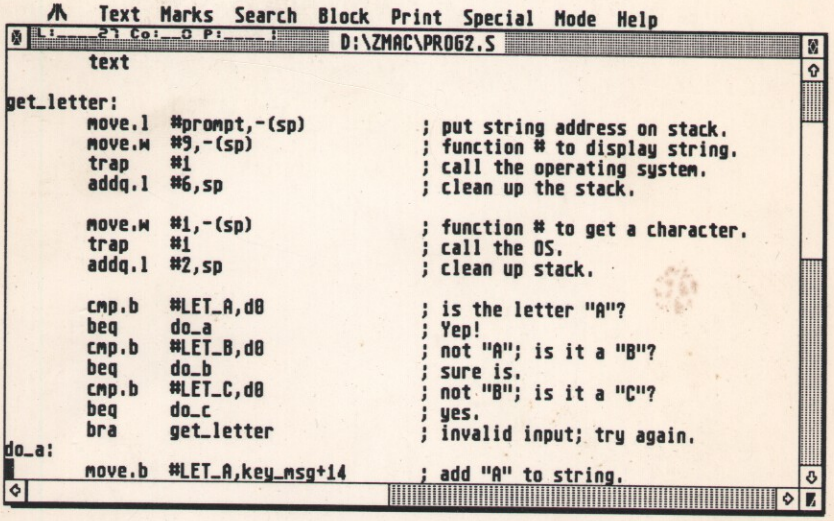
The ST Assembly Language Workshop

by Clayton Walnum



```
        text

get_letter:
        move.l  #prompt,-(sp)               ; put string address on stack.
        move.w  #9,-(sp)                    ; function # to display string.
        trap    #1                          ; call the operating system.
        addq.l  #6,sp                       ; clean up the stack.

        move.w  #1,-(sp)                    ; function # to get a character.
        trap    #1                          ; call the OS.
        addq.l  #2,sp                       ; clean up stack.

        cmp.b   #LET_A,d0                   ; is the letter "A"?
        beq     do_a                        ; Yep!
        cmp.b   #LET_B,d0                   ; not "A"; is it a "B"?
        beq     do_b                        ; sure is.
        cmp.b   #LET_C,d0                   ; not "B"; is it a "C"?
        beq     do_c                        ; yes.
        bra     get_letter                  ; invalid input; try again.
do_a:
        move.b  #LET_A,key_msg+14           ; add "A" to string.
```

A Taylor Ridge Book

The ST Assembly Language Workshop

Volume 1

The ST Assembly Language Workshop

Volume 1

Clayton Walnum

Taylor Ridge Books
Manchester, Connecticut

Published by Taylor Ridge Books, P.O. Box 78, Manchester, CT 06045.

This book was produced on an Atari Mega 4 workstation, using *Calamus* desktop publishing software.

Designed and Illustrated by

Bryan Schappel

Cover Artwork Designed by

Clayton Walnum & Bryan Schappel

Reference Section Compiled and Written by

Bryan Schappel

Second Edition

Printed in the United States of America.

To Charles F. Johnson,

One of the best programmers and musicians I know.

Table Of Contents

Introduction *xiii*

- Who This Book is For xiii
- Which Assembler? xiv
- A Multi-Volume Work xiv
- How To Use This Book xv
- The Workshop Begins xvi

Chapters

1 A First Look At Assembly Language 1

- The Answers 1
- Language Types 2
- Our Own Machine Language 3
- Machine Language to Assembly Language 4
- Assemblers and Housekeeping 5
- Registers and Memory 7
- The Program Counter 8
- The Stack 11
- Let's Get Busy 14
- Conclusion 17
- Summary 17
- Program Listing #1 19

2 Data Sizes And Number Systems 21

- Data Sizes 21
- Bits, Bytes, Words, and Long Words 22
- Binary, Hexadecimal, and Decimal Numbers 24
- Number Systems 24
- Hexadecimal Digits 26
- Converting Binary to Hexadecimal 26
- Converting from Hexadecimal to Binary 27
- Converting Decimal to Binary 28
- Converting Decimal to Hexadecimal 30
- The Easy Way 31
- Conclusion 31
- Summary 31

3 *Registers, The Stack, And Addressing Modes* **33**

Registers 33
The Program Counter 34
The Status Register 34
The Stack and ST Function Calls 35
Addressing Modes 36
Instruction Suffixes 37
Conclusion 38
Summary 39

4 *Constants, Function Calls, And Branching* **41**

The Program 41
The Data Segment 42
The Text Segment 44
Immediate Addressing 45
And Back to the Program 45
Address Register Indirect Addressing with Pre-decrement 45
The Way It Works 46
The Function *Cconws* 47
The *addq* instruction 48
Address Register Direct Addressing 48
The *Cconin* Function 48
Comparisons 49
Data Register Direct Addressing 51
String Manipulation 51
Absolute Addressing 52
Finishing It Up 52
Conclusion 53
Summary 53
Program Listing #2 55

5 *Numbers* **59**

A Number in Memory 59
Converting Numbers 60
The Program 61
Verifying the Input 62
Saving a Register 63
Simple ASCII to Integer Conversions 63
Simple Integer to ASCII Conversions 64
Programming Style 65
Conclusion 66
Summary 66
Program Listing #3 68

6 *Subroutines And Looping* 71
The Program 71
String Arrays 72
Subroutines 73
Saving Registers 74
Looping 76
Address Register Indirect Addressing with Post-Increment 77
Restoring the Registers 79
Subroutine Design 79
Commenting a Subroutine 80
Conclusion 80
Summary 80
Program Listing #4 82

7 *Numbers Revisited* 85
The Program 85
Uninitialized Data Storage 86
Generating Random Numbers 87
16-bit Integer to ASCII 90
Local Labels 92
ASCII to 16-bit Integer 93
The Main Program 95
Conclusion 96
Summary 96
Program Listing #5 99

8 *Manipulating The System Clock* 109
The Program 109
A First Look at Macros 110
Getting the System Date 112
Getting the System Time 116
Setting the System Date 117
Setting the System Time 121
Address Register Indirect Addressing with Displacement 122
Address Register Indirect Addressing with Index 123
Indexing an Address Table 124
Conclusion 125
Summary 125
Program Listing #6 127

9 *Simple File Handling And Printer Output* 145
The Program 145
More About Macros 146
A String-Input Macro 147
A Get-Character Macro 148
Opening a Disk File 148
Checking the Printer's Status 149
Reading a File 150
Why a Buffer? 153
Conclusion 153
Summary 154
Program Listing #7 156

10 *More File Handling* 165
The Program 165
Some New Macros 166
Creating a File 166
Writing to a File 167
High-Level Assembly Language 168
Inside the Program 169
Conclusion 171
Summary 171
GEMDOS Error Codes 172
Program Listing #8 173

11 *Playing With the Screen* 187
The Program 187
DEGAS File Format 188
How the Program Works 189
Getting the System's Resolution 189
Calculating Screen Memory 190
Getting the Physical Screen's Address 192
Manipulating Palettes 193
Flipping Screens 195
Conclusion 197
Summary 198
That's All Folks 199
Program Listing #9 200

A *Appendix • Assembly Instructions* 213
Assembling with DevPac2 213
Assembling with MadMac 215
Changes for MadMac 215

B 68000 Instruction Reference 217

Abbreviations 218

Addressing Modes 218

ABCD • Add Binary Coded Decimal with Extend 219

ADD • Add Binary 219

ADDA • Add Address 220

ADDI • Add Immediate 220

ADDQ • Add Quick 221

ADDX • Add Binary with Extend 221

AND • Logical AND 222

ANDI • AND Immediate 222

ANDI • AND Immediate Data to CCR 223

ANDI • AND Immediate Data to SR 223

ASL • Arithmetic Shift Left 224

ASR • Arithmetic Shift Right 224

Bcc • Branch on Condition Code 225

BCHG • Test a Bit and Change 226

BCLR • Test a Bit and Clear 226

BSET • Test Bit and Set 227

BSR • Branch to Subroutine 227

BTST • Test Bit 227

CHK • Check Register Against Boundaries 228

CLR • Clear an Operand 228

CMP • Compare 229

CMPA • Compare Addresses 229

CMPI • Compare Immediate 230

CMPM • Compare Memory 230

DBcc • Test, Decrement, and Branch 231

DIVS • Signed Divide 232

DIVU • Unsigned Divide 232

EOR • Exclusive OR 233

EORI • Exclusive OR Immediate 233

EORI • Exclusive OR Immediate Data to CCR 234

EORI • Exclusive OR Immediate Data to SR 234

EXG • Exchange Registers 234

EXT • Sign Extend 235

ILLEGAL • Take Illegal Instruction Trap 235

JMP • Jump 235

JSR • Jump to Subroutine 236

LEA • Load Effective Address 236

LINK • Link and Allocate Space 237

LSL • Logical Shift Left 237

LSR • Logical Shift Right 237

MOVE • Move Data 238

MOVEA • Move to Address Register 238

MOVE •	Move to CCR	239
MOVE •	Move from SR	239
MOVE •	Move to SR	240
MOVE •	Move to/from the User Stack Pointer	240
MOVEM •	Move Multiple	241
MOVEP •	Move Peripheral Data	241
MOVEQ •	Move Quick	241
MULS •	Signed Multiply	242
MULU •	Unsigned Multiply	242
NBCD •	Negate Decimal with Extend	243
NEG •	Negate	243
NEGX •	Negate with Extend	244
NOP •	No Operation	244
NOT •	Logical Complement	245
OR •	Inclusive Logical OR	245
ORI •	Inclusive OR Immediate	246
ORI •	Inclusive OR Immediate to CCR	246
ORI •	Inclusive OR Immediate to SR	246
PEA •	Push Effective Address	247
RESET •	Reset External Devices	247
ROL •	Rotate Left	248
ROR •	Rotate Right	248
ROXL •	Rotate Left with Extend	249
ROXR •	Rotate Right with Extend	249
RTE •	Return from Exception	249
RTR •	Return and Retore CCR	250
RTS •	Return from Subroutine	250
SBCD •	Subtract BCD with Extend	250
Scc •	Set According to Condition	251
STOP •	Load Status Register and Stop	252
SUB •	Subtract Binary	252
SUBA •	Subtract Address	253
SUBI •	Subtract Immediate	253
SUBQ •	Subtract Quick	254
SUBX •	Subtract with Extend	254
SWAP •	Swap Register Halves	254
TAS •	Indivisible Test and Set	255
TRAP •	Trap Through the Exception Table	255
TRAPV •	Trap on Overflow	255
TST •	Test an Operand	256
UNLK •	Unlink and Deallocate Stack	256

Index ***257***

INTRODUCTION

Welcome to the *ST Assembly Language Workshop*. Over the course of this volume, and the ones to come, we will teach you everything you need to know to produce your own programs in assembly language on the Atari ST. We at the workshop figure there's no reason it should be only the ST "gurus" who hold the secrets to the wonderful world of 68000 assembly language, because learning assembly language is really not very difficult. As long as you're willing to do a little work and accept some new ways of programming, you too can amaze your friends and other ST users with your assembly language skills.

Who This Book is For

While learning assembly language is not that much more difficult than learning any type of programming language, you probably should already be familiar with computer programming before you tackle this book. We aren't going to take the time here to present an introductory course in computer science. You should understand all the basic programming concepts (i.e., variables, looping, Boolean logic, etc.) and have some programming experience.

Other than some basic programming skills, the only other things you need to bring to the workshop are an assembler, an ST, and a willingness to learn.

Which Assembler?

There are several assemblers available for the ST line of computers, including *DevPac2*, *AssemPro*, *MadMac*, and *AS68*. The programs in this book were designed using the *DevPac2* assembler, published by HiSoft and distributed in the U.S. by Goldleaf Publishing. However, we have also provided full support for Atari's *MadMac* assembler, which comes with the Atari Developer's Kit. The main difference between the two assemblers is the way they handle assembler directives. These differences are mentioned in the body of the text, and are explained fully in Appendix A.

In other words, to follow the programming examples in this book most easily, your best choice for an assembler would be *DevPac2*. But if you have access to Atari's *MadMac*, that assembler will do just fine. Other assemblers can be used, of course, but it will be up to you to correct any incompatibilities that you may run into. Because the actual instructions used in 68000 assembly language are the same from one assembler to the next, code differences will probably arise only with assembler directives and minor points of syntax, all of which can be easily corrected if you're familiar with your assembler or are willing to spend a little time reading your assembler's manual.

A Multi-Volume Work

You've undoubtedly noticed the "Volume 1" tacked onto this book's title. Yes, *The ST Assembly Language Workshop* is a multi-volume work. This volume, part 1, teaches basic assembly language programming, including the most-used 68000 assembly language instructions, many common TOS functions, and basic programming strategies. By the end of this book, you should have a good understanding of 68000 assembly language and be able to write full-length TOS programs on your ST. You'll be able to convert numbers, call OS functions, handle disk files, and send data to a printer. You'll even know how to load and display DEGAS picture files.

Volume 2 of this series will take the basics you've learned here and apply them to GEM programming. You'll learn to handle file selectors, alert boxes, dialog boxes, menus, and windows. Programming a GEM application is a complex task, one that you cannot possibly tackle without a good grounding in the language with which you plan to work, in this case 68000 assembly language. Volume 3 will dig even deeper into the depth of GEM programming, covering advanced topics of interest to programmers who want to get the most from their machines.

How to Use This Book

The ST Assembly Language Workshop is a tutorial. It is not a text book, nor is it a reference book (although the index is complete enough that you should have little difficulty finding information you need). I believe that the best way to learn something is to dig in and get your hands dirty. Therefore, rather than covering topics in an "organized" text book fashion, I cover topics as they arise in the programming experiments. You won't, for example, find explanations of all the addressing modes in one place, as you would in a text book. Instead, each addressing mode is discussed when it appears in a sample program for the first time.

To get the most from each lesson, you should first read the text, getting a general understanding of what the chapter covers. You should then read the chapter again, this time reading more slowly and paying particular attention to the chapter's sample program. (Only two chapters don't include a sample program.) Once you feel comfortable with the material, read over the summary, making sure everything there makes sense to you. Only when you understand everything in the summary should you go to the next chapter.

While we've covered the material presented in this book as completely as possible, you should consider obtaining a couple of reference books on general 68000 assembly language programming and on the ST's operating system. You should have no difficulty finding the former; your local bookstore will likely have many 68000 assembly language reference books. (I use *Programming the 68000* by Steve Williams, published by Sybex.)

Unfortunately, finding reference books for your ST may be difficult (one reason Taylor Ridge Books was formed), since many are now out of print. Your best choice, if you can find it, is Sheldon Leemon's excellent three-volume reference work, which includes the volumes *Atari ST: TOS*, *Atari ST: VDI*, and *Atari ST: AES*. These books were published by Compute books. Atari, can, of course, supply you with complete documentation. To get this documentation, you must purchase the Atari Developer's Kit, which comes not only with complete reference materials, but also the Alcyon C compiler, and the *AS68* and *MadMac* assemblers. Call Atari for more information.

Finally, although not all instructions are covered in the text of this volume, there is a 68000 assembly language quick reference included at the end of this book that includes short descriptions of all 68000 assembly language instructions. Feel free to poke around in there and experiment with what you discover.

The Workshop Begins

So much for the chatter. Get your goggles on, crank up your machine, and let's get grinding. The ST Assembly Language Workshop is now open for business. See you there.

Clayton Walnum
November, 1991

1

A First Look At Assembly Language

So, you want to learn 68000 assembly language, eh? Wonderful! You've probably made one of the wisest choices of your programming career, because learning assembly language will imbue you with the power to make your machine do every trick it's capable of, no holds barred. In fact, assembly language is the only computer language that offers this power.

Why is assembly language so powerful? Because it's the closet thing to machine language, which is the only language your computer actually understands. In a way, assembly language *is* machine language. There is a one-to-one relationship between assembly language instructions and machine language instructions. Assembly language just supplies easy-to-remember names for all the 68000's machine-language instructions, instructions that are really nothing more than numbers.

But before we get too far with this discussion, let's explore some basic questions. Where does assembly language fit into the scheme of things? Is it anything like BASIC or Pascal or C? If you already know a high-level language, will that knowledge help you learn assembly language? Or will you be starting completely from scratch?

So many questions! Let's get right to the answers.

The Answers

Assembly language is similar to high-level languages like BASIC and Pascal in that an assembly language program is a series of instructions that are performed in a specific order. However, assembly language is very unlike other languages when it comes to form and technique. Learning assembly language requires thinking about programming in a whole different way.

Knowing other computer languages will certainly help you with assembly language, because many concepts associated with programming--such as looping constructs and variables--are shared by all computer languages. In fact, as you learn assembly language, you'll also learn a lot more about high-level languages, because you'll begin to understand how those languages work. Mark my words, one day you'll be writing an assembly language program and a giant light will go on inside your head, and you'll say something like "So *that's* how a FOR...NEXT loop works!" or "No wonder I ran out of memory when I called that subroutine inside an infinite loop!"

Language Types

There are three kinds of computer languages: interpreted languages, like BASIC; compiled languages like Pascal or C; and assembly languages. Because your computer can understand only machine language, all three types of languages must be converted into machine language before the computer can run them. One big difference between these three kinds of languages is the way they are converted into machine language.

BASIC, for example, is converted one line at a time. Let's say you've written the following BASIC program:

```
X=X+1
Y=X
```

When you type "run," the BASIC interpreter takes the first line of code, X=X+1, and converts it into machine language. The computer then performs that machine language version of the instruction, after which the interpreter grabs the next line of code and converts it. This constant conversion of code at run time is one reason BASIC is slow. (Most modern versions of BASIC now use compilers, as well as interpreters. A compiled BASIC program runs much faster than an interpreted one, for reasons we shall soon see.)

When you write a program in a compiled language like Pascal, the source code must be completely converted into machine language before you can run the program. This process is called compiling. Although you still have to wait for the computer to do the conversion, once you've done it, you never have to do it again (unless, of course, you make changes to the program). Your compiler stores the converted program onto your disk, from which it can be run any time you please. However, although compiled programs run much faster than interpreted ones, they are usually slower than assembly language programs. This is due to unavoidable inefficiencies in the way the source code is converted into machine language.

Assembly language programs can be converted (assembled) into machine language very quickly and also run very quickly, because assembly language really is just another form of machine language. The conversion process is perfect, involving none of the inefficiencies inherent in the compilation process. Each assembly language instruction has an exact machine language counterpart.

Our Own Machine Language

Confused? Let's say we're electronics engineers, and we're going to create a brand new computer. Because this is our machine, we can design its instruction set any way we like. We do, however, have to remember that computers understand only numbers, so each instruction code we devise for our computer must be a number. To keep things simple, let's say that our computer has only three memory locations named A, B, and C. Now, let's make up the machine language code we need to add two numbers.

First, we must retrieve the numbers we want to add. Let's say the code 12 means "load a number into A." For our new machine language instruction to make sense, we need to know the number to load. So, now let's make up a rule that says every time we use the instruction 12, it must be followed by the number to load. In this example we'll be adding the numbers 2 and 3, so our first machine language instruction for our wonderful new computer is 12 2, which means "load the number 2 into memory location A."

Now, we want to load the second number into ... oh ... how about memory location B? We pick a code to be our machine language instruction for loading a number into B. How about 35? Just as with our instruction 12, we need to follow the instruction 35 with the number to load. We now have our second machine language instruction, 35 3, which means "load the number 3 into memory location B."

Next, we need to make up an instruction that adds the numbers in A and B and stores the answer in our final memory location, C. How about 47 this time? Our complete machine language program, then, is:

```
12 2
35 3
47
```

Machine Language to Assembly Language

If you were to stop reading here and come back to this lesson tomorrow, when you looked at the list of numbers making up our machine language program, it probably wouldn't mean anything to you. Similarly, if you were to cover your eyes right now and then try to reproduce the program, you'd probably have a hard time remembering it.

Assembly language to the rescue! We can make our machine language instructions easier to read and easier to remember by giving them more readable names. Let's assign the name LDA (LoaD A) to 12, the name LDB (LoaD B) to 35, and ADD to 47. Each name we chose is a mnemonic (look that word up in the dictionary if you don't know what it means), because it helps us remember what the instructions are supposed to do. Our program now looks like this:

```
LDA 2
LDB 3
ADD
```

Wow! That's a heck of a lot easier to remember than all those numbers, isn't it? And guess what? We just created our own little assembly language. We could go on, making up more and more instructions, giving us the power to do all sorts of awesome things with our newly designed computer. Of course, with a real computer, the instructions are not just pulled out of the air like ours were; there are lots of complications that only a physics whiz can understand. But we don't care about that. As far as we're concerned, real machine language instructions are just a bunch of numbers that are used much like the machine language instructions we made up.

You should now understand what I meant when I said assembly language and machine language share a one-to-one relationship: LDA=12, LDB=35, and ADD=45. Each instruction in our machine language has its own assembly language mnemonic. An assembler changes the mnemonics back into numbers for the machine, because our computer understands 12, not LDA; 45, not ADD.

Assemblers and Housekeeping

Besides converting mnemonics back into numbers, assemblers also do a lot of dirty work for us. For example, let's say we want to add those two numbers in our machine repeatedly. We could write a long line of instructions that goes on forever, but it'd be much easier to jump somehow backward to the beginning of our three instructions every time we finished adding the numbers. That means we need another instruction. Let's use the number 39 and assign it the name JMP (JuMP). Here's our code now:

```
LDA  2
LDB  3
ADD
JMP
```

We still have a problem. When we get to the JMP instruction, how do we know where to jump? We need to tell the computer how many instructions to skip back in order to get to the beginning of the code. We need to jump backward through ADD, 3, LDB, 2, then stop on LDA, right? Counting each number (remember: our mnemonics represent numbers), we now know we must jump back five places. In our JMP instruction, we'll use negative numbers when we want to jump backward and positive numbers when we want to jump forward. Here's our code now:

```
LDA  2
LDB  3
ADD
JMP  -5
```

Many years ago programmers used to program the way we just did, counting each byte of memory in order to calculate where jump and branch instructions should go. That task was easy for us, because we have only a few instructions in our program. Can you imagine having to count bytes for programs made up of thousands of instructions? I'd rather dig holes in sidewalks with a spoon.

Again assembly language comes to our rescue, by allowing us to name places in memory and tell the assembler the name when we want to jump to that place. The assembler does all the counting for us. To see how this works, let's add a name to our example program:

```
LOOP  LDA 2
      LDB 3
      ADD
      JMP LOOP
```

As you can see, we've now replaced the -5 with the name, or "label" as it's called in assembly language, of the instruction to which we want our program to jump. We don't have to do any calculations at all. When we assemble the program into machine language, the assembler will do the counting for us. And you thought assembly language was difficult!

Using labels we can name any location in a computer's memory. It doesn't have to be the location of an instruction. It could be the location of text we want to print to the screen or the location of a variable. Labels are a powerful tool, as we will see as we learn more about assembly language.

Registers and Memory

Remember when I said our computer would have only three memory locations labeled A, B, and C? I lied. Our computer must have many more memory locations than that. Do you know why? We have to store our program someplace, right? To run the assembly language program we just wrote, we need at least seven additional memory locations, enough space in which to load our program.

What I should have said before was that our new computer would have only three "registers." Registers are special memory locations used to store numbers and perform calculations. Your ST (and any other computer using a 68000 microcomputer chip) has 16 regular registers, plus a few special-purpose registers. We'll talk about them in detail in a later lesson.

So, now we must add some extra memory to our computer. Let's give it 10 bytes, and then load our program into the new memory. If we could look inside the computer's memory, we'd now see something like this:

```
12 2 35 3 47 39 -5 0 0 0
```

The first seven locations are our program. (Recognize the numbers?) The last three locations are zeros because those locations are empty. (In a real programming situation, you can never assume that an unused memory location contains a zero; it could contain any number.)

In actuality, a computer already has names for every location in its memory. Those names are called addresses. Because a computer can deal only with numbers, it should be obvious that addresses are also numbers. To make things easy for us humans, computers name their memory locations starting with zero, incrementing each "address" for each successive location. If we wanted to draw a picture of the memory in our new computer, it might look something like figure 1.1.

The Program Counter

Okay, so now we've got some memory in our computer to go along with our three registers. Now, let's tell the computer to run our program. Go ahead: tell it. What? Nothing happened? Hmmm. I guess we need to do a little more work.

First, we need to add a marker into memory that'll tell our computer where to start processing our program. We'll

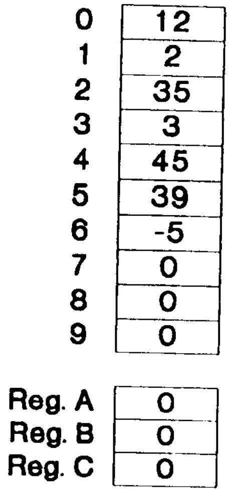


Figure 1.1

also use this marker to help the computer keep track of where in our program we are.

How about adding another register? We'll use this new register only for pointing to the next instruction we want to execute. We'll keep the marker updated so that it always points to the next instruction. Figure 1.2a shows our computer with its new register (labeled "PC"), and the program ready to run.

Now, when we tell the computer to run our program, it "fetches" the instruction pointed to by the new register, updates our marker, then executes the instruction it just fetched. When it's finished with the first instruction, the computer looks where the marker is now pointing and loads that instruction. And so it goes until our program finishes.

In the high-tech world of computers, our new marker register has a name. It's called the program counter (PC). Every computer, including your ST, has one of these handy memory bookmarks. Let's see exactly what happens when we run our program. Figures 1.2a through 1.2e show how our computer's memory is affected by the program.

In Figure 1.2a, nothing has happened yet. When the computer runs the program, it first fetches the instruction 12 (LDA) and the instruction's argument, 2. The PC moves to address 2, which holds the next instruction, after which the first instruction is performed, loading register A with the number 2. Figure 1.2b shows the computer's memory after this instruction is executed.

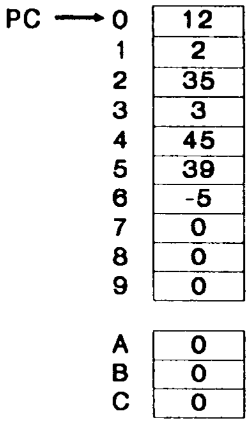


Figure 1.2a

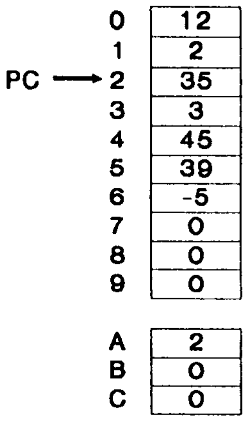


Figure 1.2b

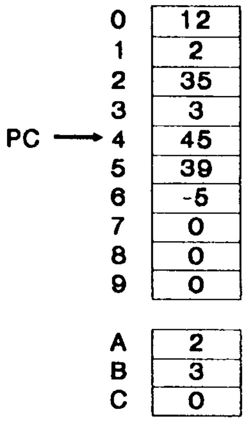


Figure 1.2c

Next, the computer fetches the instruction 35 (LDB) and its data, 3. The PC is updated to address 4. The instruction is executed, loading register B with 3. Figure 1.2c shows our memory's current state.

Having completed the LDB instruction, the computer fetches the instruction 45 (ADD), moves the PC to address 5, and then performs the instruction, adding register A and register B and placing the result in register C, as shown in figure 1.2d.

Finally, the instruction 39 (JMP) and its argument (-5) are fetched. Because of the negative jump, the PC is moved back five memory locations, as shown in figure 1.2e. This brings us almost back to the state shown in figure 1.2a; the difference is that now our registers contain values other than zero.

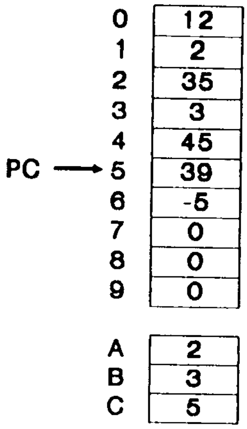


Figure 1.2d

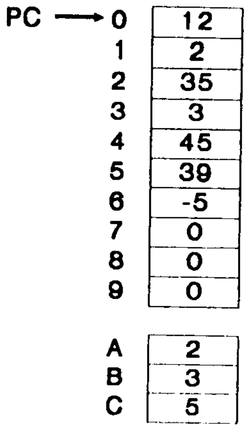


Figure 1.2e

One important thing to note about the PC is that it always points to an instruction, never to a piece of data. If the PC in your ST should ever get out of whack and point to something other than an instruction, you'll almost certainly find a string of bombs on your computer's screen.

The Stack

Suppose we now wanted to use our three registers for something else, but we didn't want to lose the values they already held? What we need in our computer is a kind of scratch pad, a place where we can temporarily store values, without worrying about trodding all over important data. What we need is a section of memory called a "stack."

This strange area of memory is called a stack because of the way it works. Imagine a stack of plates. You can add new plates onto the top of the stack. You can also take plates off the top of the stack. But you can't get at the plates on the bottom until you've unloaded all the plates off the top. A computer's stack works the same way. Values are accessed in a last-on-first-off manner.

Figure 1.3a shows the stack for our new computer. One thing you should notice right off is that our stack has a "bookmark" just like our computer's main memory does. This bookmark is called the "stack pointer (SP), and it always points to the top of the stack. In assembly language, we can put (push) values onto the stack for safe keeping and then take (pop) them off again when we need them. The stack pointer in Figure 1.3a is not pointing to anything yet, because we have nothing on our stack.

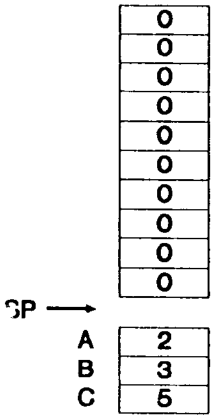


Figure 1.3a

Let's try using the stack and see what happens. Since we want to reuse our three registers without losing their values, let's save the values on the stack. First, we push register A onto the stack, as shown in figure 1.3b. To do this we first move SP forward, then copy the value in register A into the stack location pointed to by SP. Now, we do the same thing with registers B and C, as shown in figures 1.3c and 1.3d. Notice that, so far, we've done nothing to change the values of our registers; we've only copied their values onto the stack.

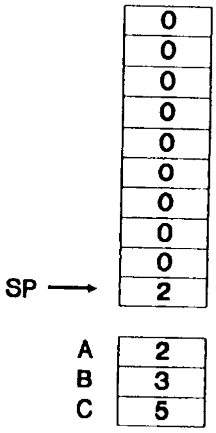


Figure 1.3b

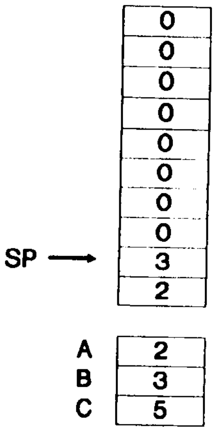


Figure 1.3c

Now the contents of our three registers are safely tucked away, waiting until we need them again. In figure 1.3e, we've gone ahead and used the registers to add two more numbers. The registers' previous values are now destroyed. No problem! We saved their values on the stack.

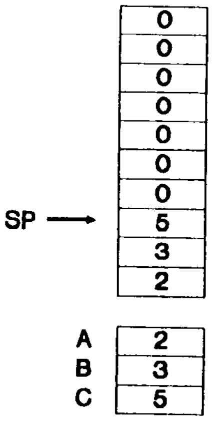


Figure 1.3d

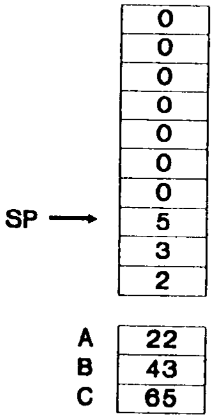


Figure 1.3e

When we want registers A, B, and C restored to their original values, all we must do is pop their values off the stack. Because we can access the stack from only one end, we must pop the register's values off in reverse order; that is, opposite of the way we pushed them on. (That's what we mean by "last-on-first-off.") Figures 1.3f, 1.3g, and 1.3h show the process of popping the three values from the stack.

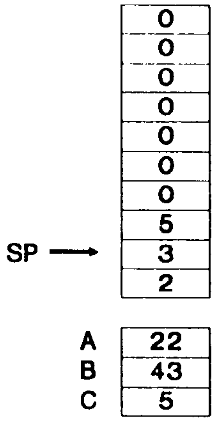


Figure 1.3f

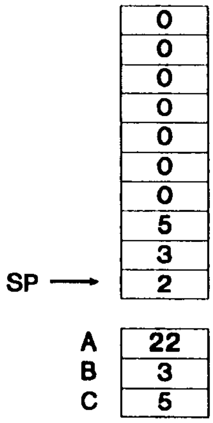


Figure 1.3g

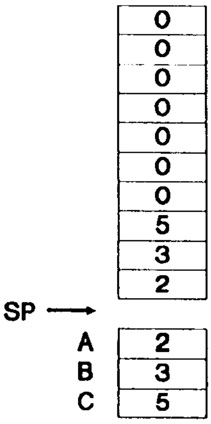


Figure 1.3h

Our registers have now been restored to their original condition. Notice, however, that the numbers we placed on the stack are still there, but because the stack pointer always points to the top of the stack, those values are as good as gone as far as we're concerned. The next time we use the stack, we'll write new values over the old ones.

Let's Get Busy

Okay, okay. I know you're impatient to finish all this theoretical stuff and get to the real programming. You'll be delighted to know that I'm not going to do what many assembly language book authors do: force you to read 10 chapters of theory and background before you actually get to do some programming. I think you learn best by doing, so let's dive right in with our first 68000 assembly language program.

