as we have left some problems to overcome.

```
0000    start   movem.l d0-d1/a0,-(sp)
0001            move.l  #$20000,a0
0002            move    #8191,d0
0003            moveq   #0,d1
0004    loop    move.l  d1,(a0)+
0005            dbra    d0,loop
0006            movem.l (a7)+,d0-d1/a0
0007            rts
0008 +          END


A=Append D=Delete I=Insert L=Load
M=Merge R=Replace S=Save X=eXit
```

FIGURE 11.2

You may have noticed in the assembler listing that the procedure **usr**, which is called in line 640, is never actually defined. This is due to a bug in version "JM" and earlier versions of the QL ROM.

We would like to use **CALL** in place of **usr**, and the **RTS** at the end of the program we typed in reflects this desire, but although **CALL** works when there is only a small program in the QL's memory, such as during loading of one of the Psion packages, it fails when a program as large as the assembler is in memory, apparently **CALL**ing the wrong address and crashing. If you have **SAVE**d copies of both the assembler and the short assembly language program you wrote, you could add the lines:

```
20000 DEFine PROCedure usr(address)
20010 CALL address
20020 END DEFine
```

and then re**RUN** the assembler, and re**LOAD** the assembly language program, or assemble it directly from its microdrive file. In the latter case, you tell the computer that you have finished the list of files to be assembled by just pressing <ENTER> when asked for the next drive number. Once assembled, you can try **RUN**ning the program, to see if your QL has the **CALL** bug.

If the program works, it will clear the screen and then give the error message **bad parameter**. This is because of something I forgot to mention about **CALL**, it takes the value in D0 when you **RTS** as an error code, and on entry **CALL** has the code for **bad parameter** in D0 so that this is an easy error message for your program to return. The error codes are usually negative, minus 1 means **not complete**, minus 2 means **invalid job** and so on, zero means no error, and positive numbers are taken as addresses pointing to your own error message text, which should consist of a word giving the length of the following message in bytes, followed by the message text. So, you might like to add **MOVEQ #0,D0** just before the **RTS** in our screen clearing program.

If **CALL** failed, there are various ways of overcoming the problem. The simplest method is to remove the assembler from memory before manually typing your **CALL** command. You could add the line:

```
20010 NEW
```

in place of the previous version and, provided you remember the address where your program is assembled, you will be able to **CALL** it yourself. This approach also allows you to pass parameters to your program, as indicated in the QL keyword manual.

You can pass up to thirteen parameters to your machine code

program by listing them after the address in a **CALL** command, separated by commas. The first parameter is placed in D1 as a 32 bit signed number, and subsequent parameters are placed in D2 to D7 and then A0 to A5. As we saw, D0 has a special use as the error code, A7 is the stack pointer, and A6 seems to be used to point to SuperBASIC's system variables.

However, this **NEW**ing of the assembler is not a very neat approach, as the reloading of the assembler takes quite a while, due to the time taken to convert the textual form of the program, as stored on microdrive, into the internal representation used by SuperBASIC to enable the assembler to run quickly.

It is possible to define your own machine code procedures for use on the QL, and the following program, written by Dr. Ian Logan, defines a machine code command **USR**. This command is a simplified version of **CALL**, which takes no parameters apart from the routine address.

```
100 a=RESPR(68):RESTORE
110 FOR n=0 TO 67
120 READ v:POKE a+n,v
130 END FOR n
140 DATA 67,250,0,8,52,120,1,16,78,210
150 DATA 0,1,0,12,3,85,83,82,0,0
160 DATA 0,0,0,0,75,235,0,8,52,120
170 DATA 1,24,78,146,36,118,152,0,72,231
180 DATA 255,254,67,250,0,20,34,143,78,146
190 DATA 67,250,0,12,46,81,76,223,127,255
200 DATA 112,0,78,117,0,0,0,0
210 CALL a
220 LRUN mdv1_asm
```

This program defines the command **USR** and then loads the assembler into memory, from the file **mdv1_asm**. The assembler has to be loaded as a separate program after **USR** is defined in order that the program line which invokes **USR** does invoke the machine code procedure rather than a SuperBASIC procedure. This program also has the advantage of being small enough for **CALL** itself to work.

The machine code contained in the **DATA** statements of the program is the assembled version of the following:

```
start   lea     usr,a1
        move.w  $110,a2
        jmp     (a2)
usr     dc.w    1
        dc.w    12
        dc.b    3
        dc.b    'USR'
        dc.w    0,0,0
code    lea     8(a3),a5
        move.w  $118,a2
        jsr     (a2)
        move.l  0(a6,a1.l),a2
        movem.l d0-d7/a0-a6,-(a7)
        lea     store,a1
        move.l  a7,(a1)
        jsr     (a2)
        lea     store,a1
        move.l  (a1),a7
        movem.l (a7)+,d0-d7/a0-a6
        moveq   #0,d0
        rts
store   dc.l    0
```

The routine **start** simply adds the table **usr** to the table of SuperBASIC procedures, causing the QL to invoke the routine **code** whenever **USR** is used. This routine takes the parameter to **USR**, which is a floating point number, converts it to a 32 bit number and calls the routine at that address, being careful to save all the registers, even the stack pointer, to ensure that the return to SuperBASIC is safely accomplished. You may like to disassemble the routines addressed by the words in locations $110 and $118 to discover how they work.