On your disk, you'll find two files in the CHAP1 folder: PROG1.S and PROG1.TOS. The former is the source code for our first program and the latter is the assembled, runnable program. To assemble the source code, follow the instructions that came with your assembler or check this book's appendix A. Remember that the assembly language code in this book is written for DevPacST or Atari's MadMac assembler. If you're using a different assembler, you may have to make modifications to the source code to get it to assemble correctly.

Our first program does nothing more than print a line of text to the screen, wait for a keypress, and then exit back to the desktop. Although the program is short, it contains many of the elements of a full assembly language program, including comments, instructions, assembler directives, and storage areas. Let's start at the top of the program and look at each line, to see what it does.

The first six lines are comments. Whenever DevPacST or MadMac sees a semicolon in the first position of a line, it knows that what follows is text we've added for our own use, usually to document or identify a program. You can write whatever you want after the semicolon. The assembler will skip over it all, going right to the next line of code.

In line 7 you see the word "text." This is an "assembler directive" that tells the assembler where the program begins. An assembler directive gives instructions to the assembler, but doesn't generate machine language code. Following the word "text," you see ten lines (counting blank lines) of assembly language code. Each line of assembly language source code has four parts, or "fields," although not all four fields are necessarily used. The four fields are:

```
LABEL:   OPCODE   OPERANDS   COMMENT
```

Whereas every line of assembly language code must have the opcode (operation code) and operand fields, they do not have to have the label and comment fields.

Look at the second line after the word "text." The word *move.l* is the opcode that tells your ST to copy a piece of data from one location to another. The *#string*, the first operand, is the data to copy, and the *-(sp)*, the second operand, tells the ST where to copy it. (The minus sign and the parentheses tell the ST to do something else as well; we'll talk about that later.) See the "sp"? Does that mean something to you? If you said that "sp" stands for "stack pointer," you get a gold star.

Following the opcode and its arguments, you'll see a semicolon. This is the beginning of another comment. Again, everything following the semicolon is ignored by the assembler. The comment is only for us humans. In assembly language, more than in any other language, comments are important. Assembly language can be confusing if you forget exactly what you're doing, and the instructions themselves are not always easy to interpret. Of course, you won't usually have to comment every line the way we have here. But you should use enough comments to make sure that, when you look at your program later down the line, you can see what you were doing.

The only part of the instruction line we haven't seen here is the label part. We discussed labels a little earlier, so you should already have a feel for what they do. We'll see lots of labels in future programs.

See the word "data"? This is another assembler directive. This time we're telling the assembler that the lines following constitute the area in which we're storing our data. The word "string" is a label that names the starting location of the string we want to print to the screen.

The *dc.b* stands for "define constant.byte." It is also an assembler directive. It tells the assembler how to store our string in memory, in this case as data that will be interpreted as bytes. We'll take a closer look at these directives when we learn more about the format in which data is stored in your ST's memory.

Conclusion

If you didn't understand some of our first 68000 assembly language program, don't worry about it. We'll be covering the details in greater depth later. You should, however, have a good feel for what assembly language is, how it is different from other computer languages, and how it is similar to machine language, the only language a computer understands. In addition, you should know how a computer stores and runs a program. Finally, you should know what program counters and stack pointers are and how they help the computer keep track of what it's doing.

To make sure you understand the concepts presented in this chapter, review the following summary carefully.

Summary

- ✓ Assembly language is the closest thing to machine language, which is the only language your computer understands.
- ✓ Machine language is made up of only numbers.
- ✓ Assembly language uses easy-to-remember mnemonics to represent machine language instructions.
- ✓ One way assemblers make programming easier is by doing much of the work for us, such as counting the length of jumps and branches.

✓ Registers are special places in memory that your computer uses to store numbers and perform calculations.

✓ The program counter is a register that acts as a program bookmark, always pointing to the next executable instruction.

✓ The stack is a section of memory that works like a scratchpad, where we can store values temporarily.

✓ The stack pointer always points to the top of the stack--that is, the last value placed on the stack.

✓ 68000 assembly language instructions are made up of four parts: the label, opcode, operand, and comment fields.

✓ With most assemblers, a semicolon marks the beginning of a comment.

```
;---------------------------------------------------------
; THE ST ASSEMBLY LANGUAGE WORKSHOP
; PROGRAM 1
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;---------------------------------------------------------
        text

        move.l  #string,-(sp) ; put string address on stack.
        move.w  #9,-(sp)      ; function # to display string.
        trap    #1            ; call the O.S.
        addq.l  #6,sp         ; clean up the stack.

        move.w  #1,-(sp)      ; function # to wait for keypress.
        trap    #1            ; call the O.S.

        move.w  #0,-(sp)      ; function # to exit program.
        trap    #1            ; call the O.S.

        data

string:         dc.b    "Our first program!",13,10,0
```

2 Data Sizes and Number Systems

In the previous chapter, we took a quick look at what assembly language is and how it works. Now, we're going to get down to business and learn some important concepts that apply specifically to 68000 assembly language, as well as tackle a little computer math. No grumbling allowed! Assembly language is very different from other programming languages you may have learned. Before you can program in assembly language, you must understand some important background material.

Data Sizes

If you've had any programming experience, you're familiar with the terms bits, bytes, words, and long words. These terms describe the basic elements that make up any data. But until now your programming experience has probably been in high-level languages. High-level languages and assembly language handle data sizes differently. For example, in most high-level languages, when you declare a variable, you implicitly declare it's size. When you declare the variable *Number* as a variable of type integer, the language automatically assigns two bytes (on a 68000-based machine) to that variable. From then on, you don't have to worry about it, except to be sure that you don't end up with type mismatches in your instructions (trying to add a floating point number to an integer, for example).

With assembly language, however, as far as the computer is concerned, variables don't have types and sizes. A variable name is nothing more than an address. Every instruction in your source code must tell the assembler what size of data that instruction is to work on. In other words, assembly language is really concerned only with addresses, not data sizes. In fact, to a computer, all memory is simply a long string of 0's and 1's. Bytes, words, and long words are human concepts that we use to help us organize all those 0's and 1's in our own minds. They really don't exist at all.

Bits, Bytes, Words, and Long Words

A bit, which stands for binary digit, is the smallest piece of data a computer can handle. A bit can be "on" (1), or it can be "off" (0). It has only these two values. Amazingly enough, those 0's and 1's are all your computer needs to perform its amazing stunts.

Other types of data are constructed by putting together a series of bits. A byte, for example, is eight bits. Because we now have eight digits, a byte can represent 256 different values, instead of only two. Bytes are usually interpreted as a value from 0 to 255. They can also represent the values -128 to [illegible], when dealing with signed numbers. (We'll discuss signed numbers later.)

1	0	0	1	1	0	1	0

Figure 2.1 • A Byte

In 68000 assembly language, a word comprises two bytes, or 16 bits. (When talking about words, it's important to specify the hardware. A word on a 6502 machine is only one byte.) Because we now have many more bits to work with, a word can represent values from 0 to 65,535 (or, with signed numbers, -32,768 to +32,767).

Because the order in which we interpret the two bytes is extremely important, we must decide on a "most significant byte" (the byte that has the greatest impact on the word's value) and a "least significant byte" (the byte that has the least impact on the word's value). So, in 68000 assembly language, a word always starts at an even address and is stored in high byte/low byte fashion. That is, the most significant byte (high byte) is in the even-numbered, lower address, and the least significant byte (low byte) is in the odd-numbered, higher address, as shown in figure 2.2.

If this ordering seems strange, think about how we interpret numbers in the real world. Take a number like 16,532, for example. Aren't the least-significant digits found on the right and the most-significant (those with the highest value) on the left?

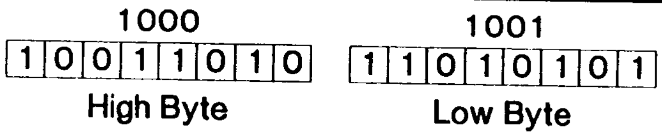


Figure 2.2 • A Word

A long word in a 68000-based machine comprises four bytes, or 32 bits, and can store numbers into the billions. Because of their size, long words are mostly used to store addresses. Just as with word-length data, long words start on an even address, but a long word is divided into a high word and a low word, each of which is in turn divided into a high byte and a low byte.

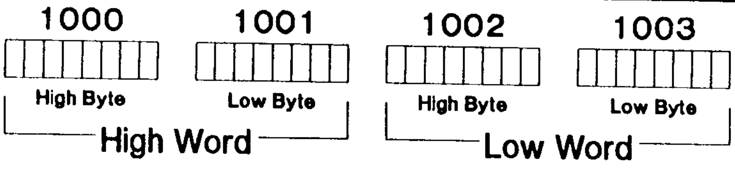


Figure 2.3 • A Long Word

Binary, Hexadecimal, and Decimal Numbers

Remember that, to your computer, all numbers are nothing more than a string of binary digits. This may be great for your machine, but it's not good enough for us humans. As assembly language programmers, we need to deal with numbers in different ways. Sometimes the binary representation will be best for what we want to do. But often, we may want to think of numbers in decimal form, or in an abbreviated "binary" form called hexadecimal.

Binary, decimal, and hexadecimal numbers are simply numbers in different base systems. Binary numbers are base 2, decimal numbers are base 10, and hexadecimal numbers are base 16. Because 16 is an even multiple of two, binary and hexadecimal numbers are closely related. You can think of hexadecimal numbers as a short form of binary. Confused?

Number Systems

Okay, let's see how number systems work, starting with our own base 10 system. Consider the number 5,302. As you know from grade school arithmetic, the location of each digit in 5,302 determines that digit's value. The "2" is in the one's place, the "0" is in the ten's place, the "3" is in the hundred's place, and the "5" is in the thousand's place. To calculate the value of the entire number, we multiply each digit times its place value, and then add the results together, like this:

```
2 x 1    = 2
0 x 10   = 0
3 x 100  = 300
5 x 1000 = 5,000

2+0+300+5,000=5,302
```

Notice that the place values--1, 10, 100, and 1,000--are all powers of 10, as shown below.

```
   1 = 10^0
  10 = 10^1
 100 = 10^2
1000 = 10^3
```

This is why we say our decimal numbering system is base 10, because the place values of each digit are powers of 10. (An interesting side note: We use a base 10 number system because we have 10 fingers. That's where the term "digits" comes from; early man used his fingers to count. If humans had been created with only eight fingers, we'd now be counting in a base eight numbering system--octal rather than decimal!)

Now let's take the same number in binary: 1010010110110. You may not believe it, but this is still 5,302. It's now in base 2 or binary format. In base 2, each digit's place is calculated as a power of 2 rather than a power of 10, so we interpret our binary number like this:

```
0 x  2^0 =     0
1 x  2^1 =     2
1 x  2^2 =     4
0 x  2^3 =     0
1 x  2^4 =    16
1 x  2^5 =    32
0 x  2^6 =     0
1 x  2^7 =   128
0 x  2^8 =     0
0 x  2^9 =     0
1 x 2^10 = 1,024
0 x 2^11 =     0
1 x 2^12 = 4,096
           -----
           5,302
```

As you can see, though binary numbers are long and look strange, they aren't much different from the decimal numbers we're used to using. The only difference is that each digit's place value is interpreted as a power of 2 rather than a power of 10. Of course, when we look at 5,302, we immediately understand the number, because we've had so much experience with base 10 numbers. When we look at 1010010110110, there's no way we can interpret it without first translating it into our own number system.

Hexadecimal Digits

There is, however, a way we can shorten binary numbers so they aren't as confusing. We do this by converting them to hexadecimal numbers, which are base 16. But using a base 16 system creates a problem for us decimal users. Notice that in our base 2, binary system the only digits we have are 0 and 1. In our base 10 decimal system, we have the digits 0 through 9. In other words, the number of digits we need to represent a number in a given base system is 0 to base-1. This means that to represent numbers in a base 16 number system, we need 16 unique digits, to represent the values 0 through 15. Uh-oh. Because we've always used a base 10 system, we have only 10 digits with which to work. Looks like we're gong to have to invent some new ones. How about if we use letters of the alphabet to represent digits with values of 10 through 15? Using this rule, our hexadecimal digits look like this:

```
Ø 1 2 3 4 5 6 7 8 9 A B C D E F
```

Converting Binary to Hexadecimal

Remember when I said base 16 numbers were a handy way to abbreviate binary numbers, because 16 is an even multiple of 2? As an example, let's take 5,302 again, which is 1010010110110 in binary. By converting this number to hexadecimal, we can see what 5,302 looks like in base 16.

First, starting from the right, divide the binary number into groups of four digits. Our binary version of 5,302 then looks like this:

```
1 0100 1011 0110
```

If you like things neat and tidy, you can add some leading zeros to keep everything in groups of four. This isn't necessary, however. If you choose to do this, your binary number looks like this:

```
0001 0100 1011 0110
```

Now, we must convert each group of four digits into a single hex digit. When converting a binary number, all we need to know is the place value of each digit. (Well, we need to know how to add too.) Starting from the right, the place values of each digit are 1, 2, 4, and 8. If a digit is 0, then the value for that place is zero. If the digit is a one, we add the place value of that digit to our number. So our binary number above converts like this:

```
0+0+0+1  =   1
0+4+0+0  =   4
8+0+2+1  =  11
0+4+2+0  =   6
```

Substituting hex digits for the values above, we get the hex number 14B6, which is the hex equivalent of 5,302.

Converting from Hexadecimal to Binary

To convert from a hexadecimal number to a binary number isn't as easy. We must rewrite each hex digit as a four-digit binary number so the sum of the place values of each binary digit is equal to the hex digit. Get that? Using 14B6 as an example, we first change each hex digit into its decimal equivalent:

```
1 4 11 6
```

We then change each decimal equivalent into binary form, remembering that bit 3 equals 8, bit 2 equals 4, bit 1 equals 2, and bit 0 equals 1. This conversion gives us the binary number 0001 0100 1011 0110, which should look familiar. In order to help you with this type of conversion, table 2.1 lists the binary equivalents of all the hex digits.

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Table 2.1 • Hex to Binary Conversion

Converting Decimal to Binary

Because of their close relationship, it's easy to convert between binary numbers and hexadecimal numbers. However, converting decimal numbers to binary or hexadecimal is a little tricky. It's not hard; it's just a meticulous process. Let's again use 5,302 as an example and convert it into binary. The first step is to find the highest binary place value that is smaller than 5,302. For example, in table 2.2, which lists the place values for a binary word, we see that the highest value that'll go into 5,302 is 4,096, which is in bit

12's place. So, we construct a binary number with 13 digits (numbered 0 through 12), placing a 1 in bit 12's place, giving us 1000000000000. We then subtract 4,096 from 5,302, giving us 1,206.

Now, we move down to the next bit's place, which is bit 11 with a value of 2,048. This number is too big to go into 1,206, so we place a 0 in bit 11's place. Moving back another bit, we see that bit 10 has a value of 1,024, which will fit into 1,206. We place a 1 in bit 10's place and subtract 1,024 from 1,206, giving us 1010000000000 and 182, respectively. We continue the above process until we have a complete binary number. Table 2.3 shows a summary of the entire conversion.

Bit #	Value	Bit #	Value	Bit #	Value	Bit #	Value
0	1	4	16	8	256	12	4,096
1	2	5	32	9	512	13	8,192
2	4	6	64	10	1,024	14	16,384
3	8	7	128	11	2,048	15	32,768

Table 2.2 • Binary Place Values

Binary Number	Bit#	Value	Operation	Remainder
				5,302
1	12	4,096	5,302 - 4,096	1,206
10	11	2,048	None	1,206
101	10	1,024	1,206 - 1,024	182
1010	9	512	None	182
10100	8	256	None	182
101001	7	128	182 - 128	54
1010010	6	64	None	54
10100101	5	32	54 - 32	22
101001011	4	16	22 - 16	6
1010010110	3	8	None	6
10100101101	2	4	6 - 4	2
101001011011	1	2	2 - 2	0
1010010110110	0	1	None	0

Table 2.3 • Decimal to Binary Conversion

Converting Decimal to Hexadecimal

Converting a decimal number to hexadecimal works almost the same way as converting into binary, except we use the place values of hexadecimal digits, and we use division, which will usually yield results larger than just 0 or 1. Taking the number 5,302 again, we start by finding the largest hex place value that'll divide into it. Looking at table 2.4, we that this value is 4,096. We divide 5,302 by 4,096, getting a result of 1 and a remainder of 1,206. The value 4,096 is in digit 3's place, so we start with the hex number 1000. Dropping down to the next digit, we divide 1,206 by 256, giving us a result of 4 and a remainder of 182. We then put a 4 in digit 2's place, giving us the hex number 1400. This process continues until we get the complete hex number. The conversion process for changing 5,302 into hex is summarized in table 2.5.

Digit #	Value
0	1
1	16
2	256
3	4,096
4	65,536
5	1,048,576
6	16,777,216
7	268,435,456

Table 2.4 • Hex Digit Place Values

Digit #	Value	Operation	Result	Remainder	Hex #
				5,302	
3	4,096	5,302 / 4,096	1	1,206	1
2	256	1,206 / 256	4	182	14
1	16	182 / 16	11	6	14B
0	1	6 / 1	6	0	14B6

Table 2.5 • Decimal to Hex Conversion

The Easy Way

Although it's important to understand how to convert between decimal, binary, and hexadecimal numbers, when you start programming, you'll waste a lot of time doing these conversions by hand. Pick up a scientific calculator that can perform these conversions for you, and you'll save yourself some aggravation. If you like having an "on-line" calculator, your *Assembly Language Workshop* disk contains a public-domain program called *The Take Note Calculator* that can be installed on your ST as a desk accessory.

Conclusion

Yeah, I know this chapter was a bit technical, but you simply cannot understand assembly language well until you understand how computers and programmers deal with numbers. You'll be doing yourself an immense favor if you take the time to learn well the concepts presented here. If you don't, mark my words, you'll be back here before you know it.

Summary

- ✓ Computers understand only binary numbers, which are comprised of bits.
- ✓ To make computing easier for us humans, we created several data types that let us combine bits into bytes, words, and long words.
- ✓ In 68000 assembly language, bytes allow values from 0 to 255, words (two bytes) allow values from 0 to 65,535, and long words (four bytes) allow values into the billions.

✓ Word and long word values must start on an even address and are stored in high byte/low byte fashion.

✓ Although a computer stores all its data as 0's and 1's, we can interpret numbers using several number systems. When programming in assembly language, we frequently need to convert between decimal, binary, and hexadecimal values.

✓ Because we need 16 digits to represent hexadecimal values, we use the letters A through F to represent digit values of 10 through 15.

✓ Binary and hexadecimal numbers are closely related, so it's fairly easy to convert between them. However, converting a decimal number into a binary or hexadecimal number requires a series of subtractions or divisions.

3

Registers, The Stack, And Addressing Modes

In this chapter, we'll look at some technical details we need to know in order to study 68000 assembly language programs. Although we touched on registers and stacks in chapter 1, we discussed them only generally. Now, we'll talk about registers and the stack as they are implemented on your ST. We'll also discuss addressing modes, which tell your ST how to find data, and instruction suffixes, which tell your assembler how to handle data. In the next chapter, we'll get to work on some real programming.

Registers

Every computer that uses the 68000 processor, including your ST, has 16 main registers. Registers are simply special memory locations that are built into the processor, where they can be accessed quickly. The 68000's registers are divided into two types: data registers and address registers. Both types of registers can hold a long word (four bytes), but address registers are used for holding addresses, whereas data registers can hold any type of data we want. Note that only the lower three bytes of an address register are used to specify an address, which means that the highest address you can access is 16,772,216. You may have heard that a 68000-based machine can never address more than 16 megabytes. Now you know why.

There are eight data registers, named D0, D1, D2, D3, D4, D5, D6, and D7, and eight address registers, named A0, A1, A2, A3, A4, A5, A6, and A7. Register A7 is the system stack pointer, which can be written as either A7 or SP in your programs. In addition, there are two special registers: the program counter and the status register.

The Program Counter

As we learned in chapter 1, a program counter always points to (contains the address of) the next executable instruction in our program. The ST's program counter (PC) is no different. When we load a program, the PC points to the first instruction. When an instruction is executed, the PC moves to the next instruction. We can manipulate the PC in several ways. Whenever we perform any kind of jump or branch instruction, we are changing the contents of the PC. Just as with any address pointer, only the three lowest bytes of the PC are used to store an address, so its maximum value is 16,772,216.

The Status Register

Most computers have a status register, which is used to store information about the machine, most importantly, the results of the last executed instruction. Our programs will use the status register to decide when to branch to other parts of the program. The ST's 16-bit status register comprises a "system byte" and a "user byte." The system byte is used mostly by systems programmers, so we won't go into much detail on it. However, you should know that the system byte is bits 8 to 15 of the status register.

The lowest eight bits of the status register (bits 0 to 7) are the user byte, which comprises three unused bits and five "condition codes" that contain information about the outcome of the last executed instruction. These condition codes are the carry bit, the overflow bit, the zero bit, the negative bit, and the extended bit. The carry bit holds the carry-over from an arithmetic instruction; the overflow bit is set when an instruction results in a value too large to be stored; the zero bit is set when an instruction results in a zero value; the negative bit is set when an operation results in a negative value (when the most significant bit of a result is set to 1, the result is considered to be negative); and the extended bit is an extension of the carry bit and is used in multiple-precision operations. These condition bits are arranged in the status register as shown in figure 3.1.

If you've no experience with assembly language, the last couple of paragraphs have probably got your head spinning. Don't worry if you don't understand all this stuff. Absorb what you can, and then move on. We'll talk about the condition codes in greater detail later on.

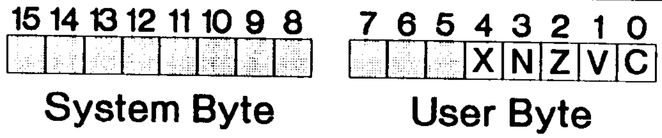


Figure 3.1 • The Status Register

The Stack and ST Function Calls

The ST's operating system provides programmers with many functions, such as reading characters from the keyboard, printing characters to the screen, performing disk functions, and myriad others. Most of these functions are called with a "trap" statement. A trap statement signals the operating system that we want it to do something for us. Before we can perform a *trap*, however, we must place the function's arguments on the stack. Then, when we call the function by executing the trap statement, the computer uses the information on the stack to perform the function.

For example, in chapter 1's program, we used the *Cconin* function to grab a character from the keyboard. Before the operating system could do that, though, we needed to tell it what function we wanted it to perform. We did that by placing the function's number onto the stack.

Then when we executed the trap instruction (*trap #1*), the computer looked on the stack, found the function number, and knew what we wanted it to do. To put it simply, we used the stack to communicate between our program and the operating system.

The *Cconin* call is a simple example. Many functions we'll use in our assembly language programs require more than one value to be placed on the stack. Some function calls, for instance, also need an address where the operating system can find additional data.

As you saw from our stack discussion in chapter 1, whatever we put on the stack stays there. It's up to us as assembly language programmers to keep the stack cleaned up. We do this by resetting the stack pointer to its original position. Whenever we use the stack to perform function calls, we must be sure to take care of this little bit of housekeeping. In our next program, we'll see how to do this. But before we can study an assembly language program in any depth, we must learn about addressing modes.

Addressing Modes

Every instruction in a computer program needs data upon which to work. Obviously, each instruction needs to know where its data is located. There are many ways that your assembler can calculate a datum's address. Each of these methods uses one of 13 addressing modes: immediate, absolute short, absolute long, data register direct, address register direct, address register indirect, address register indirect with post-increment, address register indirect with pre-decrement, address register indirect with displacement, address register indirect with index, program counter with displacement, program counter with index, and status register.

Don't let these fancy terms scare you. We'll talk about each of the addressing modes as they come up. You'll be happy to know, however, that you don't need to memorize all those tongue-twisting address-mode names. The addressing modes are implicit in the language's syntax. In other words, as you learn to program in 68000 assembly language, you can't help but learn the addressing modes, too.

Instruction Suffixes

Let's say we're writing an assembly language program, and we want to use a variable named *Number*. Because *Number* is an integer, we assign it two bytes. We'll see later exactly how to do this; the key point is that *we* assign it two bytes. The data size isn't handled automatically as it is in high-level languages. We could just as easily assign *Number* three bytes. Our assembler doesn't care. Remember: The terms "integer" and "byte" are abstractions we use to help us organize the computer's memory. They mean nothing to the computer.

We can think of *Number* as a variable name, but it's really a label for an address. If we were to print the value of *Number* to the screen, we'd see the starting address of the two bytes we just reserved for our variable. When we start working with *Number*, we can tell the computer to handle that address any way we want. We can, for example, tell the computer that *Number* is the address of a byte, in which case the computer will happily perform its instruction on the byte at the address *Number*. Likewise, we can tell the computer than *Number* is a word, in which case the computer will perform its instruction on the two bytes starting at *Number*. Although we could run into trouble, we could even tell the assembler that *Number* is the address of a long word, in which case it'd try to handle four bytes. I said "try to" because, since we reserved only two bytes at *Number*, at this point we have no idea what might be in memory beyond those two bytes.

Obviously, we need a way to tell the assembler the size of a piece of data. We do this with instruction suffixes. In 68000 assembly language, most instructions have either an explicit or implicit suffix that tells the assembler what length data that instruction is supposed to work with. For example, let's take the instruction *move.l*. The suffix *.l* following the word *move* tells the assembler that this instruction is to move a long word. Remember, I said that the computer doesn't care about how we declare our data sizes. If we were to use our integer *Number* with the *move.l* instruction, the computer would try to grab four bytes starting at *Number*. It doesn't care that we originally reserved only two bytes at that location. All it knows is that *Number* is a valid address.

So, since we have three lengths of data, our *move* instruction, like many other 68000 assembly language instructions, has three forms: *move.b*, which moves a byte; *move.w*, which moves a word; and *move.l*, which moves a long word. Because words are the standard length of data used in 68000 assembly language, we can also write the *move* command without a suffix. In this case, the assembler assumes that the instruction will be moving word-length data. This is an implicit suffix.

Many other instructions in 68000 assembly language use this same convention. Look at the listing for program 1, found at the end of chapter 1. You can see that most of the instructions have a suffix. The ones that don't have a suffix, don't need them, because they don't manipulate data.

Conclusion

Now that we know something about how our computers are constructed and how assembly language handles data, we're ready to start programming. In the next chapter, we'll tackle our first full-length program, but before we move on, please make sure you understand what we've covered so far, by reviewing the summary on the next page.

Summary

✓ The ST has 16 main registers, including eight address registers (A0 through A7) and eight data registers (D0 through D7).

✓ Register A7 is a special register, called the stack pointer, that is used to manipulate the system stack.

✓ Before we call an operating system function with a *trap* instruction, we must place the function's arguments on the system stack.

✓ The ST also has two special registers: the program counter, which always points to the next executable instruction, and the status register, which is made up of a series of flags we can use to track the status of the machine.

✓ The ST has 13 addressing modes that it uses to calculate the location of data it needs to complete an instruction. Whenever we write an instruction in 68000 assembly language, we use at least one of these addressing modes.

✓ Most 68000 assembly language instructions include a suffix that tells the assembler the size of the data we want to manipulate.

4

Constants, Function Calls, and Branching

Now that we've gotten some technical stuff out of the way, we can start studying 68000 assembly language more seriously. In this chapter, we'll learn about string constants, equates, and several new addressing modes. We'll also learn a couple of new operating system function calls, some new instructions, and how to handle the system stack when calling operating system functions.

The Program

In the CHAP4 folder of your *Assembly Language Workshop* disk, you'll find the files PROG2.S and PROG2.TOS, which are the source code and executable file for program 2. If you'd like to assemble the program yourself, follow the instructions that came with your assembler or check this book's appendix A.

When you run the program, it asks you to press the letter A, B, or C. After pressing a letter, you get a message telling you which key you pressed. For example, if you press B, you see the message "You Pressed B." If you press a key other than those requested, the program repeats its request until you enter a correct key. After printing the message, the program ends, returning control to your ST.

Now, look at the program listing. You can see that it is divided into two main sections, labeled with the words "text" and "data." As we said previously, the text section is where we place the instructions that make up our program, and the data section is where we place our initialized data. In some programs, there also will be a "bss" (block storage segment) section, where we can store uninitialized data or set aside blocks of memory for use as buffers or other types of storage.

The Data Segment

Let's start our examination of the program with the data segment. In the first three lines of the data segment, we define three constants--LET_A, LET_B, and LET_C--with the assembler directive *equ*. This directive does the same job as the *#define* statement in C. When the assembler finds any of these three labels in our source code, it'll replace the label with the number to the right of the *equ*. In this case, these numbers happen to be the ASCII values of the letters A, B, and C.

In the next line, we set up storage for a string. The 13s and the 10s are the ASCII values for carriage returns and line feeds, respectively. (A carriage return/line feed combination moves the screen cursor to the left margin of the next line.) The 0 is a null character to mark the end of the string. When the program is assembled, the label *prompt* will hold the address of the string. The *dc. b* stands for "declare constant.byte." This assembler directive tells the assembler that we are going to store a series of constant values, in byte form, in consecutive addresses starting at *prompt*. Let's say that this address turns out to be 1000. Then memory will look like figure 4.1.

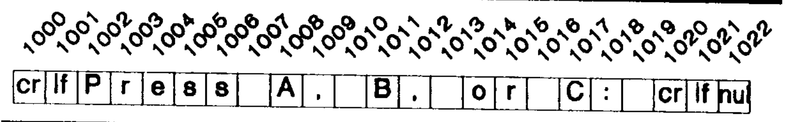


Figure 4.1 • A String in Memory

The next line, where we set up storage for a string labeled *key_msg*, is handled much the same. Notice the syntax of these lines. You can initialize a constant as a string of numbers or as a string of characters. You can even mix both together as we did here. Numbers are separated with commas, while strings must be enclosed in quotes. (The syntax may vary, depending on the assembler you're using.) To give you a clearer example, let's say we want to set up the string constant "ABC," followed by a carriage return and a line feed. Either of the following lines would do the trick:

```
letters:   dc.b   65,66,67,13,10,0
letters:   dc.b   "ABC",13,10,0
```

Here, *letters* is the label that'll hold the address of the string. The assembler directive *dc. b* tells the assembler that we want the values that follow stored as bytes. Why do we need to tell the assembler this? Remember that bytes, words, and long words all have different lengths. Just as with many other instructions, we can add a *. b*, *. w*, or *. l* to the *dc* assembler directive, telling it to store different types of values. To see the difference let's compare the lines below:

```
letters:   dc.b   65,66,67,13,10,0
letters:   dc.w   65,66,67,13,10,0
```

The byte version of this constant declaration will set up memory like this:

```
65 66 67 13 10 0
```

However, the word version of the constant declaration will set up memory like this:

```
0 65 0 66 0 67 0 13 0 10 0 0
```

Do you know why? If you said, "Because a word is two bytes," you're right.

The Text Segment

Now, let's talk about the text segment, where the actual program resides.

In the first line, we mark the beginning of the program with the label *get_letter*. This label will hold the address of the first instruction in our program. In that instruction, *move. l #prompt, -(sp)*, we copy the value stored in *prompt* onto the stack. Obviously, we need to examine this instruction a little more closely.

You already know what the *move. l* instruction does; it copies a long word from one location to another. You also know that the two values following the instruction are its arguments. Normally, these arguments would be the source and destination addresses for the instruction. However, the "#" in front of *prompt* changes it from an address into an "immediate" value. That means that the value stored in *prompt* is to be treated as a number rather than an address.

Confused? Look at the end of the program listing, where you'll see the label *prompt*, which is the address of the string "Press A, B, or C:." Suppose the program is loaded into memory such that our string starts at address 1000. The label *prompt* then contains the value 1000. When we perform the instruction *move. l #prompt -(sp)*, we copy the value 1000 to the stack. However, if we were to perform the same instruction without the pound sign, we would copy the long word starting at address 1000, which includes our line feed, carriage return, and the letters "P" and "r." (A long word is four bytes.)

See the difference? In the first case, we're interpreting *prompt* as the actual data we want to move; in the second case, we're interpreting it as an address that tells us where to get the data to move. When we include the pound sign, we're saying "This is the number we want you to use." When we don't include the pound sign, we're saying, "This is where you'll find the number we want you to use."

Immediate Addressing

By the way, we've just learned our first 68000 assembly language addressing mode. Didn't hurt a bit, did it? When we place the pound sign in front of an argument, we are using "immediate addressing." The data we want is an immediate constant; that is, the data is included in the instruction itself, rather than located in another memory location.

And Back to the Program

So, now we know we're moving a long word and that the long word we're moving is the number stored in the label *prompt* (which happens to be the address of a string constant). But where are we moving it? The *-(sp)* in the instruction tells the assembler where. You know from the previous chapter that *sp* stands for "stack pointer," right? But why is it surrounded with parentheses and preceded with a minus sign? Because we've already discovered a new addressing mode!

Address Register Indirect Addressing with Pre-decrement

Whew! That's a mouthful. As I mentioned before, the stack pointer (SP) is actually register A7, which is an address register. That explains the "address register" part of this addressing mode. The "pre-decrement" part means that the value in the register (in this case, SP) is decremented before the value is used. This is indicated in the instruction by the minus sign in front of the argument. The "indirect" part means that the value stored in SP is not to be used directly, but rather is to be interpreted as an address--here, the destination address of the instruction. Simply, we're using SP to point to the location in which we want the value stored. Guess that's why they call it a pointer, huh?

The Way It Works

Here's what's happening: We've told the computer to move a long word from one location to another. The instruction's first argument tells the computer the location of the long word we want to move. Because we're using immediate addressing with this argument, the instruction uses the number stored in the label *prompt* as the data. Now, the computer needs to know where to put the data. Let's say the stack pointer equals 5000, which means our stack starts at this address. The computer first subtracts four from 5000, giving it 4996. This gives us room for a long word, the four bytes of which will be stored in 4996, 4997, 4998, and 4999. It then takes the number stored in the label *prompt* and copies it into the stack starting at address 4996. If we hadn't put the SP in parentheses--that is, if we hadn't used its value indirectly--the instruction would have stored the value in *prompt* into SP itself, rather than onto the stack.

Wait a minute! We're subtracting from the stack pointer? What kind of craziness is this? I guess I should have mentioned the peculiar way our STs handle the stack. In order to make sure the stack has as much room as it needs, the ST's operating system places it in high memory, and then builds the stack down towards low memory. In other words, the more data you place on the stack, the lower the address in the stack pointer becomes.

Just about every time we place a value on the stack, we'll use the address register indirect with pre-decrement addressing mode. If you look at the next line in the program, you can see that we are placing the number 9 onto the stack. With the 9, we're again using immediate addressing (the pound sign), which tells the computer that 9 is the data we want, not the address at which the data is stored. We are then using address register indirect addressing with predecrement for the second argument, which places the number 9 onto the stack above the value we placed in the first line. This time 2 is subtracted from the stack pointer, because we're moving a word (two bytes) rather than a long word (four bytes). (In this addressing mode, the amount decremented is based upon the size of the data we're referencing in the source argument.) After the second move instruction, we have placed six bytes onto the stack.

When using the stack pointer, we can move only words or long words. This is because the stack pointer must always contain an even address. If you try to fool the computer into moving a byte onto the stack, it'll move a word, anyway. Obviously, you must be aware of the stack's condition at all times. If the stack pointer ever ends up pointing to the wrong address, you could crash your entire system, because your ST also uses the system stack to keep track of other things, like return addresses.

The Function *Cconws*

Why are we putting all this stuff onto the stack? We're getting ready to use a function called *Cconws* to display our prompt "Press A, B, or C:" To use this function, we must put two pieces of data onto the stack: the address of a null-terminated string and *Cconws*'s function number. We've done that with the two lines of code we just discussed. Once the data is placed onto the stack, we must execute a *trap #1* instruction, which tells the computer to check the stack and do whatever it finds there. You can see that we've done this in the third line of code. When the operating system returns from the trap, the low word of register D0 will contain the number of characters actually printed, should we, for some reason, need this information. For now, we'll assume that the entire string was printed. (By the way, we use a *trap #1* to call any GEMDOS function. Your ST's operating system provides three libraries of functions: GEMDOS [*trap #1*], BIOS [*trap #13*], and XBIOS [*trap #14*].)

After the trap instruction, we must clean up the stack, by resetting the stack pointer.

The *addq* instruction

The fourth line of code, ***addq.l #6,sp***, takes care of resetting the stack pointer. Let's say that the stack pointer contained the address 5000 before we started monkeying with it. After placing the two arguments on the stack, the stack pointer contains 4994. To clean up the stack, we don't actually have to remove the values we stored there. We can ignore them. All we must do is set the stack pointer back to where we found it. The ***addq.l*** instruction means "add quick," so the full instruction as we have it in the program adds 6 to the stack pointer, setting it back to where we started. In our example, that would be 5000.

What do we mean by "add quick"? You can use the *addq* instruction whenever you're adding an immediate value from 1 to 8. This instruction works faster than a regular add because the value is stored as part of the instruction, meaning the computer doesn't have to go out to memory to get it.

Address Register Direct Addressing

In the previous paragraph, when we added 6 to the stack pointer, we were using immediate addressing for the source argument (the 6) and address register direct addressing for the stack pointer. Whenever you directly manipulate the content of an address register (when you're not using it as a pointer), you are using address register direct addressing.

The *Cconin* Function

Now that we've displayed our prompt, we must allow the user to do what we've requested. In other words, we must accept a keystroke from the keyboard. In the next three lines of code, we place the function number for *Cconin* (1) onto the stack, call the operating system with *trap #1*, and then reset the stack pointer. Notice that this time we're adding only 2 to the stack pointer. This is because we put only the number 1, a word, onto the stack.

The function *Cconin* waits for a character from the keyboard, which it returns in D0. The character's ASCII code will be in the low byte of the low word, and the character's scan code will be in the low byte of the high word. What's the difference between an ASCII code and a scan code? Scan codes allow you to recognize keystrokes from special keys, like the cursor keys or the "F" keys, none of which have ASCII codes. Figure 4.1 shows D0 after a call to *Cconin*.

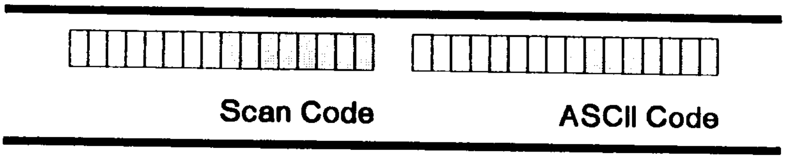


Figure 4.1 • Register D0 after Cconin

Comparisons

The next seven lines of code in our program make up an assembly language version of an IF...THEN...ELSE statement. In this part of the program, we need to figure out what key the user pressed and then jump to the appropriate section of the program. To do that, we must compare the ASCII value returned in the low byte of D0 with the ASCII values of the letters A, B, and C. The *cmp* (CoMPare) instruction, which compares its source and destination operands, is perfect for this task. (Note that the destination operand for this instruction must be a data register.)

The *cmp* instruction works by subtracting the source operand from the destination operand, without actually changing the values of the two operands. This operation sets the flags in the condition code register (CCR, the user byte of the status register), according to the results. For example, if the two operands are equal, subtracting the source from the destination yields a result of zero, which sets the CCR's zero flag. If the value in the source operand is larger than the one in the destination operand, we end up with a negative value when we do the subtraction, which sets the CCR's negative flag. By checking the CCR's flags, we can tell if the two operands are equal, or if one is larger than the other.

In the first line of this section of code, we compare the immediate constant *#LET_A*, which is the ASCII value of the letter A, with the value in D0, which was returned to us by the call to *Cconin*. See the suffix *.b* that we've added to the instruction? This means we are comparing bytes. How can this be? After all, *#LET_A* is a word value, and register D0 is a long word. How can we compare bytes? Easy. The instruction compares the low byte of *#LET_A* with the low byte of D0. If we had used *.w* as a suffix, we would have compared *#LET_A*'s full two bytes against the lower two bytes of D0. Get it?

Now, when the value in *#LET_A* is subtracted from the value in D0, if they are equal, we get a zero result, which sets the CCR's zero flag. In the next line, we use the *beq* instruction (Branch EQual) to test the zero flag. If the zero flag is set, we branch to the address following the instruction, or in this case, to the label *do_a*. If the result of the compare did not set the zero flag, the program drops down to the next instruction, which does the same compare with *#LET_B*, which is the ASCII value of the letter B. If the compare sets the zero flag, we branch to *do_b*. If it doesn't, the program drops down to the next instruction, which checks to see if the user typed the letter C. If he did, we branch to *do_c*. If he didn't, we use an "unconditional branch" to jump back to the top of the program and get another letter. When a program gets to a *bra* instruction, it branches to the destination address, no matter how the condition flags are set.

There are 15 branch instructions in 68000 assembly language, each of which branches based on certain flags in the CCR. All these instructions are listed in the reference section of this book. You might want to take a couple of moments right now to look them over.

Data Register Direct Addressing

Guess what? We've learned a new addressing mode. In the compare instructions discussed in the previous section, we used immediate addressing for the source argument and data register direct addressing for the destination argument, in this case, D0. Data register direct addressing is just like address register direct addressing, except we are manipulating a data register rather than an address register. In other words, when we directly manipulate the value of a data register, we are using data register direct addressing.

String Manipulation

Once the user has typed the proper key, we need to add the appropriate letter to the end of our string "You Pressed ." If you look at this string in the data section of our program, you'll see that we've left space at the end of the string for the letter. All we must do to complete the string is to copy the ASCII value of the letter into the correct position. If you're still awake, you realize this means we must move the low byte of one of our word constants into the fifteenth character of the string. What you probably haven't figured out is how to tell the assembler the correct address into which to copy the letter. If you look at the instruction *move.b #LET_A,key_msg+14*, you'll get a clue.

Remember that the address of our string is represented by the label *key_msg*. In other words, the address stored in *key_msg* is the address of the first character of our string. Looking at our program's data section, we see that the string starts with a carriage return/line feed combination, so the address in the label *key_msg* is the address of the carriage return. Because each character in the string is stored in successive bytes, to calculate the address of any character in the string, we add to the string's address the character's offset from the beginning of the string.

For example, let's say that *key_msg* equals 1000. Then the carriage return that starts off our string is at address 1000. More importantly, the line feed, which is the next character in the string, is at address 1001; the letter Y is at address 1002; and so on up to the period at the end of the string, which is at address 1015. This means that, to complete the string for the user, we have to copy the ASCII value of the letter he pressed into address 1014--or, since the label is our only real reference, *key_msg+14*.

In the *do_a* and *do_b* sections of the program, we do this copy and then use a *bra* instruction to jump to the part of the program that prints the now completed string to the screen. In the *do_c* section of the program, we don't need the branch, but we copy the letter the same way.

Absolute Addressing

Sheesh! We've already stumbled upon another addressing mode. In the instruction *move.b #LET_A, key_msg+14*, we're using absolute addressing to form the instruction's destination address. In other words, the operand *key_msg+14* is the actual address, or the absolute address, that we need. Whenever an operand is an actual address--that is, it isn't a register, and it isn't an immediate operand--we are using absolute addressing.

Finishing It Up

In the *prnt_strg* section of the program, we print our completed string to the screen. Here, we again call *Cconws* to print the string, and then wait for a key press with *Cconin*. In the last two lines of the text segment, we call GEMDOS function #1, *Pterm0*, which releases all memory used by the program and returns us to the desktop. If we had been using disk files in this program, *Pterm0* would have also closed any open files. Note that this is the only function that doesn't require that we reset the stack pointer after the call. Obviously, we can't reset the pointer, since this function ends the program. In this case, the system resets the stack pointer for us.

Conclusion

In this chapter, we wrote our first full-length assembly language program. The program may not do much, but it exposes us to many of the concepts we need to master in order to write more sophisticated programs. We'll be writing those more sophisticated programs soon, so make sure you understand everything presented here before moving on to the next chapter. If everything in the following summary makes sense to you, you're well on your way to becoming an assembly language programmer.

Summary

✓ The *dc* assembler directive lets us define constants. By adding the proper suffix, we can set up our constants as bytes, words, or longs.

✓ When we include the data we want to manipulate as part of our instruction, we are using immediate addressing.

✓ With address register indirect addressing with pre-decrement, we use the address register as a pointer to another part of memory. Furthermore, before using the address stored in the register, we subtract from it the length of the data referenced by the instruction.

✓ With address register direct addressing, we are directly manipulating the contents of an address register.

✓ Data register direct addressing is similar to address register direct, except we use a data register rather than an address register.

✓ The *addq* instruction adds an immediate value between 1 and 8 to the instruction's destination argument. This instruction works quickly because the immediate value is small enough to be stored as part of the instruction.

✓ The *cmp* instruction nondestructively subtracts the source operand from the destination operand, which sets the appropriate flags in the CCR. Using this instruction, we can compare arguments and determine whether or not they are equal.

✓ The *bra* instruction has 15 different forms, each of which allows us to branch to another part of our program, based on the status of the flags in the CCR.

✓ Whenever an operand is identified by an actual address, we are using absolute addressing.

✓ GEMDOS function #9, *Cconws*, writes a null-terminated string to the screen.

✓ To wait for a key press, we call GEMDOS function #1, *Cconin*.

✓ To terminate a program, we call GEMDOS function #0, *Pterm0*.

```
;------------------------------------------------------------
; THE ST ASSEMBLY-LANGUAGE WORKSHOP
; PROGRAM 2
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;------------------------------------------------------------
        text

get_letter:
        move.l  #prompt,-(sp)          ; put string address on stack.
        move.w  #9,-(sp)               ; function # to display string.
        trap    #1                     ; call the operating system.
        addq.l  #6,sp                  ; clean up the stack.

        move.w  #1,-(sp)               ; function # to get a character.
        trap    #1                     ; call the OS.
        addq.l  #2,sp                  ; clean up stack.

        cmp.b   #LET_A,d0              ; is the letter "A"?
        beq     do_a                   ; Yep!
        cmp.b   #LET_B,d0              ; not "A"; is it a "B"?
        beq     do_b                   ; sure is.
        cmp.b   #LET_C,d0              ; not "B"; is it a "C"?
        beq     do_c                   ; yes.
        bra     get_letter             ; invalid input; try again.
```

```
do_a:       move.b  #LET_A,key_msg+14   ; add ”A” to string.
            bra     prnt_strg           ; go print the string.

do_b:       move.b  #LET_B,key_msg+14   ; add ”B” to string.
            bra     prnt_strg           ; go print the string.

do_c:       move.b  #LET_C,key_msg+14   ; add ”C” to string.

prnt_strg:
            move.l  #key_msg,-(sp)      ; put string address on stack.
            move.w  #9,-(sp)            ; function # to display string.
            trap    #1                  ; call the operating system.
            addq.l  #6,sp               ; clean up stack.

            move.w  #1,-(sp)            ; function # to get a character.
            trap    #1                  ; call the OS.

            move.w  #Ø,-(sp)            ; function # to exit program.
            trap    #1                  ; call the operating system.
```

```
        data

LET_A       equ     65
LET_B       equ     66
LET_C       equ     67
prompt:     dc.b    13,10,"Press A, B, or C: ",13,10,0
key_msg:    dc.b    13,10,"You pressed  .",13,10,0
```

5
NUMBERS

Although we've talked about different number systems, until now, we've only glossed over the way your computer interprets numbers. Bytes, words, and long words are ways we group the bits that make up the computer's memory, so we can more easily interpret their values. However, in a computer, the same value may represent several different types of data, including unsigned integers, signed integers, characters, and addresses.

In this chapter, we'll start to examine numbers and the way they are interpreted. Along the way, we'll learn some new 68000 assembly language instructions and pick up some tips on programming style.

A Number in Memory

All by itself, stuck somewhere in memory, a number is just a number. Like a word in a book, the number lacks meaning until we attach some significance to it. For example, let's say we've got a byte of memory containing the number 48. That 48 can represent several things. It might be the ASCII value of the character "0," the low byte of an integer, the low byte of an address, or a binary value being used as a series of flags. It might, given the right hardware, even be part of a floating point number.

How does the computer know how to interpret that 48? It doesn't -- until we tell it how. Let's suppose the number 48 is stored in a data location labeled *num* and the number 50 is stored in D0. Look at the instruction on the next page.

```
        move.w    num,dØ
```

Does this instruction tell the computer how to interpret *num*? Nope. All we've done here is copy the two bytes at address *num* into the lower two bytes of register D0 (clobbering the 50 that was already there). The computer needs to know nothing about the number, except that it's two bytes long. Now, assuming the same conditions, look at this instruction:

```
        add.w   num,dØ
```

This instruction tells the computer to add the value in *num* to the value in the lower two bytes of D0 and store the result in D0. Did we tell the computer anything about our pal 48 this time? We sure did. By using the *add* instruction, we've implicitly told the computer that it should interpret both *num* and D0 as integers. After the instruction executes, D0 will contain the number 98. Now, how about this instruction, based on a line from chapter 4's program:

```
    move.b   num,key_msg+14
```

If you remember, *key_msg* is the address of a string constant. In this above instruction, we're placing the value 48 at the end of the string. When we call *Cconws* to print the string, that function will interpret 48 as the character "0." As you can see in the above examples, the value we stored in *num* can serve many different purposes.

Converting Numbers

Yes, numbers in assembly language are flexible entities. However, this flexibility comes with a price. As you know, when a value is input from the keyboard, our program receives it as either an ASCII value or a scan code. If we enter the number 1, our program receives the ASCII value 49. Now, what if we want to use the number we typed in a calculation? Obviously, 49 is not 1. If we try to add...oh, how about 15...to that number, we'll get 64, not 16, which is the answer we really want. This means that numbers as they are input from the keyboard must be converted into integers before they can be used mathematically. Moreover, once we are through with the math, if we want to print the results to the screen, the results must be converted back the other way, from integer to ASCII.

As you will see from program 3, when dealing with single digits, this conversion is easy. Consider the ASCII value of "1," which is 49. To change it into an integer, all we do is subtract 48, the ASCII value of "0." A piece of cake. To change it back into ASCII, all we do is add back the ASCII value of "0." However, when working with numbers of more than one digit, the conversion process can be tricky.

The Program

In the CHAP5 folder on your *Assembly Language Workshop* disk, you'll find the files PROG3.S and PROG3.TOS, the source and executable files for program 3, whose listing is printed following this chapter. If you'd like to assemble the program yourself, follow the instructions that came with your assembler or check this book's appendix A.

When you run the program, it asks you to enter a single digit. If you try to enter something other than a digit from 0-9, the program asks again. When you enter a valid digit, the program doubles the value of the number and then prints out the result. Yeah, I know; not exactly a commercial-quality program, but it does illustrate the basics of converting numbers between ASCII and integer. It also introduces us to a few new instructions.

Verifying the Input

Let's examine program 3's source code. We start by using *Cconws* to print a prompt and *Cconin* to get a character from the keyboard. Nothing new here. We covered these functions in previous chapters, so we won't go into any detail now.

After getting the character, however, we must verify that it is a digit between 0 and 9. Because we have the ASCII value of the character in the low byte of D0, we use a *cmp.b* to see if the ASCII value of the key pressed is less than the ASCII value of "0." If it is, the key typed wasn't a digit. After the compare, we use the *blt* (Branch Less Than) instruction, which sends us back to get another keypress if the user slips us some bad input.

The *blt* instruction branches to the destination label if the CCR's (condition code register, remember?) N (negative) flag is set and the V (overflow) flag is clear, or if the N flag is clear and the V flag is set. In plain English, the branch will occur if the *cmp*'s destination operand is less than its source operand. Remember that a *cmp* instruction works by subtracting the source operand from the destination operand. If the destination is smaller than the source, we'll get a negative result, which sets the CCR's N flag.

If the character passes the first test, all we know is that its ASCII value is larger than or equal to the ASCII value of "0." We now have to make sure that it's not larger than the ASCII value of "9," because then it wouldn't be a digit either. So, we do another compare, this time against the ASCII value of "9." If the *cmp*'s destination operand is larger than its source operand, the *bgt* (Branch Greater Than) sends us back for another try. (If you're interested in exactly which flags in the CCR are affected, check the reference in the back of this book. The way the flags are affected by instructions can be extremely complicated.)

Saving a Register

Once we get a digit, we drop past the compares, to where we move the low byte of D0 (the ASCII value of the key pressed) into the string, just as we did in chapter 4's program. We then copy D0 into D5. This is critical, because in the next section of the program, we're going to print the string. You may remember that *Cconws* returns in D0 the number of characters printed, meaning whatever was stored in D0 gets clobbered. We still need the number in D0, so we must move it to a new register. Forgetting to save your registers can create hard-to-find bugs. You've been warned.

Simple ASCII to Integer Conversions

After saving D0, we print the string, showing the user the key he pressed. (Hey, maybe he has a short memory.) Now, in order to perform arithmetic operations on the number, we must convert it from ASCII to integer. The instruction *subi.b #NUM_0,d5* subtracts the immediate value 48 (the ASCII value of "0") from the low byte of D5, giving us the character's integer equivalent. The instruction *subi* (SUBtract Immediate) subtracts the source operand from the destination operand, storing the result in the destination.

Now that we've converted the value into an integer, we can use it in arithmetic operations. To prove this, we use the *add.b d5,d5* instruction to add the low byte of D5 to itself, thus doubling the number. The *add* instruction adds the source argument to the destination argument, storing the result in the destination. It is in the same family as the *addq* instruction that we use to reset the stack pointer, except that, with *add*, we are not forced to add an immediate value from 1 to 8. We can use any "effective address operand." An effective address operand is simply an operand that denotes the address in which the data is stored. It's the opposite of "immediate."

Simple Integer to ASCII Conversions

Converting our digit into an integer and using it in arithmetic operations does us little good if we can't see the results. So, the next step is to take the results of the addition and convert it to ASCII for printing to the screen. This isn't as simple as converting from ASCII to integer, because the sum may be more than one digit. If, for example, we had entered the digit 6, the sum now stored in the low byte of D5 would be 12. We must convert this one byte value into a two-byte ASCII string. In the next chapter, we'll see how to do this conversion for any number (well, almost any number). This time around, we're going to cheat.

We know that the sum in the low byte of D5 has to be between 0 and 18, inclusive. In other words, all we must do is check to see if the sum is larger than 10. If it is, we place a 1 in the 10's place of the ASCII string, subtract 10 from the low byte of D5, convert the remaining byte to ASCII, and add it to the string after the 1. Of course, if we have only a one-digit sum, we can just convert the single byte and add it to the string.

To check if the sum in D5 is larger than 10, we use the instruction *subi.b #10,d5*. The *subi* (SUBtract Immediate) instruction subtracts an immediate source value from the destination, storing the result in the destination. In our case, 10 is subtracted from the low byte of D5 (notice the *.b* suffix). If the value in the low byte of D5 is less than 10 (meaning it will convert to a single ASCII digit), we get a negative result, which sets the CCR's N flag. We use the instruction *bmi one_digit* to test this condition, and jump to the label *one_digit* if the condition is true. The *bmi* (Branch MInus) instruction branches to the destination address when the CCR's N flag is set.

If we don't branch, we know the value in the low byte of D5 was greater than 9 (I say "was" because, after subtracting 10 from it, it is no longer greater than 9. It is, in fact, the number we need to convert to ASCII for the string's 1's place.), so we move the ASCII value of "1" into the string.

We then convert the byte remaining in the low byte of D5 to ASCII with the instruction *addi.b #NUM_0,d5*. The *addi* (ADD Immediate) instruction, as you may have guessed, adds an immediate source value to the destination argument, storing the result in the destination. After doing the conversion, we move the resultant ASCII value into our string and jump to the section of the program in which the string is printed.

For a single-digit sum, the *bmi* instruction branches to the label *one_digit*, where we convert the value in the low byte of D0 into a single ASCII digit and tack it onto the string. Notice that, in the conversion, we are adding *#NUM_0+10*. We need to add an extra 10 to replace the 10 we subtracted previously.

Programming Style

Before we finish this chapter, look at the style used in writing program 3. You should notice three things. First, all the address labels in the *text* segment are on lines by themselves. By doing this, the labels stand out better and can be as long and descriptive as we want, without messing up the format of the instruction columns. Also, this technique makes it easier to add new lines of code at a label's position. For example, If I had written the first program line as

```
get_number:   move.l   #prompt,-(sp)
```

to add an instruction at *get_number*, I'd have had to rewrite *move.l #prompt,-(sp)* on the next line, delete that instruction from the label's line, and then add the new line. With the label on its own line, all I do to add an instruction is add an extra line after the label and type the new instruction.

Notice also the way I've broken the instructions into small groups, each of which performs a specific task. For example, the first four lines print a prompt onto the screen, while the next three lines get a character from the keyboard. Dividing the code into logical units this way makes the code easier to read.

Finally, notice that, although in our first two programs, we commented just about every line, in this one we've changed our commenting style to show what task the code accomplishes, rather than what an instruction is actually doing. For example, in the program's first instruction *move.l #prompt,-(sp)*, we don't need to say that this line moves an address onto the stack. Anybody who knows 68000 assembly language can see that. That's not a helpful comment (except maybe for a novice just learning the instructions). Instead, the comment shows the string that will be printed. We've actually commented all four lines with this one comment.

Conclusion

Handling numbers in assembly language can be tricky, since there are so many ways in which they can be interpreted. In this chapter, we've gotten a quick introduction to some of the ways numbers can be manipulated. In forthcoming chapters, we'll gain more skills in this important area of assembly language programming.

Summary

- ✓ Numbers stored in memory can be interpreted as unsigned integers, as signed integers, as characters, and as addresses.
- ✓ The computer can't interpret a number until we tell it how, which we usually do implicitly with the instructions we choose.

- ✓ In order to use numbers in our programs, we must convert them from one type to another. Most often, we need to convert from ASCII to integer, or integer to ASCII.
- ✓ The *blt* instruction, following a *cmp*, branches to its destination address when the *cmp*'s destination argument is less than its source argument.
- ✓ The *bgt* instruction, following a *cmp*, branches to its destination address when the *cmp*'s destination argument is greater than its source argument.
- ✓ The branch instructions actually branch based on the flags in the CCR. For example, the *bmi* instruction branches to its destination address if the CCR's N flag is set. The N flag is set whenever an instruction yields a negative result.
- ✓ The *subi* instruction subtracts an immediate value from the destination operand, storing the result in the destination operand.
- ✓ The *addi* instruction adds an immediate value to the destination operand, storing the result in the destination operand.
- ✓ The *add* instruction adds an effective address operand to its destination operand, storing its result in the destination operand.
- ✓ When writing source code, we should be aware of programming style. Specifically, we should place most address labels on their own lines, as well as group instructions into logical groups based on their tasks.
- ✓ We should use informative comments, to make our code easier to understand and maintain.

```
;-----------------------------------------------------------------
; THE ST ASSEMBLY LANGUAGE WORKSHOP
; PROGRAM 3
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;-----------------------------------------------------------------

        text

get_number:
        move.l  #prompt,-(sp)           ; print "Type a single digit..."
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp

        move.w  #1,-(sp)                ; get a char from the console.
        trap    #1
        addq.l  #2,sp

        cmp.b   #NUM_0,d0               ; is the character less than "0"?
        blt     get_number              ; yep, go try again.
        cmp.b   #NUM_9,d0               ; is the character greater than "9"?
        bgt     get_number              ; yep, no good, so get another.

        move.b  d0,key_msg+14           ; add digit to string.
        move.b  d0,d5                   ; save contents of d0.
        move.l  #key_msg,-(sp)          ; print "You Pressed..."
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp

        subi.b  #NUM_0,d5               ; change from ASCII to int.
        add.b   d5,d5                   ; double the number.

        subi.b  #10,d5                  ; is number › 10?
        bmi     one_digit               ; no, only one digit.
        move.b  #NUM_1,add_msg+14       ; place "1" in 10s place.
        addi.b  #NUM_0,d5               ; convert remainder to ASCII.
        move.b  d5,add_msg+15           ; add final digit to string.
        bra     prnt_strg               ; go print the string.

one_digit:
        addi.b  #NUM_0+10,d5            ; convert number to ASCII.
        move.b  d5,add_msg+15           ; add number to string.

prnt_strg:
        move.l  #add_msg,-(sp)          ; print "The total is: "
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp
```

```
         move.w  #1,-(sp)                    ; get a character.
         trap    #1
         addq.l  #2,sp

         move.w  #0,-(sp)                    ; exit program.
         trap    #1

         data

NUM_0              equ     48
NUM_1              equ     49
NUM_9              equ     57
prompt:            dc.b    "Type a single digit number: ",0
key_msg:           dc.b    13,10,"You pressed  .",13,10,0
add_msg:           dc.b    "The total is:   .",13,10,0
```

6 Subroutines And Looping

In the previous chapters, we've written programs that execute their instructions sequentially--that is, starting at the top of the program and moving downward through the code until getting to the end. Occasionally, we've branched forward over code that we didn't want to execute, based on some sort of compare. However, we've always branched forward, never back.

To take advantage of subroutines, as well as to create efficient loops, we require special branch instructions. Granted, the branching instructions we've used so far can be used to implement subroutines and looping. However, because most programs use many subroutines and loops, the 68000 instruction set contains instructions specially designed for this purpose. In this chapter, we'll learn how these instructions work.

The Program

In the CHAP6 folder on your *Assembly Language Workshop* disk, you'll find two files: PROG4.S and PROG4.TOS. The former is the source code for the sample program, which is reprinted at the end of this chapter. The latter is the executable file, which was formed by assembling PROG4.S. If you'd like to assemble the program yourself, refer to the instructions that came with your assembler or check this book's appendix A.

When you run the program, six lines of text are printed to the screen, after which the program waits for a keypress and then returns to the desktop. Although the program seems to do very little, the source code contains many new instructions and concepts.

String Arrays

Before we can study the main program, we have to discuss a new assembler directive, as well as the data structures in the data segment. Look at the bottom of the program listing, where we've placed our program data. Immediately following the word "data" is the word "even." Remember when we discussed words and long words, we said they always must start on an even address? The assembler directive *even* tells the assembler to align the next piece of data on an even address. Because the next seven pieces of data in our data segment are all words or long words, each of them is then guaranteed to start on an even address, too.

Now, let's look at the data itself. The data labeled *s_cnt* is the number of strings we'll be printing, and the data starting at *s_adr* is a table of addresses. Each entry in the table is the address of a string we want to print. Notice that, to store a string's address, we can just use its label. The assembler automatically replaces the label with the proper address once that address is calculated.

Following the address table are the six strings, each with its own label, so we have a way to reference each string individually. You can see that the labels are the same labels we used to set up our address table. Why are we bothering with an address table? Well, we could have printed the six strings with six different calls to *Cconws*, but if you've done any programming, you know that, when you see the same process being done repeatedly, you've probably found a good place for a program loop. By enclosing a single call to *Cconws* within a loop and changing only the address we send to the function, we can greatly shorten our source code.

We could have used the strings themselves as our table and calculated within the loop the address of each string. We'd perform this calculation by adding to the starting address of the first string the number of characters between the start of the first string and the start of the string we want to print. For example, suppose the first string, *s1*, was stored starting at address 5000. We could calculate the address of the second string by counting the number of characters in *s1* and adding that count to 5000. String *s1* is made up of 32 characters, so if string *s1* started at addresss 5000, string *s2* would start at address 5032.

If all six strings were the same length, we could use the string length to easily calculate each string's address. However, they're not all the same length, which means that, to print each string in our loop, we'd need a table of string lengths. And, if we must use a table, a table of addresses is the simplest to implement, since we don't have to count characters, and we can change the length of the strings any time without having to change the table, too.

Now that we understand the way our data is set up, let's look at the subroutine itself.

Subroutines

A few chapters ago, we were introduced to the stack. So far, we've used the stack only for calling operating system functions, but, obviously, the stack has many other uses. One of those uses is the storage of return addresses when calling a subroutine. Look at the first three lines of our program. Here, we're getting ready to call a subroutine. First, we store the address of our string-address table into A5. Then, we store the number of strings to print in D5.

In the next line, we call a subroutine with the call *bsr print*. (The mnemonic *bsr* stands for Branch to SubRoutine.) When we call a subroutine, the computer takes the address of the next instruction and copies it onto the stack. It then loads the program counter with the *bsr*'s destination address--in our case, the address *print*. You already know that the program counter always contains the address of the next executable instruction. It should be obvious then that, by loading a new address into the program counter, we branch to a new part of the program.

In our sample program, that new part of the program is the subroutine that begins at the label *print*. After the branch, all the instructions that make up the subroutine are executed until the program reaches the *rts* (ReTurn from Subroutine) instruction, which causes the computer to pop the return address off the top of the stack and load it into the program counter, which sends us back to the instruction immediately following the *bsr*. Note that the computer assumes that the return address is on the top of the stack. In other words, if we place some of our own data on the stack (as we do in our subroutine *print*), we must be sure to reset the stack pointer to the return address. If we don't, the computer will load the program counter with whatever four bytes it finds on top of the stack, and we can expect to see some bombs on the screen.

Saving Registers

Subroutines, like any assembly language code, usually use registers. This means that, when you call a subroutine, you must be aware of registers that may get clobbered. An example of this is our *Trap #1* call to GEMDOS. Before we make this call, we must be sure that we have nothing important in registers A0, A1, A2, D0, D1, and D2, because these registers may be overwritten by the operating system. One way to avoid this problem in our own subroutines is to make sure that our subroutines never change any registers. To perform this seemingly impossible feat, all we must do is save the contents of whatever registers we use in the subroutine and then restore their contents before returning to the main program.

Why don't the operating system calls provide this simple courtesy? Because saving and restoring registers takes time. So that the operating system functions run as efficiently as possible, the OS designers decided to leave it up to us to avoid register conflicts by not putting important data in registers A0 through A2 and D0 through D2 when calling these functions.

It doesn't take much processor time to save a few registers, though, so in our own programs, we'll usually want to make our subroutines as transparent to the calling code as possible. Where can we store the registers? On the stack, of course. For example, we could store the registers like this:

```
move.l    a0,-(sp)
move.l    a1,-(sp)
move.l    d0,-(sp)
move.l    d1,-(sp)
```

This method requires a line of code for every register we want to save. If we save all the registers, that's 16 lines of code. And don't forget that we also must restore the registers at the end of the subroutine. That's another 16 lines of code. The fine folks who designed the 68000 assembly language anticipated this problem and created an instruction that allows us to save or restore as many registers as we want, all at once. The first line in our subroutine, *movem.l a0-a1/d0-d1,-(sp)*, moves A0, A1, D0, and D1 onto the stack. We could have saved all 16 registers just as easily, with the instruction *movem.l a0-a7/d0-d7,-(sp)*. To move just A0 and D0, we'd use the instruction *movem.l a0/d0,-(sp)*. Get the idea? With the *movem* (MOVE Multiple) instruction, we can signify a range of registers using a hyphen. We use a slash to separate address registers from data registers.