Another way to overcome the **CALL** problem would be to use the remaining SuperBASIC command which can invoke machine code routines, namely **EXEC**. **EXEC** always needs the machine code program to be specially saved in a microdrive file. If we return to the SuperBASIC routine **usr**, we could replace line 20010 by:

```
20010 EXEC_W mdv1_temp
```

To do this successfully, we also need to add:

```
12845 SEXEC mdv1_temp,codeaddr,codesize,codespace-codesize
```

Note that we use **EXEC_W** rather than plain **EXEC**, this means that SuperBASIC waits for the machine code program to finish before carrying on. Otherwise, if we used **EXEC** by itself, both programs would run on the QL time-sharing the available processing power. This would usually be quite confusing, as both programs might write conflicting instructions on the screen but, if carefully controlled, it can provide a useful feature.

One problem with **EXEC** is that you cannot **RTS** from a job created by **EXEC**, as it is a separate job and has nowhere to return to. Instead, you need to tell QDOS to terminate the job, and to do this you might add the following code to the beginning of a routine which works with **CALL**:

```
bsr.s   start
move.l  d0,d3
moveq   #-1,d1
moveq   #5,d0
trap    #1
```

provided the **CALL**ed routine starts with the label **start**. Note that the error code from a job is passed to QDOS in D3 rather than D0.

Separate jobs also use separate stack space, so you should ensure that you specify enough spare bytes to more than cope with the space that you need when you assemble a program.

To convert fully to using **EXEC** rather than **CALL**, you will need to change lines 580 and 710 in the assembler to use **EXEC** and **SEXEC** rather than **LBYTES** and **SBYTES**, and you might like to replace line 130 by a line which sets up **prog$** as our five lines of QDOS job removing code, perhaps by loading them from microdrive.

When a job is started by **EXEC** the values of A4, A5 and A6 are set on entry as follows:
(A6) points to the first byte of code, the start of the area reserved for the program.
(A6, A4) points to the first byte available for data, the first byte beyond the assembled program.
(A6, A5) points to the last byte available for data, which is also the first byte used by the stack in its descent from the top of the data area, the stack already contains a longword zero as the last item on entry to your program.

Figure 11.3 shows a listing of an alternative procedure **usr** which allows you to choose whether to use **CALL** or **EXEC_W**.

Another change you might like to make to the assembler, if you have a printer, is to divert the assembly listing output to your printer by **OPEN**ing a channel, say channel 5, to it and altering the **PRINT** statements from line 12700 onwards to use channel 5 in place of the default channel.

One thing to watch out for, if you start writing long assembly language programs, is the assembler running out of memory. However, if this does happen, the facility allowing you to perform a combined assembly from a number of files on microdrive should help. When you run out of memory, simply save what program you have already written and start


```
20000 DEFine PROCedure usr(address)
20010 CLS:CLS#0:CSIZE 2,0
20020 PRINT "This is where we have problems. We"!"ought to be able to
      CALL the machine"!"code routine, but this tends to"!"crash if
      the Assembler is still in"!"memory. So, choose your option:-"
20030 PRINT \\"1 - Try CALL anyway"\\"2 - NEW and then CALL manually"
      \\"3 - Use SEXEC and EXEC_W"
20040 PRINT \\"CALLed routines should set D0 to 0"!"before returning
      using RTS."\\"EXECuted routines should set D0 to 5, D1 to -1 and
      D3 to 0 before"!"returning"!"using TRAP #1."
20050 a$=INKEY$(-1)
20060 a="0"&a$
20070 SELect ON a
20080 =1:CALL address
20090 =2:NEW
20100 =3:INPUT#0;\"Drive number"!d$\"File name"!name$:d="0"&d$:IF d=0
      THEN RETurn
20110  CLS#0:INPUT#0;\"Data size needed, including stack"!"space and
       4 extra bytes:"!datasize$:datasize="0"&datasize$
20120  DELETE "mdv"&d$&"_"&name$
20130  SEXEC "mdv"&d$&"_"&name$,codeaddr,codesize,datasize
20140  EXEC_W "mdv"&d$&"_"&name$
20150 END SELect
20160 END DEFine
```

FIGURE 11.3

afresh with the rest of the program. When it comes to actually assembling all the files together, the assembler will even manage to resolve references to labels in different files; so your program size limit now comes when all your microdrives are full of files. After that point, to produce larger programs, you will have to develop programs in separate modules without any cross module label resolution.

Your task now is to become accustomed to using the various 68000 commands, and I would suggest that you might go through the list checking that each does what you expect. The 68008 is a challenging processor with a lot of clever features, and I am sure you will have a lot of pleasure writing programs for it.