It's important that we remember to pull the saved values off the stack before we return from the subroutine, for two reasons. First, if we don't, our registers won't get restored to their original values. But more importantly, if we don't reset the stack pointer, our subroutine can't return properly. Do you know why? Remember that, when we called the subroutine, the computer stored the return address onto the stack. By saving our registers on the stack, we've buried the return address. Since the *rts* instruction expects to find the return address on top of the stack, we better be sure that it's there.

Looping

Getting back to our subroutine *print*, after we save the registers on the stack, we subtract 1 from D5, the register that holds the number of strings we want to print. We do this because of the way the looping instruction works.

Our loop comprises five lines of code, starting with the line immediately following the label *loop*. In the loop we call *Cconws* to write a string to the screen. After the four lines that handle our call to *Cconws*, you'll see the line *dbra d5,loop*. This is the instruction that controls the loop. The *dbra* (Decrement and BRAnch) instruction first subtracts 1 from the low word of the source data register (in this case, D5). If the data register is greater than -1 after the subtraction, the program branches to the source address (in this case, *loop*). If, due to the subtraction, the data register becomes -1, the loop is terminated, and program execution begins with the first instruction following the *dbra*.

You can now see why we had to subtract 1 from the register before we started the loop. Because the loop terminates when the count register becomes -1, the loop is performed once more than the counter value. Yeah, I know it's weird, but nobody promised you that 68000 assembly language programming was logical.

Just as the *bra* instruction has many forms (*bgt*, *bmi*, *bcc*, *blt*, etc.), so too does the *dbra* instruction. By using different forms of the decrement-and-branch instruction, we can check for two conditions at once. For example, the instruction *dbeq d5,loop* will terminate the loop both if the CCR's Z flag is set (possibly meaning that a previous *cmp* instruction compared equal pieces of data) or if the loop counter, D5, reaches -1. Using this instruction, we could do something like search a string of a certain length for a particular character, breaking out of the loop if we find the character or if we reach the end of the string. (The length of the string minus 1 would be loaded into the loop counter register.)

The many forms of the *dbra* instruction are listed in Appendix B. We won't go into detail on them at this time. However, in upcoming chapters, we'll bump into some form of this looping instruction often.

Address Register Indirect Addressing with Post-Increment

The first instruction in our loop, *move.l (a5)+,-(sp)*, introduces us to a new addressing mode. We're already familiar with address register indirect addressing with pre-decrement. That's what we always use when we move something onto the stack. We are, in fact, using it here, in the destination operand. Address register indirect with post-increment is the opposite. Instead of decrementing the register before we use it, we're incrementing the register after we use it.

In the above instruction, A5 contains the address of our string-address table. We loaded that address into the register before we called the subroutine, remember? If we had written the above instruction as *move.l a5,-(sp)*, without the parentheses and the plus sign on A5, we would be moving the value stored in A5 onto the stack. By putting parentheses around A5, we're changing it into a pointer (address register indirect addressing). The instruction *move.l (a5),-(sp)*, then, moves the value pointed to by the address in A5 onto the stack. When we add the plus sign to the instruction, *move.l (a5)+,-(sp)*, we are moving the value pointed to by the address stored in A5 onto the stack and then incrementing A5 by the appropriate number of bytes--here, four, because we're moving long words.

In our program loop, this addressing mode allows us to use A5 to point to each entry in our address table, one after the other, moving forward to the next address each time through the loop. Figure 6.1 compares the address register direct, address register indirect, and address register indirect with post-increment addressing modes. Notice how the contents of the stack and A0 change with each instruction.

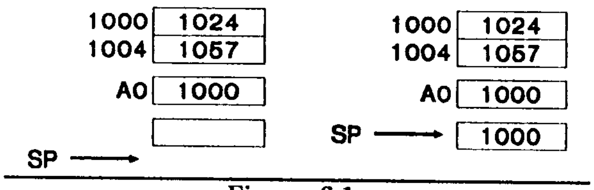


Figure 6.1a
move.l a0,-(sp)

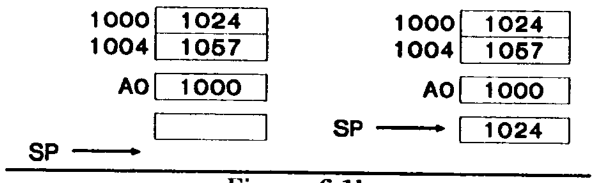


Figure 6.1b
move.l (a0),-(sp)

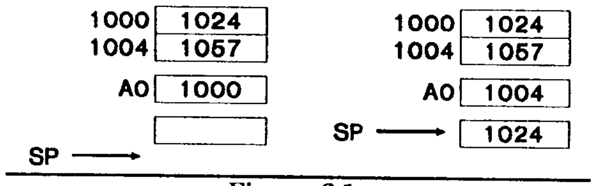


Figure 6.1c
move.l (a0)+,-(sp)

Restoring the Registers

As you know, before we return from our subroutine, we must restore the registers. We do this with the same *movem* instruction we used to save the registers, except now we reverse the source and destination operands, and we use address register indirect addressing with post-increment with the stack pointer instead of address register indirect addressing with pre-decrement. This addressing mode moves the stack pointer back down the stack as we restore each of our register's values. You can see this instruction in the sample program, right before the *rts* instruction, which returns us to the main program.

Subroutine Design

Subroutines are great for breaking our programs up into logical chunks, as well as dividing a program into sections of code that perform general tasks. By setting up our string-printing routine as a subroutine, for example, we can call it many times in the program without having to rewrite the code. Without the subroutine, every time we wanted to print a string array, we'd have to place another copy of the code into the program. In other words, subroutines help us avoid redundant program code.

Subroutines also allow us to use a structured, top-down programming approach. By using descriptive subroutine labels, we can quickly see what each subroutine call in our main program does, without having to study the details of the subroutine. We can think of a subroutine as a "black box," a device that gives us a certain output for a certain input. We are interested only in what the box does, not in how it does it. Once our subroutine is working properly, we can forget how it works and just use it.

To achieve the black-box effect, however, we must design our subroutines to be as free-standing as possible. This means passing all relevant data into the subroutine via registers or the stack. If a subroutine is to be a black box, it cannot access data external to itself, unless that data is passed into the subroutine.

Commenting a Subroutine

The black-box idea can be extended to include the subroutine's comments. At the beginning of every subroutine, we should include a comment block that describes not only what the subroutine does, but also where it expects to find its data, the type of output it generates, and what registers it affects. By commenting our subroutines this way, we can quickly see the requirements for calling the subroutine without having to look at the code. You should develop a consistent style for writing these "header" comments, so you can always tell at a glance everything you need to know about your subroutines.

Conclusion

In this chapter, we learned how to create program loops and how to implement subroutines. Both of these programming techniques allow us to streamline our code, by eliminating redundant instructions. We'll be using both of these techniques frequently in our programs from now on, so be sure you understand how they work.

Summary

- ✓ The *even* assembler directive can be used to ensure that data starts on an even address.
- ✓ The *dbra* group of instructions automatically decrements a loop counter and branches to a destination address if the counter is greater than -1. Many instructions in this group allow us to check two conditions in a loop: the loop

counter and the status of the Condition Codes Register (CCR).

✓ The *bsr* instruction allows us to branch to subroutines. The *rts* instruction returns us from a subroutine.

✓ In our subroutines, we should save registers before using them, and restore the registers before leaving the subroutine. This makes the subroutine more transparent to the calling code. The *movem* instruction allows us to save many registers at once.

✓ With address register indirect addressing with post-increment, we use a register as a pointer to the data we want to manipulate, and increment the register by the appropriate number of bytes after using it.

✓ With subroutines, we can break our programs up into logical tasks, which allows us to use top-down programming. Subroutines also help us to avoid redundant program code.

✓ Whenever possible, subroutines should be designed as "black boxes," so we need be concerned only with what a subroutine does and not with how it does it.

✓ Passing all needed data into the subroutine via registers or the stack, and composing descriptive "header" comments, are both important in creating a free-standing subroutine.

```
;-----------------------------------------------------------------------
; THE ST ASSEMBLY LANGUAGE WORKSHOP
; PROGRAM 4
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;-----------------------------------------------------------------------
        text

        move.l  #s_adr,a5               ; load table address.
        move.w  s_cnt,d5                ; load string count.
        bsr     print                   ; go print strings.

        move.w  #1,-(sp)                ; wait for keypress.
        trap    #1
        addq.l  #2,sp

        move.w  #0,-(sp)                ; back to desktop.
        trap    #1

;-----------------------------------------------------------------------
; This subroutine prints a string array to the screen.
;
; Input: a5--address of string-address table.
;        d5--number of strings in table.
;
```

```
; Registers changed: None
;--------------------------------------------------------------------
print:
        movem.l a0-a7/d0-d7,-(sp)          ; save registers onto stack.
        subq.l  #1,d5                      ; convert string count for loop.
loop:
        move.l  (a5)+,-(sp)                ; print string to screen.
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp

        dbra    d5,loop                    ; branch back for next string.
        movem.l (sp)+,a0-a7/d0-d7          ; restore registers.
        rts                                ; exit subroutine.

;--------------------------------------------------------------------

        data
        even

s_cnt:  dc.w    6

s_adr:  dc.l    s1
        dc.l    s2
        dc.l    s3
        dc.l    s4
        dc.l    s5
        dc.l    s6

s1:     dc.b    13,10,"If you ever want to print a",13,10,0
s2:     dc.b    "series of strings to the screen,",13,10,0
s3:     dc.b    "you can use a table of string",13,10,0
s4:     dc.b    "addresses, and move from one",13,10,0
s5:     dc.b    "string to the next by using an",13,10,0
s6:     dc.b    "assembly language loop.",13,10,0
```

7
Numbers Revisited

By now, you should be fairly comfortable with the basics of assembly language programming. So, in this chapter, we're going to write a full program: a number-guessing game. In this game, the computer picks a number from 1 to 100, and the player tries to guess it. Although this is a simple problem to program, it allows us to get into many new topics of discussion, including base storage segments, advanced number conversions, random number generation, and several new programming instructions.

The Program

In the CHAP7 folder of your *Assembly Language Workshop* disk, you'll find the files PROG5.S and PROG5.TOS, which are the source code and executable file for this chapter's sample program. If you'd like to assemble the program yourself, please refer to the instructions that came with your assembler or check this book's appendix A.

When you run the program, you're asked to guess a number from 1 to 100. After you've entered the number, the computer tells you whether your guess was too high or too low. Based on this hint, you should guess again. You get six tries, after which, if you haven't guessed correctly, the computer tells you the number and returns to the desktop. If you guess the number before your turns run out, the computer gives you a congratulatory message and exits to the desktop.

As you probably know, the quickest way to guess the number is to use a binary search--that is, keep splitting the possible numbers into two equal parts. For example, start with a guess of 50. If the computer says that that's too high, then guess 25. If that's too low, guess 37, and so on, eliminating 50% of the remaining numbers with each guess. Using this method, you can always guess the correct number within seven tries. That's why the game gives you only six!

Now that you know how the game works, let's start our study of the program with the data section.

Uninitialized Data Storage

Often in our programs, we need areas of memory that we can set aside for future use. Although these areas will contain data at some time during the program's execution, we don't know what that data will be. In other words, these areas start uninitialized. In this chapter's sample program, for example, we need a buffer in which to store a string. Because this string will be typed by the user after the program is run, we have no idea what the string will be. All we can do is supply space for the string and see that it gets placed there when the user types it.

Look at the end of this chapter's assembly language listing. You'll see the word *bss* followed by the line *buffer: ds.b 10*. The assembler directive *bss* (Block Storage Segment) sets aside an area for uninitialized data. You already know that *buffer* is a label that we can later use to refer to the address of this data. But what's that *ds.b*? The *ds* (Define Storage) assembler directive allows us to set aside space in memory. Here, we're setting up a storage area for 10 bytes. If we had used the assembler directives *ds.w* or *ds.l*, we would have been setting up storage for words or long words, respectively.

Why have a special area for this type of data? Why not just use the *dc* assembler directive? Since storage set up with the *dc* directive has specific data associated with it (the data used to initialize the area), all the data must be stored as part of the program file. If we had set up our buffer with the line *buffer: dc. b 0, 0, 0, 0, 0, 0, 0, 0, 0, 0*, using the define-constant directive instead of the define-storage directive, our program file would have been 10 bytes longer. (Note that, when initializing a large area of memory to the same value, we can use the *dcb* [Define Constant Block] assembler directive. To initialize *buffer* to 10 zeroes, we can write *buffer: dcb. b 10, 0*. The first number after the directive is the size of the area, and the second number is the value to which each location in the area should be initialized.)

When we use the *ds* assembler directive to define our storage, all the program loader had to do is grab 10 bytes from memory and initialize it to zeroes. It doesn't have to load it with any specific data. Furthermore, because the loader need not initialize our buffer with specific data, our program file need not store the data, so the file is that much smaller. Whenever you need uninitialized storage, you should define it in the *. bss* section of your program.

You should be able to figure out the rest of the data area yourself, so let's move on to our next topic, random numbers.

Generating Random Numbers

Although TOS provides a random-number generator, the value it supplies is too "raw" to be of any practical value. We must take the number returned from this function and change it to a number that falls within a range more suited to the current application. Most BASIC languages provide a random-number statement that returns a random number from 0 to *n*-1, where *n* is a value provided by the programmer. For example, a BASIC command like A=RANDOM(10) will return a value from 0 to 9. If we wanted a number from 1 to 10, we would use the statement A=RANDOM(10)+1.

We've used this typical BASIC implementation of random numbers as the model for our own random-number subroutine, found in the sample program at the label *rand*. This subroutine returns a random number from 0 to *n*-1. The value for *n* is stored in register D3 before calling the subroutine. A random number within the requested range is returned to the main program in D0.

Let's examine the *rand* subroutine. In the first line, we save all the registers--except D0. We don't want to save and restore D0, because we use it to return the final random number to the main program. In the next line, *move.w #RANDOM,-(sp)*, we move onto the stack the function number for the random-number call. This XBIOS function returns a 24-bit random number in D0. Notice that we've replaced the function number (17) with a descriptive name. By using constants this way, we don't need to remember all the function numbers when we want to call them; we just use their names. All the function calls in this program, and all upcoming programs in this book, use this technique. Constants are defined in the data section of a program.

The *trap #14* calls the XBIOS, which performs the random number function, returning the 24-bit value in D0. Unfortunately, a 24-bit random number can be larger than 100, the high end of the range we want. We must convert the number to the appropriate range. Because our subroutine is designed to return only 16-bit numbers, we first clear the upper word of D0, the register that contains the 24-bit random number. The line *andi.l #$0000FFFF,d0* does this for us. The instruction *andi* (AND Immediate) ANDs an immediate source value with an effective-address destination value, storing the result in the destination address. It can be used on all data lengths. (The "$" before the number tells the assembler that the value is a hexadecimal number.)

What's an AND? An AND operation compares the bits of each operand. If both bits are set, the result is 1. If either or both of the bits are not set, the result is 0. In other words, we can "mask out" any bits we want, by ANDing them with 0. Any bit ANDed with a 0 results in a 0, whereas any bit ANDed with a 1, retains its value, either 1 or 0. This operation is shown in figure 7.1.

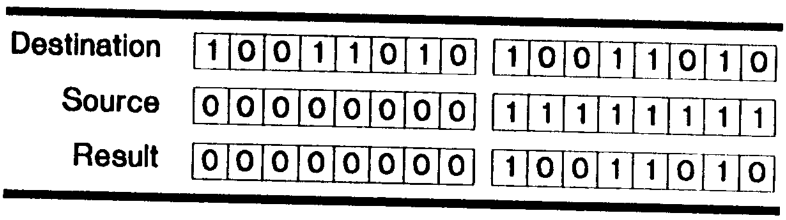


Figure 7.1

So, now that we have a 16-bit value, we can calculate our final random number by dividing D0 by the value in D3 (the value passed into the subroutine). The line *divu d3,d0* takes care of this. The instruction *divu* (DIVide Unsigned) divides a 32-bit destination data register by a 16-bit effective-address source operand and stores the result in the destination register, with the quotient in the low word and the remainder in the high word. Note that the source operand must be a 16-bit value and that both the quotient and remainder are also 16-bit values. Note also that, because this operation always operates on the same data lengths, it makes no sense to add a *.b*, *.w*, or *.l* suffix to the instruction. Figure 7.2 shows how this instruction works.

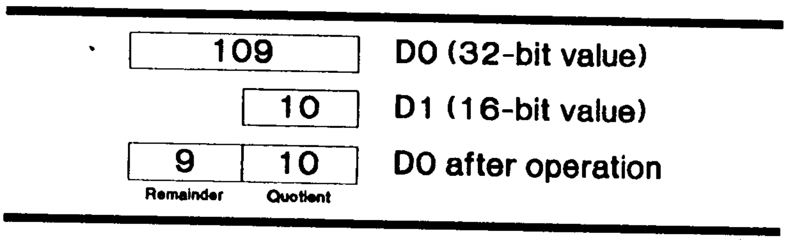


Figure 7.2

After the division, the high word of D0 (the remainder) contains a number within the requested range. However, before we can close up shop, we must clear out the quotient, which is stored in D0's low word. Sounds like a job for the *andi* operation, doesn't it? The line *andi.l #$FFFF0000,d0* clears the quotient from D0. We're not done yet, though. In order to have a 16-bit value, we must move our newly generated random number from the high word of D0 into the low word. Luckily, the 68000 assembly language instruction set contains the perfect instruction for this task. In our subroutine, the line *swap d0* swaps D0's high and low words--that is, the low word is moved into the high word, and the high word is moved into the low word. At last, we have a random number within the requested range. To finish up, we restore the registers and return from the subroutine.

16-bit Integer to ASCII

Now that we know how to perform division in assembly language, we can write a subroutine that'll convert any unsigned 16-bit integer into an ASCII string. We did a similar conversion with single-digit numbers in a previous chapter, but the process for larger numbers is a bit more complicated. The conversion is accomplished by dividing the integer by decreasing powers of ten. Each division yields a value that can be converted into one digit for the string.

For example, let's say we want to convert the integer 5,376 to a string. The highest possible 16-bit number we can have is 65,535, which means our most significant digit in the string will be in the 10,000th's place. So, we first divide 5,376 by 10,000 giving us a quotient of 0 and a remainder of 5,376. We convert the quotient into an ASCII character and add it to our string, giving us "0." Dropping to the next lowest power of 10, we divide the remainder of the previous operation by 1,000, giving us a quotient of 5 and a remainder of 376. We convert that digit, giving us the string "05." Dividing the remainder by 100 (the next lowest power of 10), we get a quotient of 3 and a remainder of 76. Adding the next digit to the string, we get "053." Using the same process, we divide by 10 and then finally by 1. After the last digit is converted, we have the string "05376."

The subroutine for this conversion is in the sample program, at the label *int2ascii16*. To call the subroutine,we place the address of a string buffer in A3 and the 16-bit integer we want to convert in D0. After the subroutine has done its processing, the ASCII string will be stored in the buffer pointed to by A3.

Look at the code. As with all subroutines, before we do anything, we save the registers. Because we aren't using any of the registers to return values to the main program, we'll save and restore all 16 registers. (Some programmers save only the registers that they know they'll use; others, to be on the safe side, save them all. Now, dear reader, you've been exposed to both methods.) Next, we clear D3, which we will use as a flag to tell us whether we've yet generated a digit greater than 0. Using this flag, we can avoid leading zeroes in our string. The *clr* (CLeaR) instruction clears the effective-address destination operand to all zeroes. After clearing the flag, we load D1 with the first divisor we'll use (10,000) to calculate the first digit in our string--that is, the digit in the 10,000th's place.

Now, we can start the conversion process, by dividing D0, which contains the integer we want to convert, by D1, which contains the current power of 10. We then check to see if we are producing an ASCII digit for the one's place. If we are, we have to allow a "0" digit, even though it's technically a leading zero. If we didn't allow this exception to our leading-zero rule, we'd be unable to convert the integer 0. After this check, we use the line *tst.w d3* to test whether we have yet generated a non-zero digit. (D3 isn't affected by the previous instructions; we will set D3 later in the subroutine when we convert the first non-zero digit.) The *tst* instruction may be used on any size data. All it does is compare the operand with zero and set the flags in the CCR accordingly. It does not change the data. If D3 is not 0, we have already generated a non-zero first character, so generating a "0" in the string is okay. Program execution jumps to *zero_ok*, where we move the quotient into our string buffer and convert it to ASCII, by adding the ASCII value of "0."

If, when we tested the flag D3, it was zero, we know we haven't yet generated a non-zero digit. So, we test the quotient of our division with another *tst* instruction. If the quotient is zero, we don't want to generate an ASCII digit in the string, because this digit would be a leading zero. We jump over the integer-to-ASCII conversion. If the digit is not a 0, we set D3 to 1 with the line *moveq #1,d3*, so we later know that we've found our first non-zero digit. (The *moveq* instruction is a more efficient form of the *move* instruction. It copies a signed immediate source operand with a value between -128 and +127 to the destination operand.) We then convert the digit to ASCII.

Despite the value of the flag D3, we eventually end up at *next_digit*, where we divide D1 by 10, to get the next smallest power of ten for a divisor. We then test the divisor for a zero result, which would mean that the conversion is complete. If the divisor is still a non-zero value, we zero out the low word of D0 (the quotient), use the *swap* instruction to move the remainder into the low word, and then loop back to convert the next digit.

When the divisor, D1, becomes zero, program execution branches to *add_null*, where we add a null to the string, restore the registers, and return to the main program.

Local Labels

Look closely at the labels in the *int2ascii16* subroutine. You'll notice that all but the first and last start with periods. A label that starts with a period is a "local" label, one that is valid only in the area between the previous and the next non-local label. Look at this example:

```
label1:
.label2:
.label3:
label4:
```

Here, *label2* and *label3* are said to be local to *label1*. These labels can be referenced only in the code that lies between *label1* and *label4*. If you try to reference one of the local labels in code before *label1* or after *label4*, you will get an "undefined symbol" error. On the positive side, we can reuse any of the local label names elsewhere in our program without confusing the assembler. As far as the assembler is concerned, the local labels don't even exist, except within their locality. Using local labels is another way we can make our subroutines more like "black boxes." Using this technique, we can define labels in our main program without fear of accidentally duplicating labels in our subroutines. In addition, local labels allow us to use more general-purpose labels like *loop* or *out* as often as we like, as long as we will never need to reference them outside their locality.

ASCII to 16-bit Integer

Converting an ASCII number to an integer is similar to converting an integer to ASCII, except we use multiplication instead of division. The algorithm works like this: Check the first digit to be sure that it is a digit and not some other character. If it's okay, convert the digit to an integer, and add it to an accumulator register, which was initialized to 0. Then, check the next character to see if it's the null marking the end of the string. If it's not, the character just converted is multiplied by 10, and the program loops back for the next digit, converting it and adding it to the accumulator. This process continues until the end of the string, at which point the accumulator contains the full, converted integer.

Let's say we have the string "426." We subtract the ASCII value of "0" from the ASCII "4," which leaves us with the integer 4. We then add 4 to the accumulator, giving us 4. Because we aren't yet at the end of the string, we multiply our accumulator by 10, giving us 40. We then convert the "2" to 2 and add it to the accumulator, giving us 42. We're still not at the end of the string, so we again multiply the accumulator by 10, giving us 420. After converting the "6" to 6, and adding the result to the accumulator, we end up with 426, the final integer. See? It's easy!

The code that does this conversion is found in the sample program, at the label *ascii2int16*. As always, we first save the registers--except D0, which we use to return the final integer to the main program. We then clear D0, which we also use as the accumulator. (The *clr* instruction sets its effective-address destination argument to 0.) We also clear register D3, which we will use as a work register. We start the conversion by comparing the byte pointed to by A3 to the ASCII values of "0" and "9," to make sure we have a digit. If we don't have a digit, we branch to *.digit_error*, where we load D0 with our error code of -1.

If the digit is okay, we load it into D3 with the line *move.b (a3)+,d3*. Notice that we're using address register indirect addressing with post-increment in this instruction. After the instruction, A3 will point to the next character in the string. After loading the character into our work register, we convert it into an integer, by subtracting the ASCII value of 0. We then add the resultant integer to D0, our accumulator. Next, the line *tst.b (a3)* compares the byte pointed to by A3 to 0. If that byte is a 0, we've reached the end of the string, and we branch to *.out*. If we haven't reached the end of the string, we multiply the accumulator by 10, with the instruction *mulu #10,d0*. The *mulu* (MULtiply Unsigned) instruction multiplies two unsigned 16-bit operands (an effective-address source operand and a destination data register) and stores the 32-bit result in the destination register. Because this instruction's arguments are always word length, the instruction should not have a suffix.

After the multiplication, we branch back to get the next digit. When the conversion is complete, we branch to *out*, where we restore the registers and return to the main program.

The Main Program

Now that we understand the subroutines, I don't think we need to go through the main program line by line. You should be able to figure out easily how it works. If you can't, you may need to reread previous chapters, to review some of the concepts. There are, however, two things in the main program we've not yet discussed–a new instruction and a new function call.

Near the top of the program, you'll see the line *cmpi. l #7,d4*, which compares the immediate value 7 with the data register D4. The *cmpi* (CoMPare Immediate) instruction is just another form of the *cmp* instruction. It is used specifically to compare an immediate value of any data length to the destination effective-address argument. In our previous programs, there were several places where we should have used this form of the *cmp* instruction but, for simplicity's sake, we used the regular *cmp* instead. Fortunately, most assemblers are intelligent enough to substitute the more efficient *cmpi* wherever appropriate.

A little farther down the program listing, you'll see a block of code that includes the line *move. w #CCONRS,-(sp)*. This block of code is the GEMDOS function call to read a string from the keyboard. To use this call, we must first place two parameters on the stack: the address of the buffer where the string should be stored and the function number (10). Before the call, we must also place in the first byte of the buffer the maximum number of characters we want to read, usually the buffer length minus 2. The function terminates either when the user enters the maximum number of characters or when the user presses Return.

After the call (using *trap #1*), both the second byte of the buffer and the register D0 contain the number of characters actually read. The string itself is stored starting at the third byte of the buffer. Figure 7.3 shows the buffer after the user typed "hello" in response to a call to *Cconrs*. Note that the string is not zero-terminated.

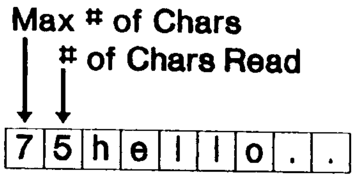


Figure 7.1

Conclusion

In this chapter, we covered a lot of new material. You should study the following summary carefully and review any topics that are not clear to you. As with most technical areas of study, it is essential that you understand the covered material before you move on to new topics. You should now fully understand why we need to convert numbers between ASCII and integer, as well as how to do it. Also, you should be comfortable with the new instructions we covered in this chapter. Take some time to write a small program or two of your own, incorporating what you've learned so far.

Summary

- ✓ The *.ds* assembler directive allows us to set up uninitialized storage areas in the *.bss* area of our program. Because these areas in memory do not need to be initialized to specific values, initialization information need not be stored with the program's executable file.
- ✓ The *.dcb* assembler directive allows us to initialize large blocks of data to the same value.
- ✓ The XBIOS function #17 (Random) returns a 24-bit random number in register D0.

✓ The *trap #14* exception allows us to call XBIOS functions.

✓ By using constants to replace function numbers with descriptive names, we make our source code easier to write and easier to read.

✓ The *andi* instruction ANDs an immediate source operand with an effective-address destination operand and stores the result in the destination operand.

✓ The *divu* instruction divides a 16-bit source operand into a 32-bit data register, storing the result in the destination. The quotient is stored in the low word, and the remainder is stored in the high word.

✓ The *swap* instruction switches the values found in the destination argument's low word and high word.

✓ The *tst* instruction compares its effective-address operand to 0. The operand is not modified, but the flags in the CCR are set accordingly.

✓ The *moveq* instruction is a more efficient form of *move*, which should be used when moving immediate values between -128 and +127.

✓ Labels that start with a period are considered local to the area between the two closest non-local labels.

✓ The *clr* instruction sets its effective-address destination argument to 0.

✓ The *mulu* instruction multiplies an unsigned effective-address source operand and an unsigned 16-bit value stored in a data register. It stores the 32-bit result in the destination register.

- ✓ The *cmpi* instruction compares an immediate source operand with an effective-address destination operand, setting the appropriate flags in the CCR.
- ✓ GEMDOS function #10 (Cconrs) retrieves a string from the keyboard.

```
;------------------------------------------------------------
; THE ST ASSEMBLY LANGUAGE WORKSHOP
; PROGRAM 5
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;------------------------------------------------------------
        text
        move.l  #100,d3                 ; get random # from 0 to 99.
        jsr     rand
        addq.l  #1,d0                   ; convert range to 1 to 100.
        move.l  d0,d2                   ; save random number.

        clr.l   d4                      ; init turn counter.

get_number:
        addq.w  #1,d4                   ; increment counter.
        cmpi.l  #7,d4                   ; game over?
        beq     loser                   ; yep.
        move.l  #buffer,a1              ; get address of buffer.
        move.w  #6,d1                   ; init loop counter.
clr_str:
        move.b  #0,(a1)+                ; zero out buffer.
        dbra    d1,clr_str

        move.l  #prompt,-(sp)           ; print "Guess a number..."
        move.w  #CCONWS,-(sp)
        trap     #1
        addq.l   #6,sp

        move.b   #3,buffer              ; init character count.
        move.l   #buffer,-(sp)          ; get string from keyboard.
        move.w   #CCONRS,-(sp)
        trap     #1
        addq.l   #6,sp

        move.l   #buffer+2,a3           ; address of first char in buffer.
        jsr      ascii2int16            ; convert to integer.

        cmpi.l   #-1,d0
        bne      no_error

        move.l   #msg1,-(sp)            ; print "Did not type number."
        move.w   #CCONWS,-(sp)
        trap     #1
        addq.l   #6,sp
        bra      get_number             ; go get another number.

no_error:
        cmp.w    d2,d0                  ; was the number guessed?
        beq      guessed                ; yes.
        bgt      too_high               ; nope, number was too low.
        move.l  #msg2,-(sp)             ; "Too low."
        bra     prnt_str                ; go print the string.
too_high:
        move.l  #msg3,-(sp)             ; "Too high."
prnt_str:
        move.w  #CCONWS,-(sp)           ; print string.
        trap    #1
        addq.l  #6,sp
        bra     get_number              ; go get another number.

guessed:
        move.l  #msg4,-(sp)             ; print "You guessed it!"
        move.w  #CCONWS,-(sp)
        trap    #1
        addq.l  #6,sp
        bra     out

loser:
        move.l  #msg5,-(sp)             ; print "Too many guesses..."
        move.w  #CCONWS,-(sp)
        trap    #1
        addq.l  #6,sp

        move.l  d2,d0                   ; move random # to d0.
        move.l  #buffer,a3              ; get address of string buffer.
```

```
         jsr      int2ascii16                ; convert # to string.

         move.l   #buffer,-(sp)              ; print correct number.
         move.w   #CCONWS,-(sp)
         trap     #1
         addq.l   #6,sp

out:
         move.w   #CCONIN,-(sp)              ; wait for keypress.
         trap     #1
         addq.l   #2,sp

         move.w   #PTERMØ,-(sp)              ; back to desktop.
         trap     #1

;-------------------------------------------------------------------
; This subroutine returns a 16-bit random number from Ø to n-1.
;
; Input: High number+1 (n) of range in d3. (A value of 1Ø yields a
;        random number from Ø to 9.)
; Output: dØ will contain 16-bit random number in the requested
;         range.
; Registers changed: dØ.
;-------------------------------------------------------------------
rand:
        movem.l a0-a7/d1-d7,-(sp)      ; save registers.
        move.w  #RANDOM,-(sp)          ; get 24-bit random number.
        trap    #14
        addq.l  #2,sp
        andi.l  #$0000FFFF,d0          ; clear high word.
        divu    d3,d0                  ; convert to requested range.
        andi.l  #$FFFF0000,d0          ; clear the quotient.
        swap    d0                     ; place the remainder in low word.
        movem.l (sp)+,a0-a7/d1-d7      ; restore registers.
        rts

;---------------------------------------------------------------
; This subroutine converts a 16-bit unsigned integer into a
; null-terminated string.
;
; Input: Address of buffer in a3.
;        Integer to convert in d0.
; Output: The buffer will contain the resultant null-terminated string.
; Registers changed: NONE.
;---------------------------------------------------------------
int2ascii16:
        movem.l a0-a7/d0-d7,-(sp)      ; save registers.
        clr.l   d3                     ; init leading-zero flag.
        move.l  #10000,d1              ; init divisor.
.convrt:
        divu     d1,d0                  ; calculate place value.
        cmpi.l   #1,d1                  ; are we at the one's place?
        beq      .zero_ok               ; if so, "0" always ok.
        tst.w    d3                     ; already have a non-zero char?
        bne      .zero_ok               ; yes, so zeroes okay.
        tst.w    d0                     ; is result zero?
        beq      .next_digit            ; yes.
        moveq    #1,d3                  ; set leading-zero flag.
.zero_ok:
        move.b   d0,(a3)                ; move result to buffer.
        add.b    #ZERO,(a3)+            ; change digit to ascii.
.next_digit:
        divu     #10,d1                 ; calculate next divisor.
        tst.w    d1                     ; are we done yet?
        beq      .add_null              ; sure are.
        move.w   #0,d0                  ; clear result from low word.
        swap     d0                     ; put remainder in low word.
        bra      .convrt                ; convert next digit.
.add_null:
        move.b   #0,(a3)+               ; add null to string.
        movem.l  (sp)+,a0-a7/d0-d7      ; restore registers.
        rts
end_int2ascii16:

;------------------------------------------------------------------
```

```
; This subroutine converts a string of ascii digits into a 16-bit
; unsigned integer.
;
; Input: Address of Ø-terminated string in a3.
; Output: 16-bit integer in dØ. A return of -1 signifies an error.
; Registers changed: dØ.
;------------------------------------------------------------------
ascii2int16:
        movem.l aØ-a7/d1-d7,-(sp)

        clr.l   dØ                      ; clear accumulator
        clr.l   d3                      ; clear work reg.
.cnvt_digit:
        cmp.b   #ZERO,(a3)              ; char < "Ø"
        blt     .digit_error            ; yep. error.
        cmp.b   #NINE,(a3)              ; char > "9"
        bgt     .digit_error            ; yep.  error.
        move.b  (a3)+,d3                ; get digit.
        sub.w   #ZERO,d3                ; convert to integer.
        add.l   d3,dØ                   ; add to accumulator.
        tst.b   (a3)                    ; end of string?
        beq     .out                    ; yep.
        mulu    #1Ø,dØ                  ; nope.
        bra     .cnvt_digit             ; go do next digit.
```

```
.digit_error:
        move.l  #-1,d0                        ; set error condition.
.out:
        movem.l (sp)+,a0-a7/d1-d7             ; restore registers.
         rts
end_ascii2int16:

;----------------------------------------------------------------------

        data

ZERO    equu    48
```

```
msg4:   dc.b    13,10,"You got it!",13,10,0
msg5:   dc.b    13,10,"Too many guesses.  The number was: ",0

        bss

buffer: ds.b    10
```

8

Manipulating The System Clock

Okay, get those thinking muscles loose and limber. In this chapter, we're going to write a program that will actually do something useful--namely, allow you to set your system clock. Although this program uses no dialog boxes or menu bars, or any other of the GEM-type objects to which you've grown accustomed, it's still lengthy. Because assembly language instructions can accomplish so little compared with most high-level language commands, even simple programs can sometimes be large.

Of course, by writing larger programs, we open many new topics of discussion. In this chapter, we'll get an introduction to macros, write a couple of new general-purpose subroutines, discuss two new addressing modes, and, of course, learn a couple of new assembly language instructions.

The Program

In the CHAP8 folder on your *Assembly Language Workshop* disk, you'll find the files PROG6.S and PROG6.TOS, which are the source code and executable file, respectively, for this chapter's sample program. If you'd like to assemble the program yourself, please refer to the instructions that came with your assembler or check this book's appendix A.

When you run the program, you are shown the current system date and asked whether you'd like to change it. Type "Y" to change the date or any other key to leave the date as it's shown. If you choose to change the date, the program asks you to enter the current date in the format mm/dd/yy.

Next, the program shows the system time and asks whether you'd like to change it. As with the date, type "Y" to change the time or any other key to leave it as it is. If you choose to change the time, you are prompted for the new time, after which you get the message "Press Return at exact minute." When you press Return, the system clock is set to the time you entered, and the program returns you to the desktop.

Note that the program is semi-intelligent and won't let you enter an invalid date or time. Note also that you can place this program into your AUTO folder, so you'll be prompted for the current date and time each time you boot your system.

A First Look at Macros

Look at the first few lines of the source code for our clock-setting program, and you'll see our first "macro." (If you use MadMac, see appendix A.) Macros allow us to replace many instructions with a single line, thus greatly reducing the size of our source code, especially when we repeat the same task often in different parts of the program.

One task we repeat often in our programs is printing lines of text to the screen. Printing a string to the screen requires four lines of code, and we must enter these four lines each time we want to do it. But by using a macro, we can force the assembler to add those lines automatically wherever we need them.

Look at the second line of the main program: *print msg1*. This is a macro call. This line tells the assembler to place the code that makes up the macro *print* at this location in the program. The word *msg1*, which is the address of the text we want to print, is this macro's parameter. A macro may have many parameters.

Now, look at the macro again. The first line, *print macro*, tells the assembler that we are about to define a macro called *print*. The assembler will read and remember every line of code from this point on, until it sees the assembler directive *endm*, which tells the assembler that it has reached the end of the macro.

Our *print* macro comprises the code necessary to call the GEMDOS function *Cconws*, which we use to print text to the screen. The first line of the macro, *move.l #\1,-(sp)*, is the first line of the call to *Cconws*, where we place the address of the string to print onto the stack. The *\1* tells the assembler to place the first parameter in this location. In other words, with our first macro call, *print msg1*, *msg1* is substituted for the *\1*, so that the final line reads *move.l #msg1,-(sp)*. If our macro call had more than one parameter, each parameter would be given a number based on its order in the call. For example, if we wanted to pass into our macro both the address of the string to print and the *Cconws* function number, the call would look like this:

```
print   msg1,#CCONWS
```

and the macro itself would look like this:

```
print   macro
move.l  #\1,-(sp)
move.w  #\2,-(sp)
trap    #1
addq.l  #6,sp
endm
```

The process of replacing a macro call with the lines that make up the macro is called "macro expansion." During macro expansion, the assembler replaces the macro call with the macro code, placing all parameters is their proper places. Note that, even though our source code shows only a single line wherever we call a macro, the object code will contain all four instructions that make up the macro. In other words, macros don't shorten the assembled program; they shorten only the source code. We can think of them as abbreviations. Macros also make our source code easier to read, by reducing a long series of instructions to a single, descriptive line (descriptive, that is, if we use good macro names that make the macro's purpose clear).

Macros can be very complicated. There are even directives that allow macros to generate different code based on the parameters passed to them. In a way, macros are a miniature programming language unto themselves. In upcoming programs, we'll learn more about macros and their amazing abilities.

Now that we have been introduced to macros, let's move on and discuss the program, starting with our new subroutines.

Getting the System Date

The first thing we must do in our program is display the current system date. After all, how can the user know whether it must be changed if he can't see it? Because this is the sort of general function that we may need in many of our programs, it's a perfect candidate for a subroutine.

If you look about half way down the program listing, you'll see the subroutine that retrieves the system date and returns it to the main program. After our call to this subroutine, the "raw" unformatted GEMDOS date will be in D0. To make things easier for the programmer, this subroutine also extracts the day, month, and year from the raw date and returns them in D5, D6, and D3, respectively.

Getting the raw GEMDOS date from the system requires only a standard GEMDOS function call to *Tgetdate*. In our subroutine, the first thing we do after saving the registers, is call *Tgetdate*, which returns the system date in the low word of D0. What? All the date information in two lousy bytes? Yep. To translate the system date into something more usable, we have to do a lot of fiddling with the value returned to us in D0. The date is returned in GEMDOS format: the day is stored in bits 0-4, the month is in bits 5-8, and the year is in bits 9-15. This bit organization is illustrated in figure 8.1.

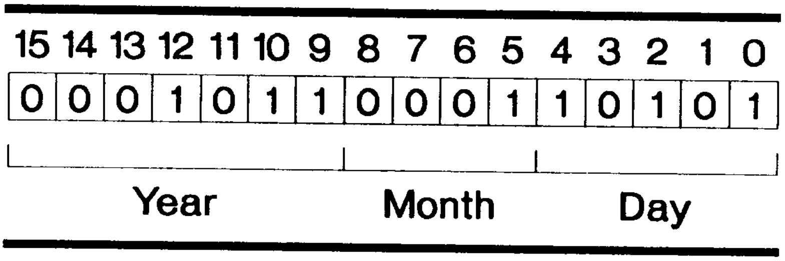


Figure 8.1

Obviously, if the makers of GEM decided to format the date like this, there must be some way to use assembly language to extract the day, month, and year. We already know about the AND operation, which allows us to mask out unneeded bits in a piece of data. To get the day portion of the date, all we must do is mask out the upper 11 bits of the raw date. Before doing that, though, we need to make a copy of the date, so when we mask out the bits, we still have the full date stored somewhere. In the subroutine, the lines that follow make a copy of the date and mask out the unneeded bits, leaving us with the day portion of the date:

```
move.l  d0,d5
andi.l  #$1f,d5
```

Unfortunately, a simple AND operation is not enough to extract the month and year portions of the date. If, for example, we were to mask out all but the month bits, we'd have the right bits isolated, but they'd be in the wrong position. To have the correct value, the bits must be shifted into the lowest bits of the register. Suppose that, after an AND operation, we are left with the bits 0000000011100000 in the low word of the register. Looking at bits 4-8, which determine the month portion of the date, we get the bit pattern 0111, which equals 7, or the month of July. But when looking at the entire 16 bits in the low word of the register, we have 224 not 7. In order for the register to equal 7, we have to shift the month bits, 0111, into the lowest bits of the register.

And shift is exactly the right word to use. The *lsr* (Logical Shift Right) instruction allows us to shift to the right the bits of any data register or memory location. (No address registers allowed.) The vacated bits on the left are filled with zeroes, whereas the bits shifted out on the right are lost forever, except the last bit shifted, which is stored in the CCR's C and X flags. This operation can be performed on bytes, words, or long words. However, when used on a memory location, the operation is restricted to word size data and can shift only one bit at a time. A shift on a memory location looks like this: *lsr.w addr*. The destination operand *addr* may be a label or an address (i.e. $4758).

So, the following lines, taken from our subroutine, extract the month portion of the date:

```
move.l  d0,d6
lsr.l   #5,d6
andi.l  #$f,d6
```

The first instruction above gets a working copy of the date into register D6. Then we shift the bits in the register five places to the right, which places the bits of the month portion of the date into bits 0-3. Finally, we use an AND operation to clear all bits except the lower four, which leaves us with the correct value for the month. The entire process is illustrated in figure 8.2.

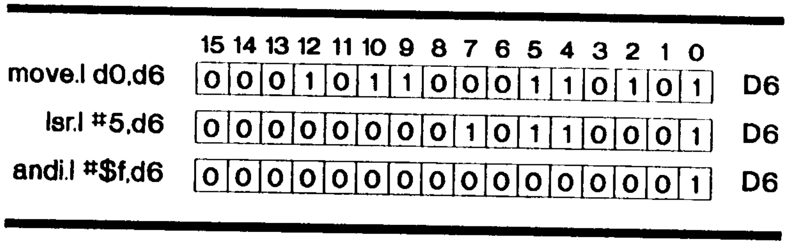


Figure 8.2

In the shift instruction that follows, we used an immediate operand to specify the number of bits to shift. We also can store the shift count in a data register, like this:

```
lsr.l d0,d1
```

In the above example, register D1 is shifted a number of bits equal to the value stored in D0. Note that the shift count cannot be greater than 8. If we need to shift an operand more than eight times, we must use more than one shift operation.

Now that we have the day and month portion of the date, all we need is the year. We extract the year from the raw date the same way we got the month, by shifting to the right and masking out the unneeded bits. The only differences are that we must now shift the register nine bits to the right, and then we must AND it with $7f rather than $f, since we aren't masking as many bits. Another important difference is that, even after getting the year extracted from the rest of the date, we still need to do a little more work. The system year is actually the year-1980. In other words, to get the correct year, we must add 1980 to the value we extract from the date. But we don't handle that wrinkle in the subroutine; we just return the year as found in the GEMDOS date.

Getting the System Time

Getting the system time isn't much different from getting the system date. We must use the same sort of shift and AND operations to extract the seconds, minutes, and hours from the GEMDOS-format time. In our sample program, we have a subroutine that takes care of all this, returning the raw time in D0, the seconds in D5, the minutes in D6, and the hours in D3.

Let's look at that subroutine now, located in the listing at the label *get_time*. In the first line, we save all the registers except the ones we'll use to return values to the main program. After that is our call to the GEMDOS function *Tgettime*, which returns the system time in D0, with the seconds in bits 0-4, the minutes in bits 5-10, and the hours in bits 11-15. Note that the seconds are actually the seconds/2. To get the current seconds, we must multiply the value retrieved from the raw date by 2. This means, of course, that the seconds are accurate only to the nearest even second. Figure 8.3 shows the bit organization for the raw GEMDOS time.

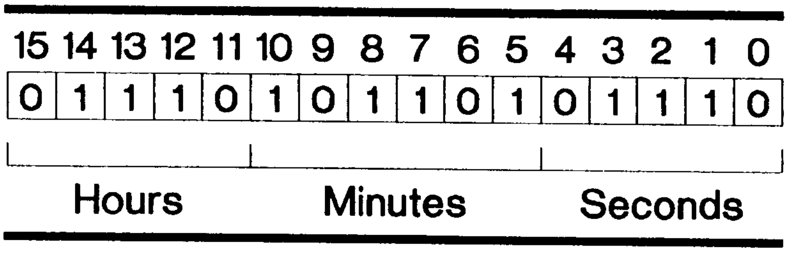


Figure 8.3

After we retrieve the raw GEMDOS time, we use a series of shifts and ANDs to extract the seconds, minutes, and hours into D5, D6, and D3, respectively. After discussing the method by which we worked with the GEMDOS date, I don't think we need to go into any detail here. The source code speaks for itself.

Setting the System Date

When retrieving the system date, we must use a series of right shifts and ANDs to extract each element of the date. When setting the system date, we must do the opposite. That is, we use a series of left shifts and ORs.

Look at the section of code starting with the label *change_date*. Here, we get a new date from the user, convert it into GEMDOS format, and then use it to reset the system clock. To get a new date from the user, we just use a call to *Cconrs*, prompting the user for the date in the standard mm/dd/yy format. After we get the date string from the user, we must do the following:

- Extract the day, month, and year portions.
- Convert the day, month, and year strings to integers.
- Check that each of the date values is valid. (Months must be 1-12, days must be 1-31, and years must be any pair of digits.)
- Merge the day, month, and year integers into a GEMDOS format date.
- Call *Tsetdate* to install the new date into the system clock.

In the source code, we first convert the year, which, after the call to *Cconrs*, is located in *buffer+8* and *buffer+9*. We copy the two characters that make up the year into their own buffer, and add a null to the string, with the lines

```
move.b buffer+8,buf2
move.b buffer+9,buf2+1
move.b #0,buf2+2
```

We then convert the string to an integer with a call to our subroutine *ascii2int16*. After the conversion, we check that the subroutine didn't return a -1, which would indicate an error, usually a non-digit entered by the user. Then, since it's not likely that anyone will be setting their system clocks for a time in the past, we check to see whether the value entered for the year was less that 91, and assume that any value less than 91 is meant to be a date after the year 2000. (Hey, you never know. Someone may still be using these machines then. The 21st century is less than a decade away!) Remember that the year in a GEMDOS-format date is actually the year-1980, so we must subtract 80 if the entered year is still in this century, or add 20 if the year value entered was 0 to 90. For example, if the year entered was 01, we must assume that the user meant the year 2001. Adding 20, we get 21, which is 2001-1980.

Once we get the year converted to an integer, we must add it to D3, which is the register we'll be using to store the GEMDOS-format date. First, we copy D0, which contains the converted integer for the year, into D3. This operation puts the year value in D3's lower bits. According to the GEMDOS format, we must put the year into bits 9-15. We do this by using the *lsl* (Logical Shift Left) instruction to shift the bits nine positions to the left. Because we can shift only eight places at once, we must use the *lsl* instruction twice to get the year in its correct position.

The *lsl* instruction is the opposite of the *lsr* instruction. It shifts bits to the left, bringing zeroes in at the right and losing the bits that are shifted out the left, except the last bit shifted out, which is stored in the CCR's C and X flags. Except the direction of the shift, this instruction works exactly like the *lsr* instruction.

After we have the year tucked safely away, we copy the digits that make up the month with the lines

```
move.b buffer+2,buf2
move.b buffer+3,buf2+1
```

This time we don't need to add a null to the string, because the null we previously placed in *buf2+2* is still there. After converting the string to an integer, we check to be sure that the value falls between 1 and 12. If it does, it's ready to be added to our date-storage register, D3. Because the value for the month must be stored in bits 5-8, we need to shift our month five bits to the left with the line *lsl.l #5,d0*.

Now, how are we going to merge the month value in D0 with the date in D3, without clobbering the data already in D3? With a neat operation called OR. An OR is the opposite of an AND, so whereas we use an AND to mask bits out, we use an OR to add bits in. When we OR two bits together, we get a result of 1 when either or both of the bits are equal to 1, and we get a 0 when both bits are equal to 0. Figure 8.4 shows how the OR operation works.

Destination	1	0	0	1	1	0	1	0	0	0	1	1	0	1	0	1
Source	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
Result	1	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1

Figure 8.4

The idea is that wherever we have a 0 in the source operand, the value in the destination operand will remain unchanged, but wherever we have a 1 in the source operand, a 1 will appear in the destination. This allows us to select exactly where in the destination operand we will copy in bits from the source operand. However, when merging two values into a single value, we must be sure that all the target bits in the destination are 0. Otherwise, we may not end up with a copy of the source bits in the target bits. (For example, if there is a 1 in the target bit and a 0 in the source bit, we will end up with a 1 in the target, when we really want a 0.)

Gasp! Got all that? So, to merge the month value with the year, we use the line *or.l d0,d3*. In 68000 assembly language, the instruction *or* ORs the source operand with the destination operand, leaving the result in the destination. You cannot use address registers with this instruction, and either the source or destination arguments must be a data register (both may be). Figure 8.5 summarizes the operations required to combine the month and the year values, using shifts and ORs.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
move.l d0,d3	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	DO
Year (1991-1980) to D3	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	D3
lsl.l #8,d3	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	DO
First shift	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	D3
lsl.l #1,d3	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	DO
Year now in position	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	D3
jsr ascii2int16	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	DO
Month (7) now in DO	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	D3
lsl.l #5,dO	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	DO
Month now in position	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	D3
or.l d0,d3	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	DO
Month merged with year	0	0	0	1	0	1	1	0	1	1	1	0	0	0	0	0	D3

Figure 8.5

After getting the year and month stored in D3, we convert the day similarly, copying the string into *buf2* and converting it to an integer, which we check for validity by comparing it against 1 and 31, the lowest and highest days possible. After getting a valid day, we OR it into the rest of the date.

To actually set the date, we must move the new GEMDOS-format date onto the stack and call the GEMDOS function *Tsetdate*. Note that when we move the date onto the stack, we are not copying the entire register, but rather only the low word. This means we update the stack pointer by adding 4, not 6--two bytes for the date and two bytes for the function number.

Setting the System Time

As you've probably guessed, setting the system time is a lot like setting the system date. Look at the program listing, starting at the label *change_time*. First, we retrieve the new time from the user in the hh:mm format, storing the string in *buffer*. Then we move the two characters that make up the hour into their own buffer, *buf2*, and add a null to the string. A call to our subroutine *ascii2int16*, converts the year string into an integer, which we then check to make sure it falls between 0 and 23 inclusive (24-hour, military time) and shift it into its correct bit position in the register.

After we've determined that the hour is valid, we convert the minutes similarly, by moving the characters into the buffer and converting them to integer. We verify that the minutes are valid by checking them against 0 and 59; then we shift them into their correct bit position and OR them in with the hours. Now that we have a valid GEMDOS-format time in the register, we prompt the user to press Return on the exact minute. We obviously can't expect the user to enter the seconds by hand. He'd have to be fast on the keyboard!

Finally, a call to the GEMDOS function *Tsettime* sets the system clock to the requested time. Before calling the function, we must move the new GEMDOS-format time and the function number (45) onto the stack. Again, as with the call to *Tsetdate*, we do not copy the entire register onto the stack, but rather only the low word.

Now, we know everything there is to know about setting the ST's system clock. (Well, almost everything.) Let's finish this chapter by learning two new addressing modes.

Address Register Indirect Addressing with Displacement

Go back to the top of the program listing and find the label *find_null*. Four lines down from the label are these instructions:

```
move.b #SPACE,1(a0)
move.b #0,2(a0)
```

In the destination argument for both these instructions, you can see an example of address register indirect addressing with displacement. You already know that when we place an address register in parentheses, we are using it as a pointer. This is address register indirect addressing. When we place a number in front of the parentheses, we are indicating a "displacement" from the address in the register. Simply, the source address is the address in the register plus the displacement. If, for example, register A0 in the above example contained the address $5000, the destination address in the first line would resolve to $5001, and the destination address in the second line would resolve to $5002. The displacement may be any 16-bit value--that is, a value from -32,768 to +32,767.

As a further example, let's say we have a string of characters stored at the address *string*. The addresses of the first three characters could be referenced using absolute addressing, like this:

```
string
string+1
string+2
```

If we load the address of the string into an address register with the instruction *move.l #string,a0*, we could then reference the first three characters in the string like this:

```
(a0)
1(a0)
2(a0)
```

These two methods are equivalent, as long as the value of *string* is first loaded into A0.

Address Register Indirect Addressing with Index

Now, we can go one step further and add an index to our address register indirect addressing. Look at the very top of the program, where we get the system date and display it on the screen. Upon return from our subroutine *get_date*, register D6 contains the date's month value in integer form. To print the correct month from our table of strings, we need to use the value in D6 (after a little converting, which we'll discuss later) to index our table of string addresses. For example, if the system time returned a month value of 2, we need to print the string "February." The address of this string is the second address in our table, which is located at the label *months* in the data segment.

Probably your first thought is to use address register indirect addressing with displacement, like this: D6(A0). Not a bad idea. Unfortunately, it's not a legal instruction. A data register can never be used as a displacement. It can, however, be used as an index, which, when you get right down to it, is actually much the same. The proper format for the argument, then, is *x(An,Dn)*, which is an example of address register indirect addressing with index. (The displacement *x* can be any value from -128 to +127. The register number *n* can be any number from 0 to 7.) The address is computed by adding all three values together. Suppose A0 contains the address $5000, and D0 contains the value 5. In this case, the operand 1(A0,D0) would resolve to $5000+1+5 or $5006. In this addressing mode, we can use either a data register or an address register as the index, and we can specify a word or long word index by adding a suffix to the index register (i.e. D0.W or D0.L).

In the sample program, we really don't need the displacement, but because we want to use a data register as an index, we have to use this addressing mode. No problem. We just use a displacement of 0.

Indexing an Address Table

As I mentioned previously, before we can use the value in D6 as an index, it must be converted. The reason for this conversion may not at first be obvious. In our program we have a table of addresses, with each entry in the table being the address of text for a month of the year. The addresses are in the order of the months; that is, the first is the address of the string "January," the second is the address of the string "February," the third is the address for the string "March," and etc.

When we get the value for the month back from our subroutine *get_date*, it will be an integer from 1 to 12, just as you would expect. But because we must use this value as an offset--a value to be added to the starting address of the table--we must first subtract 1. Why? Well, our table of string addresses starts at the label *months*. The first address, the one that points to the string "January," is the table entry *months+0*, right? Yet, the integer value for January is 1. If we use this index value as it is, we will be end up with February rather than January.

After changing the month value into a correct index by subtracting 1, we then have to consider that each entry in our address table is four bytes long, not one. (Addresses are always long words.) If January's entry is at *months+0*, February's entry is at *months+4*, and March's entry is at *months+8*. This is why, after subtracting 1, we multiply the result times 4.

Conclusion

This chapter has been the most challenging yet. We covered a lot of material, learned several new instructions, addressing modes, and system functions. You should study this chapter carefully, reviewing previous chapters where necessary. You also should try writing your own programs and experiment with the various addressing modes and new instructions. Go over the following summary to be sure you understand all the material presented here.

In the next chapter, we'll learn how to send a text file to the printer.

Summary

✓ Macros allow us to shorten our source code, by replacing many instructions with a single macro call.

✓ The GEMDOS function call *Tgetdate* (#42) retrieves the system date and returns it in the low word of register D0. The day is in bits 0-4, the month is in bits 5-8, and the year-1980 is in bits 9-15.

✓ The *lsr* instruction shifts the bits of its destination operand to the right, filling the vacated left-hand bits with zeroes and storing the last bit shifted out the right in the CCR's C and X flags.

- ✓ The *lsl* instruction shifts the bits of its destination operand to the left, filling the vacated right-hand bits with zeroes and storing the last bit shifted out the left in the CCR's C and X flags.
- ✓ The GEMDOS function *Tgettime* (#44) returns the system time in register D0, with the seconds/2 in bits 0-4, the minutes in bits 5-10, and the hours in bits 11-15.
- ✓ The *or* instruction ORs its source operand with its destination operand, storing the result in the destination.
- ✓ The GEMDOS function *Tsetdate* (#43) sets the system date according to the GEMDOS-format date passed to the function on the stack.
- ✓ The GEMDOS function *Tsettime* (#45) sets the system time according to the GEMDOS-format time passed to the function on the stack.
- ✓ With address register indirect addressing with displacement, *x(An)*, an address is calculated by adding the address in register *An* and the displacement *x*.
- ✓ With address register indirect addressing with index, *x(An,Dn)*, the address is the sum of three values: the value stored in the index register, the value of the displacement, and the address stored in register *An*.

```
;----------------------------------------------------------------------
; THE ST ASSEMBLY LANGUAGE WORKSHOP
; PROGRAM 6
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;----------------------------------------------------------------------

;----------------------------------------------------------------------
; This macro prints a string to the screen, using the Cconws GEMDOS
; function. It requires one parameter, the address of the string
; to print.
;----------------------------------------------------------------------
print   macro
        move.l  #\1,-(sp)
        move.w  #CCONWS,-(sp)
        trap    #1
        addq.l  #6,sp
        endm

;----------------------------------------------------------------------
; MAIN PROGRAM
;----------------------------------------------------------------------
        text
        jsr      get_date                  ; get system date.

        print    msg1                      ; "The current date is..."
        subi.l   #1,d6                     ; calculate offset of string.
        mulu     #4,d6                     ; convert to long word count.
        move.l   #months,a0                ; get address of first string.
        move.l   0(a0,d6),-(sp)            ; print current month string.
        move.w   #CCONWS,-(sp)
        trap     #1
        addq.l   #6,sp

        move.l   #buffer,a3                ; convert day to string.
        move.l   d5,d0
        jsr      int2ascii16

        move.l   #buffer,a0                ; get address of string...
find_null:
        cmp.b    #0,(a0)+                  ; and look for end.
        bne      find_null
        move.b   #COMMA,-(a0)              ; add comma to string.
        move.b   #SPACE,1(a0)              ; add space to string.
        move.b   #0,2(a0)                  ; add null to string.

        print    buffer                    ; print day string.
        move.l   d3,d0                   ; convert year to string.
        jsr      int2ascii16

        print    buffer                  ; print year string.
        print    crlf                    ; move to next line.
        print    msg2                    ; "Would you like to change it?"

        move.w   #CCONIN,-(sp)           ; get answer from user.
        trap     #1
        addq.l   #2,sp
        move.l   d0,d3
        print    crlf                    ; go to new line.

        cmp.w    #UC_Y,d3                ; is answer "Y"?
        beq      change_date             ; yep, go change date.
        cmp.w    #LC_Y,d3                ; is answer "y"?
        bne      time                    ; no, go do time.

change_date:
        print    msg3                    ; "Enter new date..."
        move.b   #9,buffer               ; number of chars to input.
        move.l   #buffer,-(sp)           ; get date string from user.
        move.w   #CCONRS,-(sp)
        trap     #1
        addq.l   #6,sp
        print    crlf

        clr.l    d3                      ; init register for date.
        move.b   buffer+8,buf2           ; get first digit of year.
        move.b   buffer+9,buf2+1         ; get second digit of year.
        move.b   #0,buf2+2               ; add null.
        move.l   #buf2,a3                ; get address of string.
        bsr      ascii2int16             ; convert year to integer.
        cmpi.w   #-1,d0                  ; did number convert ok?
        beq      date_no_good            ; nope, generate error.
        cmpi.l   #91,d0                  ; if year < 91, then...
        blt      year_2000               ; assume it's 21st century.
        subi.l   #80,d0                  ; convert to GEMDOS year format.
        bra      store_year
year_2000:
        addi.l   #20,d0                  ; convert to GEMDOS year format.
store_year:
        move.l   d0,d3                   ; store year and...
        lsl.l    #8,d3                   ; shift it to...
        lsl.l    #1,d3                   ; its correct position.

        move.b   buffer+2,buf2           ; get first digit of month.
        move.b   buffer+3,buf2+1         ; get second digit of month.
        bsr     ascii2int16             ; convert month to integer.
        cmpi.w  #1,d0                   ; if value < 1, the month...
        blt     date_no_good            ; is invalid.
        cmpi.w  #12,d0                  ; and if value > 12 the...
        bgt     date_no_good            ; month is invalid.
        lsl.l   #5,d0                   ; shift month to correct position.
        or.l    d0,d3                   ; add month to date.

        move.b  buffer+5,buf2           ; get first digit of day.
        move.b  buffer+6,buf2+1         ; get the second digit of day.
        bsr     ascii2int16             ; convert day to integer.
        cmpi.w  #1,d0                   ; The day value can't...
        blt     date_no_good            ; be less than 1 or...
        cmpi.w  #31,d0                  ; greater than 31.
        bgt     date_no_good
        or.l    d0,d3                   ; add day to date.

        move.w  d3,-(sp)                ; move date onto stack.
        move.w  #TSETDATE,-(sp)         ; GEMDOS function #.
        trap    #1
        addq.l  #4,sp
        print   msg5                    ; "New date set."
        bra     time
```

```
date_no_good:
        print   msg4                    ; "Not a valid date!"
        bra     change_date

time:
        jsr     get_time

        print   msg6                    ; "The system time is..."
        move.l  #buffer,a3              ; get address of string buffer.
        move.l  d3,d0                   ; change integer hours...
        jsr     int2ascii16             ; to ascii string.

        move.l  #buffer,a0              ; get address of hour string.
        move.l  #-1,d4                  ; init character count.
find_null2:
        addq.l  #1,d4                   ; increment character count.
        cmpi.b  #0,(a0)+                ; found null yet?
        bne     find_null2              ; nope.

        move.l  #buffer,a0              ; reset pointer to string.
        cmpi.l  #1,d4                   ; only one character?
        bne     no_leading_zero         ; nope, don't need the "0".
        move.b  (a0),1(a0)              ; move character over.
        move.b  #ZERO,(a0)              ; add "0".
no_leading_zero:
        move.b  #COLON,2(a0)        ; add ":" to string.
        move.l  #buffer+3,a3        ; addr of minutes part of string.
        move.l  d6,d0               ; convert the integer...
        jsr     int2ascii16         ; minutes to ascii string.
        move.l  #buffer+3,a0        ; adr of start of minutes.
        move.l  #-1,d4              ; init character count.
find_null3:
        addq.l  #1,d4               ; increment character count.
        cmpi.b  #0,(a0)+            ; found null yet?
        bne     find_null3          ; no.
        move.l  #buffer+3,a0        ; reset pointer.
        cmpi.l  #1,d4               ; only one character?
        bne     no_leading_zero2    ; no, don't need "0".
        move.b  (a0),1(a0)          ; move character over.
        move.b  #ZERO,(a0)          ; add "0".
no_leading_zero2:
        move.b  #0,2(a0)            ; add null.
        print   buffer
        print   crlf

        print   msg2                ; "Would you like to change..."
        move.w  #CCONIN,-(sp)       ; get answer from user.
        trap    #1
        addq.l  #2,sp
        move.l   d0,d3                  ; save keystroke.
        print    crlf                   ; go to new line.

        cmpi.w   #UC_Y,d3               ; is answer "Y"?
        beq      change_time            ; yep, go change time.
        cmpi.w   #LC_Y,d3               ; is answer "y"?
        bne      out                    ; no, leave program.

change_time:
        print    msg7                   ; "Enter new time..."
        move.b   #6,buffer              ; set # of chars to input.
        move.l   #buffer,-(sp)          ; get date from user.
        move.w   #CCONRS,-(sp)
        trap     #1
        addq.l   #6,sp
        print    crlf

        clr.l    d3                     ; clear register for new time.
        move.b   buffer+2,buf2          ; move first char of hour.
        move.b   buffer+3,buf2+1        ; move second char of hour.
        move.b   #0,buf2+2              ; add null to hour string.
        move.l   #buf2,a3               ; get addr of hour string.
        jsr      ascii2int16            ; convert to integer.
        cmpi.w   #0,d0                  ; if value < 0...
        blt      time_no_good           ; hour value is invalid.
```

```
cmpi.w   #23,d0                      ; if value › 23...
bgt      time_no_good                ; hour value no good.
lsl.l    #8,d0                       ; shift hours into place.
lsl.l    #3,d0
move.l   d0,d3                       ; add hours to new time.

move.b   buffer+5,buf2               ; move first char of minutes.
move.b   buffer+6,buf2+1             ; move second char of minutes.
jsr      ascii2int16                 ; convert minutes to integer.
cmpi.w   #0,d0                       ; minutes value can't...
blt      time_no_good                ; less than 0. nor...
cmpi.w   #59,d0                      ; can it be...
bgt      time_no_good                ; greater than 59.
lsl.l    #5,d0                       ; shift minutes into place.
or       d0,d3                       ; add minutes to new time.
print    msg9                        ; "Press Return on minute..."

move.w   #CCONIN,-(sp)               ; wait for keypress.
trap     #1
addq.l   #2,sp

move.w   d3,-(sp)                    ; move new time onto the stack.
move.w   #TSETTIME,-(sp)             ; put function # on stack.
trap     #1                          ; set new time.
```

```
        addq.l  #4,sp
        bra     out

time_no_good:
        print   msg8                    ; "Not a valid time!"
        bra     change_time

out:
        move.w  #PTERMØ,-(sp)           ; back to desktop.
        trap    #1

;------------------------------------------------------------
; This subroutine gets the system date and separates it into the
; year, month, and day portions.
;
; Input: None.
; Output: Raw GEMDOS date in dØ, day in d5, month in d6, and
;         year in d3.
; Registers changed: dØ, d5, d6, and d3.
;------------------------------------------------------------
get_date:
        movem.l aØ-a2/d1-d2,-(sp)       ; save registers.

        move.w  #TGETDATE,-(sp)         ; get system date.
        trap    #1
        addq.l  #2,sp

        move.l  d0,d5                       ; get copy of date.
        andi.l  #$1f,d5                     ; mask out all but day bits.

        move.l  d0,d6                       ; get copy of date.
        lsr.l   #5,d6                       ; shift month into low bits.
        andi.l  #$f,d6                      ; mask out all but month bits.

        move.l  d0,d3                       ; get copy of date.
        lsr.l   #8,d3                       ; shift year into low bits.
        lsr.l   #1,d3
        andi.l  #$7f,d3                     ; mask out all but year bits.
        addi.l  #1980,d3                    ; convert from GEMDOS year.

        movem.l (sp)+,a0-a2/d1-d2           ; restore registers.
        rts

;----------------------------------------------------------------
; This subroutine gets the system time and separates it into its
; hour and minute portions.  The seconds are ignored.
;
; Input: None.
; Output: Raw GEMDOS time in d0, seconds in d5, minutes in d6,
;          and hours in d3.
; Registers changed: d0, d5, d6, and d3.
;------------------------------------------------------------
get_time:
        movem.l a0-a2/d1-d2,-(sp)        ; save registers.
        move.w  #TGETTIME,-(sp)          ; get system time.
        trap    #1
        addq.l  #2,sp

        move.l  d0,d5                    ; get copy of time.
        andi.l  #$1f,d5                  ; mask all bits but seconds.

        move.l  d0,d6                    ; get copy of time.
        lsr.l   #5,d6                    ; shift minutes into low bits.
        andi.l  #$3f,d6                  ; clear all but minutes.

        move.l  d0,d3                    ; get copy of time.
        lsr.l   #8,d3                    ; shift hours to low bits.
        lsr.l   #3,d3
        andi.l  #$1f,d3                  ; mask out unneeded bits.

        movem.l (sp)+,a0-a2/d1-d2        ; Restore registers.
        rts
```

```
;------------------------------------------------------------------
; This subroutine converts a 16-bit unsigned integer into a
; null-terminated string.
;
; Input: Address of buffer in a3.
;        Integer to convert in d0.
; Output: The buffer will contain the resultant null-terminated string.
; Registers changed: NONE.
;------------------------------------------------------------------
int2ascii16:
        movem.l a0-a7/d0-d7,-(sp)       ; save registers.
        clr.l   d3                      ; init leading-zero flag.
        move.l  #10000,d1               ; init divisor.
.convrt:
        divu    d1,d0                   ; calculate place value.
        cmpi.l  #1,d1                   ; are we at the one's place?
        beq     .zero_ok                ; if so, "0" always ok.
        tst.w   d3                      ; already have a non-zero char?
        bne     .zero_ok                ; yes, so zeroes okay.
        tst.w   d0                      ; is result zero?
        beq     .next_digit             ; yes.
        moveq   #1,d3                   ; set leading-zero flag.
.zero_ok:
        move.b  d0,(a3)                 ; move result to buffer.
        add.b   #ZERO,(a3)+             ; change digit to ascii.
.next_digit:
        divu    #10,d1                  ; calculate next divisor.
        tst.w   d1                      ; are we done yet?
        beq     .add_null               ; sure are.
        move.w  #0,d0                   ; clear result from low word.
        swap    d0                      ; put remainder in low word.
        bra     .convrt                 ; convert next digit.
.add_null:
        move.b  #0,(a3)+                ; add null to string.
        movem.l (sp)+,a0-a7/d0-d7       ; restore registers.
        rts
end_int2ascii16:

;-----------------------------------------------------------------
; This subroutine converts a string of ascii digits into a 16-bit
; unsigned integer.
;
; Input: Address of 0-terminated string in a3.
; Output: 16-bit integer in d0. A return of -1 signifies an error.
; Registers changed: d0.
;-----------------------------------------------------------------
ascii2int16:
        movem.l a0-a7/d1-d7,-(sp)
        clr.l   d0                          ; clear accumulator
        clr.l   d3                          ; clear work reg.
.cnvt_digit:
        cmp.b   #ZERO,(a3)                  ; char < "0"
        blt     .digit_error                ; yep. error.
        cmp.b   #NINE,(a3)                  ; char > "9"
        bgt     .digit_error                ; yep.  error.
        move.b  (a3)+,d3                    ; get digit.
        sub.w   #ZERO,d3                    ; convert to integer.
        add.l   d3,d0                       ; add to accumulator.
        tst.b   (a3)                        ; end of string?
        beq     .out                        ; yep.
        mulu    #10,d0                      ; nope.
        bra     .cnvt_digit                 ; go do next digit.

.digit_error:
        move.l  #-1,d0                      ; set error condition.
.out:
        movem.l (sp)+,a0-a7/d1-d7           ; restore registers.
        rts
end_ascii2int16:

;----------------------------------------------------------------------
```

```
                data

SPACE           equ     32
COMMA           equ     44
SLASH           equ     47
ZERO            equ     48
ONE             equ     49
TWO             equ     50
THREE           equ     51
FIVE            equ     53
NINE            equ     57
COLON           equ     58
UC_Y            equ     89
LC_Y            equ     121

PTERM0          equ     0
CCONIN          equ     1
CCONWS          equ     9
CCONRS          equ     10
TGETDATE        equ     42
TSETDATE        equ     43
TGETTIME        equ     44
TSETTIME        equ     45
```

```
months: dc.l    jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec

jan:    dc.b    "January ",0
feb:    dc.b    "February ",0
mar:    dc.b    "March ",0
apr:    dc.b    "April ",0
may:    dc.b    "May ",0
jun:    dc.b    "June ",0
jul:    dc.b    "July ",0
aug:    dc.b    "August ",0
sep:    dc.b    "September ",0
oct:    dc.b    "October ",0
nov:    dc.b    "November ",0
dec:    dc.b    "December ",0

msg1:   dc.b    "The current system date is ",0
msg2:   dc.b    "Would you like to change it?",13,10,0
msg3:   dc.b    "Enter new sytem date (mm/dd/yy): ",0
msg4:   dc.b    "Not a valid date!",13,10,0
msg5:   dc.b    "New date set.",13,10,0
msg6:   dc.b    "The current system time is ",0
msg7:   dc.b    "Enter new system time (hh:mm): ",0
msg8:   dc.b    "Not a valid time!",13,10,0
msg9:   dc.b    "Press RETURN at exact minute.",13,10,0
```

```
crlf:   dc.b    13,10,0

        bss

buffer: ds.b    10
buf2:   ds.b    3
```

9

Simple File Handling And Printer Output

It's pretty tough to write a full-scale program that doesn't, at one time or another, access disk files or send data to a printer. In this chapter, we'll get an introduction to both of these important programming topics, and, along the way, we'll discover some new things about macros.

The Program

In the CHAP9 folder on your *Assembly Language Workshop* disk, you'll find the files PROG7.S and PROG7.TOS, which are the source code and executable program files for this chapter's sample program. (A listing of the program can be found at the end of this chapter.) If you'd like to assemble the program yourself, please refer to the instructions that came with your assembler or check this book's appendix A.

When you run the program, you are asked to enter a filename. This should be the name of a text file you want to send to the printer. If the file is in the same folder from which you ran the program, you need type only the filename. If the file is in another folder, you must type the entire pathname--for example, D:\DOCUMENT\MY-FILE.TXT. Your pathname may be up to 64 characters in length, so you should be able to get to any file on your disks.

After you enter the filename, the program checks to see that the file exists. If it doesn't, the program displays a file-open error and asks if you want to try again. If the program finds the file, it then checks that the printer is responding correctly. If the printer doesn't respond, the program displays a printer error. If you want to try again, check your printer, and then respond "Y" to the prompt. If all goes well, the complete document is sent to your printer, after which you'll see the message "All done." Press Return to return to the desktop.

More About Macros

At the top of the program listing, you'll find three macros, the first being a revision of our print-string macro from a previous chapter. (MadMac users should consult Appendix A for their versions of these macros.) In this version, we've added code to check that the right number of parameters has been sent to the macro. We do this using the *ifc* assembler directive, which compares two strings, both of which should be enclosed in single quotes. If the strings are equal, the code between the *ifc* and the *endc* (the *endc* marks the end of the "if" construct) is assembled. If the strings are not equal, assembly continues after the *endc*.

In our macro, we compare an empty string with the macro's first parameter, like this:

```
ifc '','\1'
```

If, when we called the macro, we forgot to send the required parameter, the \1 would be an empty string, in which case, we would execute the following assembler directives:

```
fail Missing parameter!
mexit
```

The first line sends an error to the user. (Here, the user is you, the programmer; this is an assembly-time error, not a run-time error.) The second line forces the assembler to exit from the macro, without processing further instructions.

If the parameter sent to the macro is not an empty string, the assembler starts processing the code that follows the *endc*. This is the same code that our macro originally contained.

When we use if-type directives in our assembly language programs, we are using "conditional assembly." That is, assembly of particular pieces of code is performed only under certain conditions. Conditional assembly is a powerful tool that allows us to make our programs more versatile and more easily changeable.

A String-Input Macro

The second macro, *input_s*, simplifies our getting a string from the keyboard. We call the macro with the line

```
input_s string,x
```

where *string* is the address of the buffer in which the string is to be stored and *x* is the maximum allowable length of the string, which should be no larger than the length of the buffer minus 2. Remember that *Cconrs*, the GEMDOS function we call in the macro, expects to find the maximum allowable string length in the first byte of the buffer, and it will place the length of the string that was entered in byte 2 of the buffer.

Normally, when we call this function, we must, besides load the right parameters onto the stack, make sure the right value has been stored in the first byte of the buffer. Our macro takes care of all this for us, making this function call a snap to use. Notice that, again, we're using the *ifc* directive to check the macro's parameters. In this case, we're comparing an empty string to \2, the second parameter. If we checked \1, all we'd know is whether the first parameter was passed. By checking \2, we can verify that both parameters were sent. Of course, this doesn't guarantee that the parameters are correct--only that they exist.

A Get–Character Macro

Our second new macro, *get_char*, simplifies our getting a character from the keyboard. Rather than having to type three lines of assembly code every time we want a character, we can just type *get_char*, and our new macro will take care of everything for us.

In the *get_char* macro, we don't use the *ifc* directive to check for parameters, because this macro requires no parameters. All we do is call the macro by name, after which register D0 contains the character that the user typed.

Now, let's look at the main program.

Opening a Disk file

The first thing we do in the main program is prompt the user for a filename. We use this filename in a call to the GEMDOS function *Fopen*, which opens a disk file and returns an identifier, called a "handle," that we use whenever we need to reference the file. The *Fopen* call requires that three parameters be placed onto the stack: the operations mode, the address of the null-terminated filename, and the function's number (61). The operations code determines what operations may be performed on the file and must be one of three values: 0 for read only, 2 for write only, and 3 for read or write. After the call (via *trap #1*), register D0 will contain the file handle or a negative value, the latter of which indicates an error. In our program, we call the function like this:

```
move.w #READ_ONLY,-(sp)
move.l #filename+2,-(sp)
move.w #FOPEN,-(sp)
```

```
        trap #1
        addq.l #8,sp
```

Notice the use of constants to increase the readability of the code. It's a heck of a lot easier to understand the program when we use a constant like READ_ONLY rather than the number 0. Moral: Wherever possible, replace values in your code with descriptive constant names.

After we call *Fopen*, we test D0, to make sure we didn't get an error. The branch instruction *bge* (Branch Greater than or Equal) branches if the result of our *tst* instruction is greater than or equal to 0. If D0 contains a negative number, we display an error message and give the user a chance to try again. If we don't get an error (D0 is positive), we branch to the label *read_file*, where we save the file handle that was returned to us in D0. Note that the first file handle is always 6, because TOS reserves handles -3 through 5 for system devices, such as the screen and the printer.

Checking the Printer's Status

The next step, before we can start sending text to the printer, is to check that the printer port is ready to receive data. We do this with a call to the GEMDOS function *Cprnos*, like this:

```
        move.w #CPRNOS,-(sp)
        trap #1
        addq.l #2,sp
```

The function call's single parameter is the function number, 17. After the call, register D0 will contain 0 or -1. The former indicates that the Centronics parallel printer port is not ready to accept output; the latter indicates that the port is ready to receive data.

After the call to *Cprnos*, we test D0. If the register contains 0, we display an error message and give the user a chance to correct the situation. If the printer is ready, we branch to the label *printer_ready*, where we start sending data to the printer.

Reading a File

Of course, before we can send data to the printer, we first must have data. The data we'll be printing is found in the file we just opened. To send the contents of the file to the printer, we must read data from the file into a buffer and then send the contents of the buffer to the printer, all the while making sure that the printer is paying attention.

This is easier than it sounds. First, we read data from the file into our buffer, like this:

```
move.l #buffer,-(sp)
move.l #256,-(sp)
move.w handle,-(sp)
move.w #FREAD,-(sp)
trap #1
add.l #12,sp
```

The above is an example of a call to the GEMDOS function *Fread*, which reads a specific number of bytes from a file into a buffer. As you can see, this function requires that four parameters be placed on the stack: the address of the buffer into which the data will be read, the number of bytes to read, the source file's handle, and the function number (63).

After the call, register D0 will contain the number of bytes actually read (it may not be what we requested), a negative number if we experienced a read error, or a 0 if we tried to read data after reaching the end of the file (eof).

Why would *Fread* sometimes read less than we requested? Usually because, at the time of the call, we have fewer bytes in the file than the number requested. For example, in our program, we're reading the file 256 bytes at a time. Let's say that our file contains 550 bytes. On the first read, we'll copy 256 bytes from the file into our buffer, after which D0 will contain 256. The second read will get us another 256 bytes, and D0 will again contain 256. At this point, we have already read 512 bytes, which means that our file has only 38 bytes remaining. On the third read, we'll transfer those 38 bytes to the buffer, and D0 will contain 38--less than we actually requested, but all that was available. Finally, on our fourth read, D0 will contain 0, because we were at the end of the file and could read no data.

So, getting back to our program, after we read data into our buffer, we test D0. If it contains a 0, we have printed all the data in the file. In that case, we branch to the label *eof*, where we inform the user that we are done printing his file, after which we close the file using the GEMDOS function *Fclose* (#62) and exit to the desktop. If D0 contains a negative number, we experienced some sort of read error, so we print an error message to the user and exit to the desktop.

If D0 contains a value greater than 0, we have new data in our buffer. We branch to the label *print_buffer*, where we...well...print the buffer, of course! To do this, we first save the number of characters in the buffer, by copying D0 into D5. (Remember: D0 is frequently corrupted by system function calls.) We then move the address of our buffer into A3, so we can use that register as a pointer. We are going to need to use two data registers: one to convert each character into the proper format for the print-character function and one to use as a loop counter. Before we start, we clear both registers.

At the label *next_char*, we actually start printing, with a call to the GEMDOS function *Cprnout*, which looks like this in our program:

```
move.b 0(a3,d3),d4
move.w d4,-(sp)
move.w #CPRNOUT,-(sp)
trap #1
addq.l #4,sp
```

Using A3 as a pointer and our loop counter as an index, we use address register indirect with index addressing to move a character from the buffer into D4. We move the character into D4 first, because we can't move bytes onto the stack. We must convert each of our characters into words. After moving the character into D4, we move it again, this time in word form, onto the stack. We follow that with the function number (5) for *Cprnout*. The *trap #1* instruction calls the function, which prints the character and returns, after storing a status code in D0. A negative code means the character was printed; a 0 means the character wasn't printed, probably due to a printer time-out of some kind.

After calling *Cprnout*, we test D0. In the case of an error, we branch to the label *timeout*, where we print an error message to the screen and give the user a chance to correct the problem. If the character printed okay (D0 is negative), we increment our loop counter with the line *addq.l #1,d3*; and compare d3 with d5, the character counter, to see if we've yet printed all the new characters in the buffer. If we have printed the last character in the buffer, we branch to *printer_ready*, where we refill the buffer. If D3 is still smaller than D5, we branch to *next_char*, to print the next character in the buffer.

Gasp! Bet you never realized how hard your computer has to work to do something as ostensibly simple as printing a little text!

Why a Buffer?

In computing, we see the word "buffer" a lot. There isn't a computer system on the planet that doesn't use buffers. If you don't know much about computer architecture, you're probably not sure why we bother with buffers. Why not just read a character from the disk and then send it directly to the printer? If we did that, we could get rid of the buffer completely, not to mention the time it takes to load the buffer from disk.

The fact is that, due to the glacial speed of disk drives (at least glacial in terms of computing speed), it would take an immense amount of time to have to access a disk drive for each byte. In other words, it is much faster to read 256 bytes in one big chunk than it is to read 256 bytes one at a time--*much* faster. But, you ask, how about the time it takes to read the characters from the buffer? And, I say, what about it? Your computer's memory is *fast*. Comparing a disk drive to memory is like comparing a skate board to Star Trek's *Enterprise*.

Now you're probably thinking that, if using a 256-byte buffer is so fast, wouldn't a 1024-byte buffer be even faster? It sure would! This is one of those times when you have to balance memory usage against speed. If we wanted, we could have a buffer big enough to hold an entire text file, but then that memory would be unavailable for other uses. As you know from running the program, a 256-byte buffer seems to work pretty well for our purposes. If you like you can increase it, although you probably wouldn't want to make it any larger than 1,024 bytes (a full kilobyte).

Conclusion

Reading a disk file and sending text to a printer is a heck of a lot easier in assembly language than one might expect. Except for having to fool with a buffer, these tasks are really not much more difficult than they are in a high-level language. In fact, although you're not aware of it, even a high-level language like BASIC has to use a buffer when reading from a disk file or sending to a printer. For example, when you use a language like BASIC to read data from a disk, you read the data into a variable. In this case, the variable is your buffer.

As always, go over the following summary carefully, to be sure you understand the topics covered in this chapter. When you're ready, I'll meet you in chapter 10.

Summary

- ✓ Using if-type constructs to control the assembly process is called conditional assembly.
- ✓ The *ifc* and *endc* assembler directives allow us to compare two strings and assemble different sections of code based on the outcome of the compare.
- ✓ The assembler directive *fail* allows us to generate our own error messages. The text following the directive will be printed to the screen as a user error message.
- ✓ The *mexit* assembler directive allows us to exit a macro prematurely, usually because of a detected error condition.
- ✓ The GEMDOS function *Fopen* opens a disk file for one of three operations. It requires three parameters: the operations code, the address of a null-terminated filename, and the function number (61). The three operations codes are 0, 2, and 3, which stand for read only, write only, and read and write, respectively. After the call, D0 will contain a file handle or a negative error code.

✓ The *bge* instruction causes a branch when the result of a previous *cmp* or *tst* is greater than or equal to 0.

✓ The GEMDOS function *Cprnos* allows us to check the status of a printer. It requires one argument, the function number (17). After the call, a negative value in D0 indicates that the printer is ready to receive data. A 0 in D0 indicates that the printer did not respond.

✓ The GEMDOS function *Fread* reads a specified number of bytes from an open disk file and places them into a buffer. The function requires four arguments: the address of the buffer, the number of bytes to read, the handle of the file, and the function number (63). After the call, D0 will contain the number of bytes read, a zero if the read was at the end of the file, or a negative error code.

✓ The GEMDOS function *Cprnout* sends a single character to a printer. It requires two arguments: the character to be printed (in word form) and the function number (5). After the call, D0 will contain a negative value if the character was printed and a 0 if there was an error.

✓ The GEMDOS function *Fclose* closes an open file. It requires two arguments: the file's handle and the function number (62). After the call, D0 will contain a 0 if all went well or a GEMDOS error number if it didn't.

✓ Buffers are used to speed up the transfer of data in a computer system. Generally, the larger the buffer, the faster data can be moved between peripherals (i.e., disks and printers). A 1K buffer is usually more than sufficient for most uses.

```
;------------------------------------------------------------
; THE ST ASSEMBLY LANGUAGE WORKSHOP
; PROGRAM 7
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;------------------------------------------------------------

;------------------------------------------------------------
; This macro prints a string to the screen, using the Cconws GEMDOS
; function. It requires one parameter, the address of the string
; to print.
;------------------------------------------------------------
print   macro
        ifc     '','\1'
        fail    Missing parameter!
        mexit
        endc
        move.l  #\1,-(sp)
        move.w  #CCONWS,-(sp)
        trap    #1
        addq.l  #6,sp
        endm

;------------------------------------------------------------
; This macro gets a string from the keyboard using the Cconrs GEMDOS
; function. It requires two parameters: the address of the buffer
; in which to store the string and the max number of characters
; to read.
;-----------------------------------------------------------------
input_s macro
        ifc     '','\2'
        fail    Missing parameters!
        mexit
        endc
        move.b  #\2,\1
        move.l  #\1,-(sp)
        move.w  #CCONRS,-(sp)
        trap    #1
        addq.l  #6,sp
        endm

;-----------------------------------------------------------------
; This macro gets a single character from the keyboard, using the
; Cconin GEMDOS function.
;-----------------------------------------------------------------
get_char        macro
        move.w  #CCONIN,-(sp)
        trap    #1
        addq.l  #2,sp
        endm
```

```
;-----------------------------------------------------------------
; MAIN PROGRAM
;-----------------------------------------------------------------
        text
get_filename:
        print    msg1                    ; "Enter name of file..."
        input_s  filename,64             ; get filename
        print    crlf
        move.l   #filename,a3            ; add a null to...
        jsr      add_null                ; the filename.

        move.w   #READ_ONLY,-(sp)        ; open the requested...
        move.l   #filename+2,-(sp)       ; file for read-only...
        move.w   #FOPEN,-(sp)            ; access.
        trap     #1
        addq.l   #8,sp

        tst.l    d0                      ; error opening file?
        bge      read_file               ; nope, go read file.
                                         ; yep, we got an error.
        print    msg2                    ; "File open error!"
        print    msg3                    ; "Exit (y/n)?"
        get_char                         ; get user's answer.
        move.l   d0,d3                   ; save character typed.
        print    crlf                       ; next screen line.
        cmp.w    #UC_Y,d3                   ; user typed "Y"?
        beq      out                        ; yep, we're outta here.
        cmp.w    #LC_Y,d3                   ; user typed "y"?
        beq      out                        ; sure did.
        bra      get_filename               ; try for filename again.

read_file:
        move.w   d0,handle                  ; save file handle.

check_printer:
        move.w   #CPRNOS,-(sp)              ; check printer's status.
        trap     #1
        addq.l   #2,sp

        tst      d0                         ; is printer ready?
        bmi      printer_ready              ; yep, go print.
                                            ; no, printer didn't respond.
        print    msg5                       ; "Printer not responding!"
        print    msg6                       ; "Want to try again?"
        get_char                            ; get user's answer.
        move.l   d0,d3                      ; save answer.
        print    crlf                       ; got to next screen line.
        cmp.w    #UC_Y,d3                   ; check character typed...
        beq      check_printer              ; and go back for...
        cmp.w   #LC_Y,d3                ; another try if user...
        beq     check_printer           ; typed "Y" or "y".
        bra     out                     ; else, exit program.

printer_ready:
        move.l  #buffer,-(sp)           ; get address of buffer.
        move.l  #256,-(sp)              ; # of bytes to read.
        move.w  handle,-(sp)            ; file handle.
        move.w  #FREAD,-(sp)            ; function #.
        trap    #1                      ; read from file.
        add.l   #12,sp                  ; fix stack.
        tst.l   d0                      ; did we read anything?
        bgt     print_buffer            ; yep, go print it.
        beq     eof                     ; no, reached eof, or...
                                        ; got an error.
        print   msg4                    ; "File read error!"
        get_char                        ; wait for user to...
        bra     out                     ; press key, then exit.

print_buffer:
        move.l  d0,d5                   ; save read count.
        move.l  #buffer,a3              ; get address of buffer.
        clr.l   d3                      ; init char storage.
        clr.w   d4                      ; init loop counter.
```

```
next_char:
        move.b  Ø(a3,d3),d4             ; move char to register...
        move.w  d4,-(sp)                ; then to stack as word.
        move.w  #CPRNOUT,-(sp)          ; function #.
        trap    #1                      ; send char to printer.
        addq.l  #4,sp                   ; correct stack.

        tst     dØ                      ; was there an error?
        beq     timeout                 ; yep.
        addq.l  #1,d3                   ; no. increment count.
        cmp.l   d3,d5                   ; last character?
        beq     printer_ready           ; yes. go refill buffer.
        bra     next_char               ; no. go print next char.

timeout:                                ; handle printer time out.
        print   msg7                    ; "Printer timed out."
        print   msg6                    ; "Want to try again?"
        get_char                        ; give the user...
        move.l  dØ,d3                   ; a chance to fix...
        print   crlf                    ; the problem...
        cmp.w   #UC_Y,d3                ; and try again.
        beq     next_char
        cmp.w   #LC_Y,d3
        beq     next_char               ; go try again to print.
        bra     out                     ; or exit the program.
```

```
eof:
        print    msg8                        ; "All done."
        get_char                             ; wait for keypress.

out:
        move.w  #PTERMØ,-(sp)                ; adios.
        trap    #1

;-------------------------------------------------------------------
; This subroutine adds a null to the string retrieved with a call
; to the GEMDOS function Cconrs.
;
; Input: Address of string buffer in a3.
; Output: Null after last character in string.
; Registers changed: NONE.
;-------------------------------------------------------------------
add_null:
        movem.l a3/d3,-(sp)
        clr.l   d3
        move.b  1(a3),d3
        move.b  #Ø,2(a3,d3)
        movem.l (sp)+,a3/d3
        rts

;-------------------------------------------------------------------
```

```
          data

READ_ONLY           equ     Ø
UC_Y                equ     89
LC_Y                equ     121

PTERMØ              equ     Ø
CCONIN              equ     1
CPRNOUT             equ     5
CCONWS              equ     9
CCONRS              equ     10
CPRNOS              equ     17
FOPEN               equ     61
FCLOSE              equ     62
FREAD               equ     63

msg1:     dc.b      "Enter name of file to print: ",Ø
msg2:     dc.b      "File open error!",13,1Ø,Ø
msg3:     dc.b      "Exit (y/n)?",Ø
msg4:     dc.b      "File read error!",13,1Ø,Ø
msg5:     dc.b      "Printer not responding!",13,1Ø,Ø
msg6:     dc.b      "Want to try again (y/n)? ",Ø
msg7:     dc.b      "Printer timed out.",13,13Ø,Ø
msg8:     dc.b      "All Done.",13,1Ø,Ø
crlf:     dc.b      13,1Ø,Ø
          bss
```

```
        even
handle:     ds.w    1
filename:   ds.b    67
buffer:     ds.b    256
```

10 More File Handling

In the previous chapter, we learned how to open and read files. Many programs, however, also require that we write data to a file. Moreover, in many cases, the file to which we want to write the data doesn't yet exist, which means we need a way to create new files, as well as write to them. In this chapter, we'll study these other important file-handling basics.

The Program

In the CHAP10 folder on your *Assembly Language Workshop* disk, you'll find the files PROG8.S and PROG8.TOS, which are the source code and executable program files for this chapter's sample program. (A listing of the program is included at the end of this chapter.) If you'd like to assemble the program yourself, please refer to the instructions that came with your assembler or check this book's appendix A.

PROG8.TOS does nothing more than copy the contents of a source file into a new destination file. When you run the program, you're asked for the source filename. Enter the filename and press Return. You're then asked for the destination file. If the file you enter for the destination already exists on the disk, you are asked whether you want the old file deleted. If you respond "Y," the old file is overwritten with the new one. If you respond "N," you're asked for a new filename. Once the file has been copied, press Return to get back to the desktop.

Some New Macros

In order to keep the main program as understandable and as short as possible, PROG8.TOS uses some new macros. These macros--*open_file*, *create_file*, *read_file*, *write_file*, and *close_file*--supply us with all the tools we need for general file handling. We looked at a couple of these functions previously. Specifically, the *open_file*, *read_file*, and *close_file* macros use the same GEMDOS functions we studied in chapter 9, so we won't cover them here. You should, however, take a little time to compare the macro versions with the functions as we used them previously.

One thing you'll notice immediately is a new instruction, *pea*. The instruction *pea* (Push Effective Address) pushes the effective address of an operand onto the stack. For example, the lines *move.l #buffer,-(sp)* and *pea buffer* accomplish the same thing, placing the address of *buffer* onto the stack, but in the latter case, we don't have to manipulate the stack pointer ourselves; the system does it for us.

Creating a File

So much for opening, reading, and closing a file. Now, let's look at the new GEMDOS file-handling functions, which are used in our new macros. In case you couldn't figure it out by its name, the *create_file* macro creates a new file. It's invoked like this:

```
create_file attribute,filename,handle
```

In this macro call, *attribute* determines the type of file to be created. A value of 0 creates a normal file, a 1 creates a read-only file (one that can't be deleted or written to), a 2 creates a hidden file (one that doesn't show up in the normal directory), a 4 creates a system file (also one that doesn't appear in a normal directory), and an 8 creates a volume label. The argument *filename* is the address of a null-terminated filename string. Finally, *handle* is the address where the file handle returned from the function is to be stored. After creating the file, we will use the handle to refer to the file. The filename is used only when opening or creating the file.

If you look at the *create_file* macro, you can see how we call the GEMDOS function *Fcreate* (#60). As always, we first place the function's arguments on the stack, in this case the file's attribute flag, the address of the filename, and the function's number. Then we call the function with a *trap #1*. The file handle is returned by the function in D0, from which we copy it into the address represented by the third macro parameter before ending the macro. If the function returns a negative value, an error occurred.

Writing to a File

Another new macro, *write_file*, writes data to an open file. It is invoked like this:

```
write_file buffer,num,handle
```

In this macro call, *buffer* is the address of the source data (the data to be written to the file), *num* is the number of bytes to write to the file, and *handle* is, of course, the file's handle, which was returned to us by *Fcreate* or *Fopen*. (Note that *num* must be a long word!) This macro uses the GEMDOS function *Fwrite* (#64), which is called by placing the buffer address, the number of bytes to write, the file handle, and the function number on the stack, and then using *trap #1* to call the operating system. After a call to *Fwrite*, the actual number of bytes written to the file is returned in D0. Hopefully, this will be same the number that we requested be written. If an error occurs, the function returns a negative error code.

High–level Assembly Language

Look at the program listing. Notice that, although the complete program is fairly lengthy, the main program section is quite short. Moreover, the main program section reads more like a high-level language than assembly language. This is because we've placed all the details of the program into macros or subroutines.

PROG8.TOS has been written using top-down programming techniques. This style of programming provides both advantages and disadvantages. The main advantage is that, using many subroutines and macros, our main program section is, in a general way, easy to understand. On the other hand, if we want to see exactly what the program is doing, we must continually jump from the code in the main program to the code in the subroutines and macros. Moreover, we need to add many extra comments to our program, to document the subroutines and macros. This makes our program listings longer, although the size of the assembled program should stay about the same.

Deciding whether to hide all your program's details in subroutines and macros or write in a more straight-line fashion is a matter of taste. You should, however, always use subroutines for longer tasks that you perform often in different parts of your program. Doing this will reduce the size of the assembled program. Macros, on the other hand, because they are always expanded to their full size wherever they are invoked, don't shorten your program's executable file at all. Instead they shorten the size of the source code. This difference between macros and subroutines will help you decide which to use when.

The astute among you may notice that some macros used in this book could be written as subroutines and thus save us a couple of instructions in the assembled object code. However, because a macro invocation is much easier to read than the two or more instructions required to call a subroutine (you usually must load values into registers, as well as use the *bsr* instruction), I usually choose a macro over a subroutine wherever it seems reasonable. Exceptions to this are when a subroutine can be called with only one instruction (i.e., *bsr get_filename1*) or when the number of instructions to perform a task is too large to make using a macro reasonable. Remember: the code that makes up a macro replaces the macro invocation wherever the invocation appears, whereas a subroutine's code appears only once in the assembled program. If you invoke a large macro many times in a program, the program could grow to monstrous proportions.

Inside the Program

Let's take a quick look at the main program section, starting at the label *get_source*. First, we call the subroutine *get_filename1*, which gets the source filename from the user. This subroutine uses the macro *input_s* to get the filename (with *Cconrs*). It also calls the subroutine *add_null*, which adds a null character to the end of the string. (Yes, you can use macro invocations and subroutine calls inside subroutines.) Finally, the subroutine returns the address of this string in the variable *Filename2*.

After we have the source filename, we then invoke the *open_file* macro, passing the READ_ONLY file flag, the address of the filename string, and the address of the variable where we'll store the file's handle. Notice that, for the filename's address,we're passing *filename1+2*. Remember that *Cconrs* uses the first two bytes of the buffer to store and retrieve information about the string.

After the call to *open_file*, a file handle or a negative error code (which *Fopen* placed in D0) will be stored in the variable *handle*. We use the *tst* instruction to make sure the file opened correctly. If we have a valid file handle (a positive number), we branch to the label *get_dest*. If we discover that an error occurred, we give the user a chance to try again or to quit the program.

Assuming the file opened successfully, we get the destination filename by calling the subroutine *get_filename2*, which works similarly to *get_filename1*, except it returns the address of the filename in the variable *filename2*. We use *filename2* in another call to the macro *open_file*. Unlike our first call to *open_file*, this time we want to get an error code, since that indicates the file doesn't already exist. (You can't open a non-existent file.) If we get an error code, we can go ahead and branch to the label *make_file*, where we'll create the new file.

If our call to *open_file* yields a valid file handle, we must warn the user that the destination file already exists and ask whether he wants it overwritten. Because the *get_answer* subroutine returns a 0 if the user answers "N," after the call to the subroutine, we use the instruction *beq get_dest* to give the user a chance to type in a different filename.

At *make_file*, we use our *create_file* macro to create the new file. Like opening a file, when we create a file, a negative file handle indicates an error. After invoking the macro, we test for this condition with the instruction *tst handle2*, after which, if we have a valid file handle, the instruction *bge copy* will send us to the section of the code that copies the file. If we encounter an error when creating the file, we inform the user and end the program.

Our final task, after getting both files open, is to copy the source file to the destination file. To do this, we call our *read_file* macro to read 1K of data into our buffer. If, after the macro invocation, D0 contains a positive, non-zero value, we branch to the label *write_buffer*, where we invoke our *write_file* macro to write the data from the buffer into our destination file. If D0 contains 0 after invoking *read_file*, we've finished copying the file, and so we branch to the label *eof*, where we inform the user that we're done and exit the program. If D0 contains a negative value, we've encountered a read error. If this happens, we inform the user and exit the program.

Note that, although we don't take advantage of it in this program, the *Fwrite* function, which we use in our *write_file* macro, also returns a negative error code in D0 if there's a problem writing the data to the file. Otherwise, this function returns the number of bytes written. Note also that, before we exit the program, we use our *close_file* macro to close both the open files.

Conclusion

As you can see from this chapter's program, subroutines and macros help take the details of your program and hide them out of the way. Our main program section here contains only about 40 instructions.

Also, this chapter's program should have given you a firm grasp on file handling with assembly language. Because you will often use these file-handling functions in your programs, make sure you understand how they work. Besides the file-handling functions discussed in the last two chapters, there is also a complete set of functions that allow you to do such things as delete files, create directories (folders), get the current drive name or path, and set the current path. We'll look at these functions in volume II of *The ST Assembly Language Workshop*.

Summary

✓ The instruction *pea* (Push Effective Address) pushes the effective address of an operand onto the stack.

✓ The GEMDOS function *Fcreate* creates a new file. It requires three arguments: an attribute flag, the address of a null-terminated filename, and the function number (60). For the attribute flag, a value of 0 creates a normal file, a 1 creates a read-only file, a 2 creates a hidden file, a 4 creates a system file, and an 8 creates a volume label. After the call, D0 will contain the file's handle or a negative error code. See table 10.1 for a list of GEMDOS error codes.

✓ The GEMDOS function *Fwrite* writes data to an open file. It requires four arguments: the address of the buffer holding the data, the number of bytes to write, the file handle, and the function's number (64). After the call, D0 will contain the number of bytes written or a negative error code.

✓ By using many macros and subroutines, we can make our programs generally much easier to understand.

Error #	Description
-32	Invalid function number
-33	File not found
-34	Path not found
-35	Too many open files
-36	Access denied
-37	Invalid file descriptor
-39	Insufficient memory
-40	Invalid memory block address
-46	Invalid drive specified
-49	No more files
-64	Range error
-65	Internal error
-66	Invalid program load format
-67	Setblock failure do to growth restrictions

Table 10.1 • GEMDOS Error Codes

```
;------------------------------------------------------------------
; THE ST ASSEMBLY LANGUAGE WORKSHOP
; PROGRAM 8
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;------------------------------------------------------------------

;------------------------------------------------------------------
; This macro opens a file using the unbuffered, Fopen GEMDOS
; function. It requires three parameters: the file mode, the address
; of the filename, and the address for file-handle storage.
;------------------------------------------------------------------
open_file macro
        ifc     '','\3'
        fail    Missing parameters!
        mexit
        endc
        move.w  #\1,-(sp)
        pea     \2
        move.w  #FOPEN,-(sp)
        trap    #1
        addq.l  #8,sp
        move.w  d0,\3
        endm
```

```
;------------------------------------------------------------------
; This macro creates a file using the unbuffered, Fcreate GEMDOS
; function. It requires three parameters: the file attribute, the
; address of the filename, and the address for file-handle storage.
;------------------------------------------------------------------
create_file macro
        ifc     '','\3'
        fail    Missing parameters!
        mexit
        endc
        move.w  #\1,-(sp)
        pea     \2
        move.w  #FCREATE,-(sp)
        trap    #1
        addq.l  #8,sp
        move.w  d0,\3
        endm

;------------------------------------------------------------------
; This macro reads a file using the unbuffered, Fread GEMDOS
; function. It requires three parameters: the buffer address,
; the number of bytes to read, and the file handle.
;------------------------------------------------------------------
read_file macro
        ifc     '','\3'
        fail    Missing parameters!
        mexit
        endc
        pea     \1
        move.l  #\2,-(sp)
        move.w  \3,-(sp)
        move.w  #FREAD,-(sp)
        trap    #1
        add.l   #12,sp
        endm
;-----------------------------------------------------------------------
; This macro writes to a file using the unbuffered, Fwrite GEMDOS
; function. It requires three parameters: the buffer address,
; the address of the number of bytes to write, and the file handle.
;-----------------------------------------------------------------------
write_file macro
        ifc     '','\3'
        fail    Missing parameters!
        mexit
        endc
        pea     \1
        move.l  \2,-(sp)
        move.w  \3,-(sp)
        move.w  #FWRITE,-(sp)
        trap    #1
        add.l   #12,sp
        endm

;-------------------------------------------------------------------
; This macro closes a file using the unbuffered, Fclose GEMDOS
; function. It requires one parameter: the file handle.
;-------------------------------------------------------------------
close_file macro
        ifc     '','\1'
        fail    Missing parameter!
        mexit
        endc
        move.w  \1,-(sp)
        move.w  #FCLOSE,-(sp)
        trap    #1
        addq.l  #4,sp
        endm

;-------------------------------------------------------------------
; This macro prints a string to the screen, using the Cconws GEMDOS
; function. It requires one parameter, the address of the string
; to print.
;-------------------------------------------------------------------
print   macro
        ifc     '','\1'
        fail    Missing parameter!
        mexit
        endc
        move.l  #\1,-(sp)
        move.w  #CCONWS,-(sp)
        trap    #1
        addq.l  #6,sp
        endm
;-----------------------------------------------------------------
; This macro gets a string from the keyboard using the Cconrs GEMDOS
; function. It requires two parameters: the address of the buffer
; in which to store the string and the max number of characters
; to read.
;-----------------------------------------------------------------
input_s macro
        ifc     '','\2'
        fail    Missing parameters!
        mexit
        endc
        move.b  #\2,\1
        move.l  #\1,-(sp)
        move.w  #CCONRS,-(sp)
        trap    #1
        addq.l  #6,sp
        endm
```

```
;------------------------------------------------------------
; This macro gets a single character from the keyboard, using the
; Cconin GEMDOS function.
;------------------------------------------------------------
get_char        macro
        move.w  #CCONIN,-(sp)
        trap    #1
        addq.l  #2,sp
        endm

;------------------------------------------------------------
; MAIN PROGRAM
;------------------------------------------------------------
        text
get_source:
        jsr     get_filename1                   ; get source filename.
        open_file READ_ONLY,filename1+2,handle1 ; open source file.

        tst.l   handle1                         ; error opening file?
        bge     get_dest                        ; nope, go read file.

        print   msg3                            ; "File open error!"
        print   msg4                            ; "Exit (y/n)?"
        jsr     get_answer
        bne     out                             ; User typed "Y".
        bra     get_source                        ; try for filename again.

get_dest:
        jsr     get_filename2                     ; get destination file.
        open_file READ_ONLY,filename2+2,handle2  ; open destination file.
        tst     handle2                           ; does file already exist?
        bmi     make_file                         ; nope, okay to create it.
        print   msg6                              ; "File already exists!"
        print   msg7                              ; "Delete it?"
        jsr     get_answer
        beq     get_dest                          ; try for filename again.
make_file:
        create_file 0,filename2+2,handle2         ; create new file.
        tst     handle2                           ; Error?
        bge     copy                              ; Nope, go copy file.
        print   msg8                              ; "Error creating file."
        print   msg9                              ; "Program aborting."
        get_char                                  ; wait for keypress.
        close_file handle1                        ; close file.
        bra     out                               ; skidaddle.

copy:
        read_file buffer,1024,handle1             ; read source file.
        tst.l   d0                                ; did we read anything?
        bgt     write_buffer                      ; yep, go write it out.
        beq     eof                             ; nothing read, so eof.

        print   msg5                            ; "File read error!"
        print   msg9                            ; "Program aborting."
        get_char                                ; wait for keypress.
        bra     out                             ; abort.

write_buffer:
        write_file buffer,d0,handle2            ; write buffer to dest. file.
        bra     copy                            ; read next bufferful.

eof:
        print   msg10                           ; "All done."
        get_char                                ; wait for keypress.

out:
        close_file handle1                      ; close source file.
        close_file handle2                      ; close destination file.
        move.w  #PTERM0,-(sp)                   ; adios.
        trap    #1

;---------------------------------------------------------------
; This subroutine adds a null to the string retrieved with a call
; to the GEMDOS function Cconrs.
;
```

```
; Input: Address of string buffer in a3.
; Output: Null after last character in string.
; Registers changed: NONE.
;-----------------------------------------------------------------
add_null:
        movem.l a3/d3,-(sp)
        clr.l   d3
        move.b  1(a3),d3
        move.b  #0,2(a3,d3)
        movem.l (sp)+,a3/d3
        rts

;-----------------------------------------------------------------
; This subroutine retrieves the name of the source file.
;
; Input:        NONE.
; Output:       Null-terminated filename in filename1.
; Regs changed: NONE.
; Calls:        print, input_s, add_null
; Data used:    msg1, filename1, crlf
;-----------------------------------------------------------------
get_filename1:
        print   msg1                          ; "Enter name of file..."
        input_s filename1,64                  ; get filename
        print   crlf
        move.l  #filename1,a3                   ; add a null to...
        jsr     add_null                        ; the filename.
        rts
;-----------------------------------------------------------------
; This subroutine retrieves the name of the destination file.
;
; Input:        NONE.
; Output:       Null-terminated filename in filename2.
; Regs changed: NONE.
; Calls:        print, input_s, add_null
; Data used:    msg1, filename1, crlf
;-----------------------------------------------------------------
get_filename2:
        print   msg2                            ; "Enter new filename:"
        input_s filename2,64
        print   crlf
        move.l  #filename2,a3
        jsr     add_null
        rts

;-----------------------------------------------------------------
; This subroutine retrieves a one-character response from the
; keyboard and compares it against Y and N.
;
; Input:        NONE.
; Output:        1->D0 if Y typed. 0->D0 if any other key typed.
; Regs changed: D0.
; Calls:         get_char
; Data used:     UC_Y, LC_Y
;-----------------------------------------------------------------------
get_answer:
         get_char                                  ; get user's answer.
         move.l   d0,d3
         print    crlf
         cmp.w    #UC_Y,d3                         ; user typed "Y"?
         beq      .yes                             ; yep.
         cmp.w    #LC_Y,d3                         ; user typed "y"?
         beq      .yes                             ; yep.
         clr.l    D0
         bra      .out
.yes:    moveq.l  #1,D0
.out:
         rts
end_get_answer:


;-----------------------------------------------------------------------

         data
```

```
READ_ONLY       equ     0
UC_Y            equ     89
LC_Y            equ     121

PTERM0          equ     0
CCONIN          equ     1
CPRNOUT         equ     5
CCONWS          equ     9
CCONRS          equ     10
CPRNOS          equ     17
FCREATE         equ     60
FOPEN           equ     61
FCLOSE          equ     62
FREAD           equ     63
FWRITE          equ     64

msg1:   dc.b    "Enter name of file to copy: ",0
msg2:   dc.b    "Enter new filename: ",0
msg3:   dc.b    "File open error!",13,10,0
msg4:   dc.b    "Exit (y/n)?",0
msg5:   dc.b    "File read error!",13,10,0
msg6:   dc.b    "File already exists!",13,10,0
msg7:   dc.b    "Delete it (y/n)? ",0
msg8:   dc.b    "Error creating file.",13,10,0
msg9:   dc.b    "Program aborting.",13,10,0
```

```
msg10:  dc.b    "All Done.",13,10,0
crlf:   dc.b    13,10,0

        bss

        even
handle1:        ds.w    1
handle2:        ds.w    1
filename1:      ds.b    67
filename2:      ds.b    67
buffer:         ds.b    1024
```

11
PLAYING WITH THE SCREEN

We've been working hard throughout this workshop. I figure it's about time we had a little fun, especially considering that this is the last program in the book. In this chapter, we won't learn much new about 68000 assembly language, but we will tackle some new--and important--XBIOS functions. We'll use these functions to load and display a DEGAS-format picture file.

The Program

In the CHAP11 folder on your *Assembly Language Workshop* disk, you'll find the files PROG9.S and PROG9.TOS, which are the source code and executable program files for this chapter's sample program. (A listing of the program is included at the end of the chapter.) If you'd like to assemble the program yourself, please refer to the instructions that came with your assembler or check this book's appendix A.

When you run the program, you're asked to type in the name of a DEGAS-format picture file. The file should be in uncompressed format, but may be from either the original *DEGAS* or from *DEGAS Elite*. (For your convenience, the CHAP11 folder contains three DEGAS pictures, one in each resolution.) After you type the filename, the program will check the disk for the file. If the file exists, the program will then ensure that the picture's resolution is the same as your current system resolution. If it is, the program loads and displays the picture. To return to the desktop, press any key after the picture is displayed.

DEGAS File Format

Uncompressed DEGAS files come in two types (see figure 11.1), those created by the original *DEGAS* and those created by *DEGAS Elite*. A file from the original *DEGAS* contains 16,017 words (32,034 bytes). The first word is the picture's resolution, and will be 0 for low resolution, 1 for medium resolution, and 2 for high resolution. The next 16 words make up the picture's palette. It is these values that control the colors in which the picture is displayed. Finally, the last 16,000 words are the actual picture data.

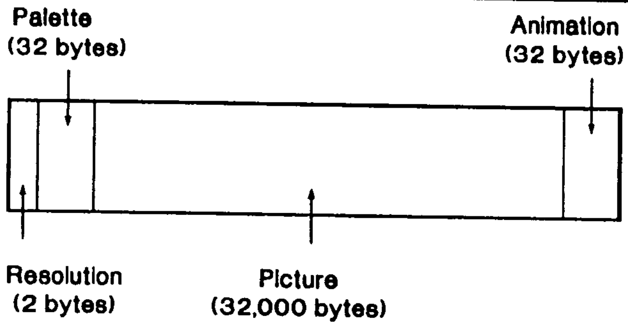


Figure 11.1 • DEGAS Picture File Format

An uncompressed file from *DEGAS Elite* is a bit larger, 16,033 words (32,066 bytes). The first 16,017 words contain the same information as a file from the original *DEGAS*: 1 word for the resolution, 16 words for the palette, and 16,000 words for the picture data. The extra 16 words tacked on to the end of a *DEGAS Elite* uncompressed file are information for controlling animation (the rotation of colors), which is a feature that was added to *DEGAS* when it became *DEGAS Elite*. So, to load and display a picture from either *DEGAS* or *DEGAS Elite*, we need be concerned with only the first 32,034 bytes of the file. Any additional bytes can be ignored, as long as we're not trying to reproduce the animation as well as the picture.

How the Program Works

Let's go through the program's main-program section and see exactly what's required to display DEGAS picture files. Once you learn these basics, you should be able to extend the program to accommodate many different types of picture files. You only need to know the file format.

The first part of the program should be familiar territory. At the label *get_source*, we get a filename from the user, using a subroutine we've seen before, *get_filename*. (It was previously named *get_filename1*.) We then use the filename to open the file. If the file opens okay, we branch to the label *read_rez*, where we start to read the file. If the file didn't open properly, we notify the user of the error and give him another chance. Old hat, right?

At the label *read_rez*, we get into the real meat of the program. Here, we read the first two bytes of the picture file, which gives us the picture's resolution. We store this information in the variable *pic_rez*. We'll use this information later, when we compare the picture's resolution to the system's resolution. After the read, we check to make sure we actually read two bytes. If we did, we branch to *get_rez*. If we didn't (probably because the file is too short), we notify the user of the error.

Getting the System's Resolution

Now, to be sure we can display the picture correctly, we must compare the system's current resolution with the picture's resolution. How can we determine the system's resolution? Easy! All we need is a call to XBIOS function #4, *Getrez*. After a call to *Getrez*, register D0 will contain the system's current resolution, with 0 meaning low resolution, 1 meaning medium resolution, and 2 meaning high resolution. We call *Getrez* like this:

```
move.w  #GETREZ,-(sp)
trap    #14
addq.l  #2,sp
```

Unless you've been dozing, you've probably noticed that the resolution values used in the first word of a DEGAS-format picture file are the same as those returned from *Getrez*. How convenient! To be sure we can display the picture properly, we need only compare our variable *pic_rez* with the value returned in D0, like this:

```
cmp.w  pic_rez,d0
```

If *pic_rez* and D0 are equal, we can go ahead and display the picture. If they're different, we're in the wrong resolution for the currently selected picture, so we must inform the user of his error and give him a chance to pick another file.

Assuming the system and picture resolutions match, we branch to *read_palette*, where we read the 16 words (32 bytes) of color data into the buffer *new_palette*. We'll use this information later in the program, when we learn to set the system's palette.

Calculating Screen Memory

Now that we've read the picture's resolution and palette, we're ready to read the actual picture data. But, where can we store this data so that it can be displayed on the screen?

Your ST can use just about any part of its memory for the screen display,but there's one important rule: An ST's screen memory must always start on a 256-byte boundary. In other words, the screen's starting address must be evenly divisible by 256.

Look at the data portion of our program. In the *bss* section, we have a buffer called, appropriately enough, *buffer*. This buffer is 32,256 bytes long, a little larger than we need to load the DEGAS picture data, which is 32,000 bytes. (By the way, that the picture data is 32,000 bytes long is no arbitrary decision. Your ST's full screen display also uses 32,000 bytes, despite its resolution.)

Why did we make our buffer 256 bytes larger than needed? The answer to that question is a little complicated. When we define our buffer, we have no idea where in memory the system will decide to place it. Yet, we have to be sure that our picture data (and thus the screen display) starts on a 256-byte boundary. This means that we're going to have to load our picture data into the first address after *buffer* that is evenly divisible by 256. In other words, we'll be bumping the screen address up a little higher than the starting address of *buffer*. By making *buffer* 256 bytes larger than we actually need, we are sure to have enough room to load the picture data after we bump up the address.

The calculations required to position our picture and screen data on a 256-byte boundary are found at the label *set_boundary*. We do it like this:

```
move.l  #buffer,d0
andi.l  #$ffffff00,d0
addi.l  #256,d0
move.l  d0,a3
```

In the first line, we copy the address of our buffer into D0, where we can work with it. In the second line, we use an AND operation to mask out the low byte. This operation makes the contents of D0 evenly divisible by 256. However, what we've really done by clearing the lower byte is make the address in D0 lower. In other words, after the AND, the address in D0 is lower than the address of *buffer*. If we try to load the picture at this address, we'll overwrite our program in memory. We can't have that! So, in the third line we add 256 to D0, which moves the address past *buffer* but still on a

256-byte boundary. See why we needed 32,256 bytes rather than just 32,000?

Now that we have a valid screen address, we copy it into register A3, and, using address register indirect addressing, we call our *read_file* macro to read the 32,000 bytes of picture data into the buffer.

Getting the Physical Screen's Address

Your ST actually has two screens: a physical screen and a logical screen. The physical screen is what is displayed on your monitor; it's the part of memory you can see. However, there's also a logical screen, which is the section of memory to which all screen-writing functions are directed. Most of the time, the physical screen and the logical screen are set to the same address, allowing you to see the screen as well as any changes that are being made to it. For example, when you use a word processor, the words you type appear on the screen because the screen you're viewing and the screen you're writing to--that is, the physical and logical screens--are at the same address.

As a programmer, you can separate the physical and logical screens, so that you're looking at one display while you're modifying another--behind the user's back, as it were. If, in our word processor example, we were to separate the physical and logical screens, you'd be able to see your word processor on the screen, but when you typed, no words would appear. Your typing would still be going into the computer's memory, but it would be going to the logical screen instead of the physical screen.

Our DEGAS picture viewer doesn't require that we separate the physical and logical screens. However, because using the function that changes the screen's location requires an understanding of this concept, I thought you'd appreciate being clued in. Just one of the many services we provide at the ST Assembly Language Workshop.

At the label *show_pic*, we start the process of displaying our newly loaded picture. We first save the address of the current physical screen, so we can restore it when we're done. We do this with a call to the XBIOS function #2, *Physbase*, like this:

```
move.w  #PHYSBASE,-(sp)
trap    #14
addq.l  #2,sp
```

Of course, the constant, PHYSBASE, is equal to the function's number, which is 2. After this call, D0 will contain the address of the physical screen. In our program, the first thing we do after the call is copy D0 into A4, where the physical screen address will be safe until we need it.

Manipulating Palettes

Before switching to the picture screen, we must save the system's palette, and then change the system's palette to the picture's palette. Unfortunately, while there is a single call that'll simultaneously set all 16 color values in the system's palette, there is not a similar call for retrieving those values. Instead, we must use 16 separate calls to a function known as *Setcolor*, XBIOS function #7. This function can both set the value of a single color register and return the register's current value to us. Of course, we'll use a loop, so we need have only one actual call to *Setcolor* in our program.

This loop is located right after the call to *Physbase*. First, we load into A6 the address of the buffer in which we'll store the 16 color values. The buffer we'll be using is located in our data section, at *old_palette*, and is 32 bytes, or 16 words, in length--just long enough to store the entire palette. After loading the buffer's address into A6, we initialize a loop counter, D3, to 15, and initialize D4 to be used as an index register. After the loop initialization, we call *Setcolor* like this:

```
move.w   #-1,-(sp)
move.w   d3,-(sp)
move.w   #SETCOLOR,-(sp)
trap     #14
addq.l   #6,sp
```

In the first line, we place onto the stack the color value to which we want the register set. In our case, we use -1, since this tells the function that we don't want to change the register; we only want the current value returned to us. In the second line, we put the color register number onto the stack. The 16 color registers are numbered 0 through 15, so we can use our loop counter, D3, to keep track of the color register numbers as well as the loop count. In the third line, we place the function number onto the stack, and in the fourth we call the XBIOS with the usual *trap #14*. Finally, as always, we clean up the stack.

After the call to *Setcolor*, the color register's old setting (and, in our case, this setting also happens to be the current setting, since we didn't change the register's value) will be in D0. We place that value into its appropriate position in *old_palette* with this instruction:

```
move.w   dØ,Ø(a6,d4)
```

Here, we're using address register indirect addressing with index in our destination argument. Remember: the destination address is calculated by adding the address register (in this case, A6), the index (stored in D4), and the offset (0). The first time through the loop, we'll be saving the value of color register 15 (the value stored in D0) into the last two bytes of *old_palette* (whose address is stored in A6).

After saving the retrieved color value, we subtract 2 from D4, so that it will index the next two bytes (working backward) of our buffer. Then the *dbra* instruction decrements D3, our loop counter and color-register number, and sends us back to the label *color_loop* for the next iteration. When D3 becomes -1, the looping stops and program execution drops down to the next instruction. At that point, we will have grabbed the values of all 16 color registers and stored them in our buffer, *old_palette*, where they'll be when we want to restore them.

With the old color values safely tucked away, we're ready to set the system's colors to the picture's palette. The XBIOS function #6, *Setpalette*, can handle that for us. We call it in our program like this:

```
move.l   #new_palette,-(sp)
move.w   #SETPALETTE,-(sp)
trap     #14
addq.l   #6,sp
```

As you can see, the only two arguments we need to pass to this function are the address of the buffer that holds the new color values and the function number. In our program, *new_palette* holds the palette data we loaded from the picture file.

Flipping Screens

Are you ready for the magic? We can now display the picture. We do this by changing the address of the physical screen to the address of our picture data. If you recall, we have the address of our picture data stored in register A3. To display the picture, we use the XBIOS function #5, *Setscreen*, to switch to the new screen (see figure 11.2). In our program, we call the function like this:

```
move.w  #-1,-(sp)
move.l  a3,-(sp)
move.l  a3,-(sp)
move.w  #SETSCREEN,-(sp)
trap    #14
add.l   #12,sp
```

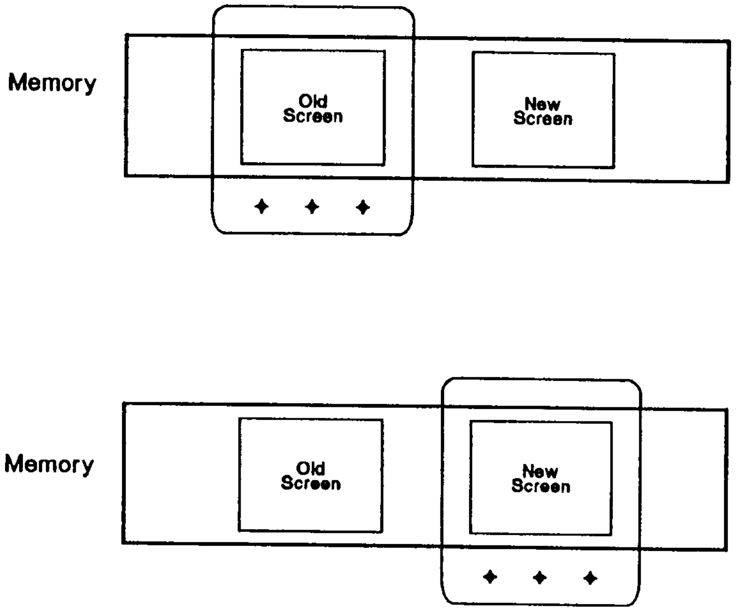


Figure 11.2 • Effect of SETSCREEN

In the first line, we're placing the code for the new resolution onto the stack. (Yes, you can use *Setscreen* to change screen resolution, but, unless you really know what you're doing, I wouldn't advise fooling with this feature.) By using -1 as the resolution code, we're telling the function to leave the resolution as it is. In the second and third lines, we're placing onto the stack the addresses of the physical screen and the logical screen, respectively. Because we want both set to the same address--our picture--we pass the address stored in A3 for both parameters. (Actually, because we aren't going to be writing to the screen, we don't need to set the logical address. We could have left the logical address set to its original value by passing a -1 as its address.) The last parameter is the function number, 5.

After the *Setscreen* function call...*presto!*...the picture is on the screen. To end the program, all we must do is reset the palette and the screen address to their original values, which we do with two more calls to *Setpalette* and *Setscreen*, using the values we saved previously.

Conclusion

That's all there is to loading and displaying a DEGAS-format picture file. To make the new function calls in this program as understandable as possible, I resisted the urge to program them all as macros. But in the spirit of readable code, why don't you now take the time to change the *Physbase*, *Getrez*, *Setscreen*, *Setpalette*, and *Setcolor* function calls into macros? For example, you might write the Setcolor macro like this:

```
set_color macro
    ifc   '','\2'
    fail  Missing parameters!
    mexit
    endc
    move.w   \1,-(sp)
    move.w   \2,-(sp)
    move.w   #SETCOLOR,-(sp)
    trap     #14
    addq.l   #6,sp
  endm
```

You could then set (or get) a color register with a simple, one-line call like this:

```
set_color #-1,#10
```

Also, notice that each error-message routine in the program is virtually identical. You might want to convert all of them into a single subroutine and reduce the size of the assembled program. Remember, there's only three things that can guarantee your successful programming: practice, practice, and practice.

NOTE: Because of the assumptions they make regarding the size of the monitor's screen and screen memory, the graphics functions discussed in this chapter are guaranteed to work properly only with standard Atari monitors.

Summary

✓ XBIOS function #4, *Getrez*, retrieves the system resolution and returns it in D0. It requires only one argument, the function number, to be placed on the stack before it is called with *trap #14*.

✓ The ST's screen memory, which is 32,000 bytes in length, must always start at a 256-byte boundary in memory.

✓ The physical screen is the part of memory that is displayed on your monitor. The logical screen is the part of memory to which all screen writes are directed. Usually, the physical screen and logical screen are set to the same address, but they can be different.

✓ XBIOS function #2, *Physbase*, retrieves the physical screen address and stores it in D0. One argument, the function number, must be placed onto the stack before calling the function with *trap #14*.

- ✓ XBIOS function #7, *Setcolor*, sets or retrieves the color value of a single color register. Before calling the function with *trap #14*, three arguments must be placed onto the stack: the new color value (-1 for no change), the color register number (0 through 15), and the function number. After the call, D0 will contain the color register's old value.
- ✓ XBIOS function #6, *Setpalette*, changes the entire system palette (all 16 color registers) with a single call. It requires that two arguments be placed onto the stack before the call: the address of the new palette (containing 16 word values) and the function number.
- ✓ XBIOS function #5, *Setscreen*, changes the physical screen and the logical screen to new locations in memory. It can also be used to change the system's resolution. Before the call, four arguments must be placed onto the stack: the new resolution (-1=no change, 0=low, 1=medium, and 2=high), the physical screen address (-1 for no change), the logical screen address (-1 for no change), and the function number.

That's All Folks

Congratulations! You've reached the end of *The ST Assembly Language Workshop, Volume 1*. If you've studied hard, you're all set to write TOS programs on your ST. In volume 2 of the workshop, we'll start learning to tame the beast called GEM, using what we've learned so far about 68000 assembly language.

```
;------------------------------------------------------------------
; THE ST ASSEMBLY LANGUAGE WORKSHOP
; PROGRAM 9
;
; COPYRIGHT 1991 BY CLAYTON WALNUM
;------------------------------------------------------------------

;------------------------------------------------------------------
; This macro opens a file using the unbuffered, Fopen GEMDOS
; function. It requires three parameters: the file mode, the address
; of the filename, and the address for file-handle storage.
;------------------------------------------------------------------
open_file macro
        ifc     '','\3'
        fail    Missing parameters!
        mexit
        endc
        move.w  #\1,-(sp)
        pea     \2
        move.w  #FOPEN,-(sp)
        trap    #1
        addq.l  #8,sp
        move.w  d0,\3
        endm
```

```
;--------------------------------------------------------------------
; This macro reads a file using the unbuffered, Fread GEMDOS
; function. It requires three parameters: the buffer address,
; the number of bytes to read, and the file handle.
;--------------------------------------------------------------------
read_file macro
        ifc     '','\3'
        fail    Missing parameters!
        mexit
        endc
        pea     \1
        move.l  #\2,-(sp)
        move.w  \3,-(sp)
        move.w  #FREAD,-(sp)
        trap    #1
        add.l   #12,sp
        endm

;--------------------------------------------------------------------
; This macro closes a file using the unbuffered, Fclose GEMDOS
; function. It requires one parameter: the file handle.
;--------------------------------------------------------------------
close_file macro
        ifc     '','\1'
        fail    Missing parameter!
        mexit
        endc
        move.w  \1,-(sp)
        move.w  #FCLOSE,-(sp)
        trap    #1
        addq.l  #4,sp
        endm
;-------------------------------------------------------------------
; This macro prints a string to the screen, using the Cconws GEMDOS
; function. It requires one parameter, the address of the string
; to print.
;-------------------------------------------------------------------
print   macro
        ifc     '','\1'
        fail    Missing parameter!
        mexit
        endc
        move.l  #\1,-(sp)
        move.w  #CCONWS,-(sp)
        trap    #1
        addq.l  #6,sp
        endm

;-------------------------------------------------------------------
; This macro gets a string from the keyboard using the Cconrs GEMDOS
; function. It requires two parameters: the address of the buffer
; in which to store the string and the max number of characters
; to read.
;----------------------------------------------------------------
input_s macro
        ifc     '','\2'
        fail    Missing parameters!
        mexit
        endc
        move.b  #\2,\1
        move.l  #\1,-(sp)
        move.w  #CCONRS,-(sp)
        trap    #1
        addq.l  #6,sp
        endm

;----------------------------------------------------------------
; This macro gets a single character from the keyboard, using the
; Cconin GEMDOS function.
;----------------------------------------------------------------
get_char        macro
        move.w  #CCONIN,-(sp)
        trap    #1
        addq.l  #2,sp
        endm
```

```
;------------------------------------------------------------------
; MAIN PROGRAM
;------------------------------------------------------------------
        text
get_source:
        jsr     get_filename                ; Get source filename.
        open_file READ_ONLY,filename+2,handle ; Open source file.
        tst.l   handle                      ; Error opening file?
        bge     read_rez                    ; Nope, go read file.
        print   msg3                        ; "File open error!"
        print   msg4                        ; "Exit (y/n)?"
        jsr     get_answer
        bne     out                         ; User typed "Y".
        bra     get_source                  ; Try for filename again.
read_rez:
        read_file pic_rez,2,handle          ; Read picture resolution.
        cmp.w   #2,d0                       ; Read 2 bytes OK?
        beq     get_rez                     ; Yep.
        print   msg5                        ; "File read error!"
        print   msg4                        ; "Exit? (Y/N)"
        jsr     get_answer
        bne     out                         ; User typed "Y".
        close_file handle
        bra     get_source
```

```
get_rez:
        move.w  #GETREZ,-(sp)           ; Get system resolution.
        trap    #14
        addq.l  #2,sp

        cmp.w   pic_rez,d0              ; Pic & system rez the same?
        beq     read_palette            ; Yep.
        print   msg2                    ; "Wrong resolution!"
        print   msg4                    ; "Exit? (Y/N)"
        jsr     get_answer
        bne     out
        close_file handle
        bra     get_source

read_palette:
        read_file new_palette,32,handle ; Read picture palette.
        cmp.w   #32,d0                  ; Got all 32 bytes?
        beq     set_boundary            ; Sure did.
        print   msg5                    ; "File read error!"
        print   msg4                    ; "Exit? (Y/N)"
        jsr     get_answer
        bne     out
        close_file handle
        bra     get_source
```

```
set_boundary:
        move.l   #buffer,d0                  ; Get address of buffer.
        andi.l   #$ffffff00,d0               ; Set buffer address...
        addi.l   #256,d0                     ; to 256-byte boundary.
        move.l   d0,a3                       ; Save recalculated address.

read_pic:
        read_file (a3),32000,handle          ; Read picture data.
        cmp.w    #32000,d0                   ; Got 32,000 bytes?
        beq      show_pic                    ; Yes.
        print    msg5                        ; "File read error!"
        print    msg4                        ; "Exit? (Y/N)"
        jsr      get_answer
        bne      out
        close_file handle
        bra      get_source
show_pic:
        move.w   #PHYSBASE,-(sp)             ; Get addr of screen mem.
        trap     #14
        addq.l   #2,sp
        move.l   d0,a4                       ; Save screen address.

        move.l   #old_palette,a6             ; Get addr of buffer.
        move.l   #15,d3                      ; Init loop counter.
        move.l   #30,d4                      ; Init address index.
```

```
color_loop:
        move.w  #-1,-(sp)               ; Get value of color reg.
        move.w  d3,-(sp)
        move.w  #SETCOLOR,-(sp)
        trap    #14
        addq.l  #6,sp

        move.w  d0,0(a6,d4)             ; Save color value in buffer.
        subi.w  #2,d4                   ; Calculate next addr index.
        dbra    d3,color_loop           ; Loop for next color value.

        move.l  #new_palette,-(sp)      ; Set to picture colors.
        move.w  #SETPALETTE,-(sp)
        trap    #14
        addq.l  #6,sp

        move.w  #-1,-(sp)               ; Set screen to picture.
        move.l  a3,-(sp)
        move.l  a3,-(sp)
        move.w  #SETSCREEN,-(sp)
        trap    #14
        add.l   #12,sp

        get_char
        move.w  #-1,-(sp)                ; Set back to old screen.
        move.l  a4,-(sp)
        move.l  a4,-(sp)
        move.w  #SETSCREEN,-(sp)
        trap    #14
        add.l   #12,sp

        move.l  #old_palette,-(sp)       ; Restore old palette.
        move.w  #SETPALETTE,-(sp)
        trap    #14
        addq.l  #6,sp

out:    close_file handle                ; close picture file.
        move.w  #PTERM0,-(sp)            ; Back to desktop.
        trap    #1

;--------------------------------------------------------------------
; This subroutine adds a null to the string retrieved with a call
; to the GEMDOS function Cconrs.
;
; Input: Address of string buffer in a3.
; Output: Null after last character in string.
; Registers changed: NONE.
;--------------------------------------------------------------------
add_null:
        movem.l a3/d3,-(sp)
        clr.l   d3
        move.b  1(a3),d3
        move.b  #0,2(a3,d3)
        movem.l (sp)+,a3/d3
        rts

;------------------------------------------------------------------
; This subroutine retrieves the name of the source file.
;
; Input:         NONE.
; Output:        Null-terminated filename in filename.
; Regs changed:  NONE.
; Calls:         print, input_s, add_null
; Data used:     msg1, filename, crlf
;------------------------------------------------------------------
get_filename:
        print    msg1                        ; "Enter name of file..."
        input_s  filename,64                 ; get filename
        print    crlf
        move.l   #filename,a3                ; add a null to...
        jsr      add_null                    ; the filename.
        rts

;------------------------------------------------------------------
```

```
; This subroutine retrieves a one-character response from the
; keyboard and compares it against Y and N.
;
; Input:        NONE.
; Output:       1->D0 if Y typed. 0->D0 if any other key typed.
; Regs changed: D0.
; Calls:        get_char
; Data used:    UC_Y, LC_Y
;-----------------------------------------------------------------
get_answer:
        get_char                            ; get user's answer.
        move.l  d0,d3
        print   crlf
        cmp.w   #UC_Y,d3                    ; user typed "Y"?
        beq     .yes                        ; yep.
        cmp.w   #LC_Y,d3                    ; user typed "y"?
        beq     .yes                        ; yep.
        clr.l   D0
        bra     .out
.yes:   moveq.l #1,D0
.out:
        rts
end_get_answer:
```

```
;--------------------------------------------------------------------

        data

READ_ONLY       equu     0
UC_Y            equ     89
LC_Y            equ     121

PTERM0          equ     0
CCONIN          equ     1
PHYSBASE        equ     2
GETREZ          equ     4
SETSCREEN       equ     5
SETPALETTE      equ     6
SETCOLOR        equ     7
CCONWS          equ     9
CCONRS          equ     10
FOPEN           equ     61
FCLOSE          equ     62
FREAD           equ     63

msg1:   dc.b    "Enter name of DEGAS file: ",0
msg2:   dc.b    "Wrong resolution!",13,10,0
msg3:   dc.b    "File open error!",13,10,0
msg4:   dc.b    "Exit (y/n)?",0
```

```
msg5:       dc.b    "File read error!",13,10,0
crlf:       dc.b    13,10,0

            bss

            even
handle:             ds.w    1
filename:           ds.b    67
pic_rez:            ds.w    1
new_palette:        ds.w    16
old_palette:        ds.w    16
buffer:             ds.b    32256
```

A
Assembly Instructions

Although the programs in this book were developed using *DevPac2* from HiSoft (distributed in the USA by Goldleaf Publishing), all the programs are also compatible with Atari's *MadMac* assembler, which comes with the Atari Developer's Kit. However, certain changes are required if you want to take advantage of this compatibility.

First, in some cases, *MadMac* uses different assembler directives than *DevPac2*, meaning you must modify certain sections of code--specifically, the macros. We've included modification instructions below. In addition, the disk contains both *DevPac2* and *MadMac* versions of the programs, when appropriate.

Second, *DevPac2* and *MadMac* use different procedures for assembling source code into executable form. General assembly instructions for both assemblers are also included below.

Assembling with DevPac2

To assemble a program with *DevPac2*, first load the assembler, by double-clicking the GENST2.PRG file. When the assembler loads, click the Load entry of the File menu to load the source code you want to assemble (see figure A.1). When the source code is loaded, click the Assemble entry of the Program menu. The dialog box shown in figure A.2 will appear. By clicking the dialog's Assemble button, you can assemble the program into memory, from which you can run it using the Run option of the Program menu.

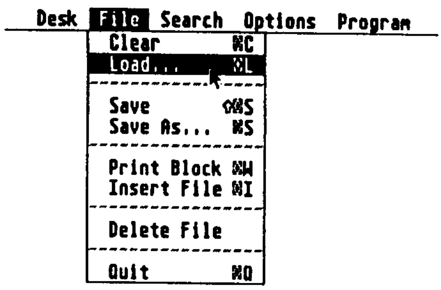
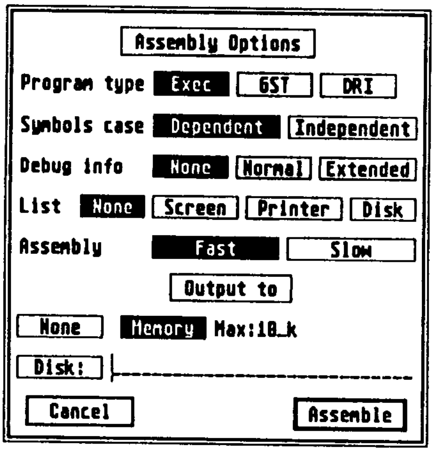


Figure A.1



Assembly Options

Program type [✓ Exec] GST DRI

Symbols case [✓ Dependent] Independent

Debug info [✓ None] Normal Extended

List [✓ None] Screen Printer Disk

Assembly [✓ Fast] Slow

Output to

None [✓ Memory] Max:10_k

Disk: ________________

Cancel Assemble

Figure A.2

When your program is complete, you'll want to assemble it to disk, so it can be run by others. To do this, click the Disk button before assembling. The program will then be assembled to disk, into an executable file. For example, if you named your source MYPROG.S, the assembler will create a runnable file named MYPROG.PRG. If you'd like the assembled file to have a different name, type the new name in the space following the Disk button.

Assembling with MadMac

MadMac uses a command-line interpreter to accept instructions from the user, unlike *DevPac2*, which features a complete GEM interface. To load *MadMac*, double-click the file MAC.PRG. When the program loads, you'll see a "*" prompt. To assemble a source-code file, type the filename and press return. (Note: The file must have a .S extension and, for ease of access, be located in the same directory as the MAC.PRG file.) When you assemble code this way, the assembler creates a .O file (object file) on your disk. This type of file is not executable. (It must first be linked with other routines.)

To assemble a program into an executable disk file, type the source code's filename followed by the *-p* switch. That is, to assemble a program called MYPROG.S located in the same directory as MAC.PRG, type *MYPROG -p* at the *MadMac* prompt. The assembler will name the resultant, executable file MYPROG.PRG.

Changes for MadMac

Because *MadMac* features a different set of assembler directives for use with conditional assembly, users of that assembler must modify macros written for *DevPac2*. Using the *input_s* macro from chapter 9 as an example, here are the changes required:

```
.macro  input_s
  .if \?2
    move.b   #\2,\1
    move.l   #\1,-(sp)
    move.w   #CCONRS,-(sp)
    trap     #1
    addq.l   #6,sp
  .else
   .assert \?2
  .endif
.endm
```

In the preceeding macro, the *.if*, *.else*, and *.endif* assembler directives work just as we would expect. If the line *.if \?2* evaluates to true, *MadMac* assembles the statements between the *.if* and the *.else*. Otherwise, the assembler will jump to the statements between the *.else* and the *.endif* (which marks the end of the "if" construct). In *input_s*, there is, between the *.else* and *.endif*, only an assembler directive that prints an error message to the screen.

The "\?2" following the *.if* and *.assert* directives is interpreted as "if parameter 2 is specified and non-empty." This condition allows us to check whether the correct number of parameters have been sent to the macro. The *.assert* directive evaluates the expressions following it (there can be several expressions, separated by commas), and if any equals 0, the assembler prints a warning to the screen.

Notice also that the *macro* and *endm* directives are used differently (as compared with *DevPac2*). With *MadMac* both directives must be preceded by periods. In addition, you must place the *.macro* directive before the macro's name rather than after it.

Other macros throughout the book require similar changes if they are to be used with *MadMac*. For your convenience, *MadMac* versions of programs that need these changes are included on your *Assembly Language Workshop* disk.

B

68000 Instruction Reference

by Bryan P. Schappel

This appendix gives a complete list of the 68000 assembly language instructions, their syntax, valid addressing modes, and affected Condition Code Register (CCR) flags. Figure B.1 illustrates the layout of the entries in the reference section.

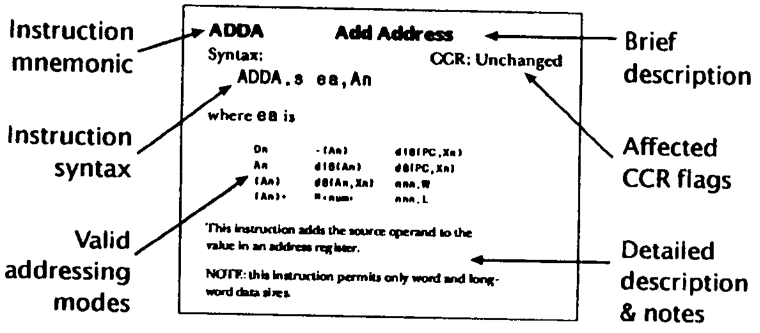


Figure B.1 • Layout of Reference Entries

On the following page you will find a list of abbreviations that are used throughout the reference. The abbreviations are not only used in the instruction descriptions, but in the instruction syntax as well.

A list of addressing modes is also included. This list gives the proper syntax and a brief description of each of the M68000's addressing modes.

Abbreviations

An	Address register 0-7
CCR	Condition Codes Register
Dn	Data register 0-7
d8	8-bit displacement value
d16	16-bit displacement value
ea	effective address
nnn.L	32-bit address
nnn.W	16-bit address
PC	Program counter
SP	Stack pointer
SR	Status register
Rn	Any 68000 data or address register
Xn	Register used as an index
.s	Operand size (Byte, Word, Long)

Addressing Modes

Dn	Data register direct
An	Address register direct
(An)	Address register indirect
(An)+	Address register indirect with post-increment
-(An)	Address register indirect with pre-decrement
d16(An)	Address register indirect with 16-bit displacement
d8(An,Xn)	Address register indirect with 8-bit displacement and index
nnn.W	Absolute short
nnn.L	Absolute long
#<num>	Immediate
d16(PC)	PC indirect with 16-bit displacement
d8(PC,Xn)	PC indirect with 8-bit displacement and index

ABCD — Add Binary Coded Decimal with Extend

Syntax: CCR: ZCX

```
ABCD      Dn,Dn
ABCD      -(An),-(An)
```

This instruction affects only eight bits of data. It adds the source operand, the value in the eXtend bit, the byte in the destination, and stores the result in the destination operand.

ADD — Add Binary

Syntax: CCR: NZVCX

```
ADD.s     ea1,Dn
ADD.s     Dn,ea2
```

where `ea1` is

```
Dn        -(An)        d16(PC,Xn)
An        d16(An)      d8(PC,Xn)
(An)      d8(An,Xn)    nnn.W
(An)+     #<num>       nnn.L
```

where `ea2` is

```
          -(An)        d16(PC,Xn)
          d16(An)      d8(PC,Xn)
(An)      d8(An,Xn)    nnn.W
(An)+     #<num>       nnn.L
```

This instruction adds the source and destination operands together and stores the result in the destination operand.

NOTE: at least one of the operands must be a data register.

ADDA — Add Address

Syntax: CCR: Unchanged

```
    ADDA.s  ea,An
```

where ea is

```
Dn        -(An)       d16(PC,Xn)
An        d16(An)     d8(PC,Xn)
(An)      d8(An,Xn)   nnn.W
(An)+     #<num>      nnn.L
```

This instruction adds the source operand to the value in an address register.

NOTE: this instruction permits only word and long-word data sizes.

ADDI — Add Immediate

Syntax: CCR: NZVCX

```
    ADDI.s    #<num>,ea
```

where ea is

```
Dn        -(An)       d16(PC,Xn)
          d16(An)     d8(PC,Xn)
(An)      d8(An,Xn)   nnn.W
(An)+     #<num>      nnn.L
```

This adds the immediate value of the source operand to the destination operand and stores the result in the destination.

ADDQ Add Quick

Syntax: CCR: NZVCX

```
ADDQ.s    #<num>,ea
```

where ea is

```
Dn        -(An)
An        d16(An)
(An)      d8(An,Xn)    nnn.W
(An)+                  nnn.L
```

This instruction adds a value between 1 and 8 to the destination operand and stores the result in the destination.

ADDX Add Binary With Extend

Syntax: CCR: ZCX

```
ADDX.s    Dsrc,Ddest
ADDX.s    -(Asrc),-(Adest)
```

This instruction adds the value in the source operand, the value of the eXtend bit, the destination, and stores the value in the destination.

AND Logical AND

Syntax: CCR: NZVC

```
AND.s    ea,Dn
AND.s    Dn,ea
```

where the destination is a data register, ea is

```
Dn       -(An)        d16(PC,Xn)
         d16(An)      d8(PC,Xn)
(An)     d8(An,Xn)    nnn.W
(An)+    #<num>       nnn.L
```

where the source is a data register, ea is

```
         -(An)
         d16(An)
(An)     d8(An,Xn)    nnn.W
(An)+                 nnn.L
```

This instruction performs a bitwise logical AND with the source and destination operands. The result is stored in the destination.

NOTE: At least one of the operands must be a data register.

ANDI AND Immediate

Syntax: CCR: NZVC

```
ANDI.s   #<num>,ea
```

where ea is

```
Dn       -(An)
         d16(An)
(An)     d8(An,Xn)    nnn.W
(An)+                 nnn.L
```

This instruction performs a bitwise logical AND with the immediate source operand and stores the result in the destination.

ANDI to CCR — AND Immediate Data To the Condition Codes

Syntax:

CCR: NZVCX

```
ANDI    #<num>,CCR
```

This instruction performs a bitwise logical AND with the immediate source operand and the CCR. This instruction may be used to clear one or more of the condition code bits.

ANDI to SR — AND Immediate To the Status Register (Privileged)

Syntax:

CCR: NZVCX

```
ANDI    #<num>,SR
```

This instruction performs a bitwise logical AND with the 16-bit immediate source and the status register. This may only be executed in supervisor mode.

ASL and ASR Arithmetic Shift Left And Right

Syntax: CCR: NZVCX

```
ASL.s   Dn,Dn
ASL.s   #<num>,Dn
ASL     ea
ASR.s   Dn,Dn
ASR.s   #<num>,Dn
ASR     ea
```

where ea is

```
         -(An)
         d16(An)
(An)     d8(An,Xn)   nnn.W
(An)+                nnn.L
```

These instructions will shift contents of the destination operand a given number of times. If the destination is a data register, the number of shifts may be 1-8. If the destination is a memory location it may be shifted only one bit at a time.

Bcc Branch On Condition Code

Syntax: CCR: Unchanged

```
    Bcc         displacement
```

The 15 different branch instructions are:

```
BHI -- Branch Hi
BLS -- Branch if Low or Same
BCC -- Branch on Carry Clear
BCS -- Branch on Carry Set
BNE -- Branch on Not Equal
BEQ -- Branch if Equal
BVC -- Branch if Overflow Clear
BVS -- Branch if Overflow Set
BPL -- Branch if Plus
BMI -- Branch if Minus
BGE -- Branch if Greater Than or Equal
BLT -- Branch if Less Than
BGT -- Branch if Greater Than
BLE -- Branch if Less Than or Equal
BRA -- Branch Always
```

These instructions test a specific bit in the CCR and branch if that bit is set.

BCHG Test a Bit and Change

Syntax: CCR: Z

```
    BCHG.s   Dn,ea
    BCHG.s   #<num>,ea
```

where ea is

```
        Dn        -(An)
                  d16(An)
        (An)      d8(An,Xn)    nnn.W
        (An)+                  nnn.L
```

and where .s is .B or .L
*** .L is only valid when ea is a data register

This instruction tests a bit value, moves the bit value into the Zero flag, and flips the value of the bit. The operand may be 32 bits wide if it resides in a data register and only eight bits wide if it is in memory.

BCLR Test a Bit and Clear

Syntax: CCR: Z

```
    BCLR.s   Dn,ea
    BCLR.s   #<num>,ea
```

where ea is

```
        Dn        -(An)
                  d16(An)
        (An)      d8(An,Xn)    nnn.W
        (An)+                  nnn.L
```

and where .s is .B or .L
*** .L is only valid when ea is a data register

This instruction tests a bit, sets the Zero flag, and clears the bit.

BSET Test Bit and Set

Syntax: CCR: Z

```
BSET.s    Dn,ea
BSET.s    #<num>,ea
```

where ea is

```
Dn        -(An)
          d16(An)
(An)      d8(An,Xn)    nnn.W
(An)+                  nnn.L
```

This instruction tests the value of a bit, sets the Zero flag, and sets the bit's value to 1.

BSR Branch to Subroutine

Syntax: CCR: Unchanged

```
BSR       offset
```

This instruction branches to a subroutine at the address: OFFSET + PC + 2. The offset is an address label defined in the program's text area.

BTST Test Bit

Syntax: CCR: Z

```
BTST.s    Dn,ea
BTST.s    #<num>,ea
```

where ea is

```
Dn        -(An)        d16(PC,Xn)
          d16(An)      d8(PC,Xn)
(An)      d8(An,Xn)    nnn.W
(An)+                  nnn.L
```

This instruction tests the value of a bit and sets the Zero flag accordingly.

CHK Check Register Against Boundaries

Syntax: CCR: N

```
    CHK.s     ea,Dn
```

where ea is

```
Dn        -(An)       d16(PC,Xn)
          d16(An)     d8(PC,Xn)
(An)      d8(An,Xn)   nnn.W
(An)+     #<num>      nnn.L
```

This instruction compares the source operand to a data register. If the value in the data register is negative or greater than the source operand the 68000 traps through exception vector #6.

CLR Clear an Operand

Syntax: CCR: NZVC

```
    CLR.s     ea
```

where ea is

```
Dn        -(An)
          d16(An)
(An)      d8(An,Xn)   nnn.W
(An)+                 nnn.L
```

This instruction zeroes (moves a zero to) the destination.

CMP Compare

Syntax: CCR: NZVC

```
    CMP.s    ea,Dn
```

where ea is

```
Dn       -(An)        d16(PC,Xn)
An       d16(An)      d8(PC,Xn)
(An)     d8(An,Xn)    nnn.W
(An)+    #<num>       nnn.L
```

This instruction subtracts the source operand from a data register (the destination) and sets the CCR accordingly.

CMPA Compare Addresses

Syntax: CCR: NZVC

```
    CMPA.s    ea,An
```

where ea is

```
Dn       -(An)        d16(PC,Xn)
An       d16(An)      d8(PC,Xn)
(An)     d8(An,Xn)    nnn.W
(An)+    #<num>       nnn.L
```

This instruction subtracts the source operand from an address register (the destination) and sets the CCR accordingly.

CMPI Compare Immediate

Syntax: CCR: NZVC

```
    CMPI.s    #<num>,ea
```

where ea is

```
Dn         -(An)          d16(PC,Xn)
           d16(An)        d8(PC,Xn)
(An)       d8(An,Xn)      nnn.W
(An)+                     nnn.L
```

This instruction subtracts the immediate source operand from the destination operand and sets the CCR accordingly.

CMPM Compare Memory

Syntax: CCR: NZVC

```
    CMPM.s    (An)+,(An)+
```

This instruction subtracts the value pointed to by the source address register from the value pointed to by the destination address register and sets the CCR accordingly.

DBcc Test, Decrement, and Branch

Syntax: CCR: Unchanged

```
    DBcc      Dn,displacement
```

The mnemonics for this instruction follow:

```
DBT  -- True
DBF  -- False
DBHI -- High
DBLS -- Low or Same
DBCC -- Carry Clear
DBCS -- Carry Set
DBNE -- Not Equal
DBEQ -- Equal
DBVC -- Overflow Clear
DBVS -- Overflow Set
DBPL -- Plus
DBMI -- Minus
DBGE -- Greater Than or Equal
DBLT -- Less Than
DBGT -- Greater Than
DBLE -- Less Than or Equal
```

This instruction is used to perform a loop. The loop terminates when the CCR matches the given condition or the data register used as the loop counter becomes -1. (This instruction decrements the data register with each iteration of the loop.) Most assemblers will accept DBRA in place of DBF. The displacement in this instruction is a label in the program.

DIVS Signed Divide

Syntax: CCR: NZVC

```
    DIVS.W   ea,Dn
```

where ea is

```
    Dn        -(An)        d16(PC,Xn)
              d16(An)      d8(PC,Xn)
    (An)      d8(An,Xn)    nnn.W
    (An)+     #<num>       nnn.L
```

This instruction divides the destination operand by the source and stores the result in the destination. This performs signed division.

DIVU Unsigned Divide

Syntax: CCR: NZVC

```
    DIVU.W   ea,Dn
```

where ea is

```
    Dn        -(An)        d16(PC,Xn)
              d16(An)      d8(PC,Xn)
    (An)      d8(An,Xn)    nnn.W
    (An)+     #<num>       nnn.L
```

This instruction divides the destination operand by the source operand and stores the result in the destination. This performs unsigned division.

EOR — Exclusive OR

Syntax:

CCR: NZVC

```
    EOR.s     Dn,ea
```

where ea is

```
Dn        -(An)
          d16(An)
(An)      d8(An,Xn)    nnn.W
(An)+                  nnn.L
```

This instruction performs a bitwise Exclusive OR with the contents of the data register and the destination operand. The result is stored in the destination.

EORI — Exclusive OR Immediate

Syntax:

CCR: NZVC

```
    EORI.s     #<num>,ea
```

where ea is

```
Dn        -(An)
          d16(An)
(An)      d8(An,Xn)    nnn.W
(An)+                  nnn.L
```

This instruction performs a bitwise Exclusive OR with the immediate source operand and the contents of the destination. The result is stored in the destination.

EORI to CCR — Exclusive OR Immediate Data to the Condition Codes

Syntax: CCR: NZVCX

```
    EORI      #<num>,CCR
```

This instruction performs a bitwise Exclusive OR with the immediate source operand and the condition codes register. This is an excellent way to invert a number of the CCR's flags at once.

EORI to SR — Exclusive OR Immediate Data to Status Register (Privileged)

Syntax: CCR: NZVCX

```
    EORI      #<num>,SR
```

This instruction performs a bitwise Exclusive OR with the immediate source operand and the status register. This instruction may only be executed in supervisor mode.

EXG — Exchange Registers

Syntax: CCR: Unchanged

```
    EXG       R1,R2
```

This instruction swaps the values of any two registers. The instruction swaps all 32 bits. You may swap data registers, address registers, or a combination.

EXT Sign Extend

Syntax: CCR: NZVC

```
    EXT.s     Dn
```

This instruction sign extends the data register's value.

ILLEGAL Take Illegal Instruction Trap

Syntax: CCR: Unchanged

```
    ILLEGAL
```

This instruction causes the 68000 to trap through exception vector 4 (the Illegal instruction vector).

JMP Jump

Syntax: CCR: Unchanged

```
    JMP       ea
```

where ea is

```
                         d16(PC,Xn)
             d16(An)     d8(PC,Xn)
    (An)     d8(An,Xn)   nnn.W
                         nnn.L
```

This instruction causes program execution to JuMP to the specified address.

JSR — Jump to Subroutine

Syntax: CCR: Unchanged

```
    JSR        ea
```

where ea is

```
                              d16(PC,Xn)
               d16(An)        d8(PC,Xn)
    (An)       d8(An,Xn)      nnn.W
                              nnn.L
```

This instruction calls the subroutine at the specified address. When the subroutine terminates program execution begins at the instruction immediately after the JSR.

LEA — Load Effective Address

Syntax: CCR: Unchanged

```
    LEA        ea,An
```

where ea is

```
                              d16(PC,Xn)
               d16(An)        d8(PC,Xn)
    (An)       d8(An,Xn)      nnn.W
                              nnn.L
```

This instruction calculates an effective address and moves that address into the specified address register. Since this calculation is performed at run-time use of this instruction can help you write relocatable code.

LINK — Link and Allocate Space

Syntax: CCR: Unchanged

```
    LINK      An,#<num>
```

This instruction pushes the contents of the given address register onto the stack, moves the value of the stack pointer into the address register, and then adds the signed 16-bit immediate value to the stack pointer. Use the UNLK instruction to reverse this process.

LSL and LSR — Logical Shift Left and Right

Syntax: CCR: NZVCX

```
    LSL.s     Dn,Dn
    LSL.s     #<num>,Dn
    LSL       ea
    LSR.s     Dn,Dn
    LSR.s     #<num>,Dn
    LSR       ea
```

where ea is

```
              -(An)
              d16(An)
    (An)      d8(An,Xn)    nnn.W
    (An)+                  nnn.L
```

These instructions shift the contents of the destination operand the given number of times. The restrictions of the ASL/ASR instructions also apply here.

MOVE Move Data

Syntax: CCR: NZVC

```
    MOVE.s    ea1,ea2
```

where ea1 is

```
Dn        -(An)        d16(PC,Xn)
An        d16(An)      d8(PC,Xn)
(An)      d8(An,Xn)    nnn.W
(An)+     #<num>       nnn.L
```

and where ea2 is

```
Dn        -(An)
          d16(An)
          d8(An,Xn)    nnn.W
(An)+                  nnn.L
```

This instruction moves the contents of the source operand to the destination operand thus destroying the destination.

MOVEA Move to Address Register

Syntax: CCR: Unchanged

```
    MOVEA.s    ea,An
```

where ea is

```
Dn        -(An)        d16(PC,Xn)
An        d16(An)      d8(PC,Xn)
(An)      d8(An,Xn)    nnn.W
(An)+     #<num>       nnn.L
```

This instruction moves either the word or long-word at the effective address into an address register.

MOVE to CCR Move to the Condition Codes Register

Syntax: CCR: NZVCX

```
    MOVE        ea,CCR
```

where ea is

```
Dn          -(An)          d16(PC,Xn)
            d16(An)        d8(PC,Xn)
(An)        d8(An,Xn)      nnn.W
(An)+       #<num>         nnn.L
```

This instruction moves the word-sized source operand to the CCR. Only the least significant five bits of the source are actually used; the rest are ignored.

MOVE from SR Move From the Status Register

Syntax: CCR: Unchanged

```
    MOVE        SR,ea
```

where ea is

```
Dn          -(An)
            d16(An)
(An)        d8(An,Xn)      nnn.W
(An)+                      nnn.L
```

This instruction moves a copy of the contents of the status register to the destination operand.

MOVE to SR — Move to the Status Register (Privileged)

Syntax: CCR: NZVCX

```
    MOVE      ea,SR
```

where ea is

```
Dn         -(An)        d16(PC,Xn)
           d16(An)      d8(PC,Xn)
(An)       d8(An,Xn)    nnn.W
(An)+      #<num>       nnn.L
```

This instruction moves the word-sized source operand into the status register. This instruction may be executed only in supervisor mode.

MOVE USP — Move To/From the User Stack Pointer (Privileged)

Syntax: CCR: Unchanged

```
    MOVE      USP,An
    MOVE      An,USP
```

This instruction moves the contents of the user stack pointer to or from a given address register. This instruction may be executed only in supervisor mode.

MOVEM Move Multiple

Syntax: CCR: Unchanged

```
MOVEM.s  reg.list,ea
MOVEM.s  ea,reg.list
```

where ea is

```
Dn       -(An)        d16(PC,Xn)
An       d16(An)      d8(PC,Xn)
(An)     d8(An,Xn)    nnn.W
(An)+    #<num>       nnn.L
```

These instructions move the values of specified registers to or from consecutive memory locations.

MOVEP Move Peripheral Data

Syntax: CCR: Unchanged

```
MOVEP.s  Dn,disp(An)
MOVEP.s  disp(An)
```

This instruction moves data between a data register and alternating bytes of memory. (These would be all odd or even addresses.)

MOVEQ Move Quick

Syntax: CCR: NZVC

```
MOVEQ    #<num>,Dn
```

This instruction moves a signed 8-bit number into a data register. The 8-bit value is sign extended through all 32 bits of the register.

MULS Signed Multiply

Syntax: CCR: NZVC

```
MULS.W    ea,Dn
```

where ea is

```
Dn       -(An)        d16(PC,Xn)
         d16(An)      d8(PC,Xn)
(An)     d8(An,Xn)    nnn.W
(An)+    #<num>       nnn.L
```

This instruction performs a signed multiplication of the source and destination operands. It stores the signed result in the destination. The source operand is 16-bits wide. It is multiplied with the low 16-bits of the destination, producing a 32-bit product.

MULU Unsigned Multiply

Syntax: CCR: NZVC

```
MULU.W    ea,Dn
```

where ea is

```
Dn       -(An)        d16(PC,Xn)
         d16(An)      d8(PC,Xn)
(An)     d8(An,Xn)    nnn.W
(An)+    #<num>       nnn.L
```

This instruction multiplies the 16-bit unsigned source with the lower 16-bits of the unsigned destination and stores the unsigned result in the destination. The product is a 32-bit unsigned integer.

NBCD Negate Decimal With Extend

Syntax: CCR: NZVCX

```
    NBCD      ea
```

where ea is

```
        Dn        -(An)
                  d16(An)
        (An)      d8(An,Xn)     nnn.W
        (An)+                   nnn.L
```

This instruction subtracts the value of the eXtend flag and the destination from zero and stores the result back into the destination. This instruction uses BCD (Binary Coded Decimal) arithmetic.

NEG Negate

Syntax: CCR: NZVCX

```
    NEG.s     ea
```

where ea is

```
        Dn        -(An)
                  d16(An)
        (An)      d8(An,Xn)     nnn.W
        (An)+                   nnn.L
```

This instruction calculates the two's complement of the operand. It performs this task by subtracting the operand from zero. The result is stored back in the operand.

NEGX Negate With Extend

Syntax: CCR: NZVCX

```
    NEGX.s    ea
```

where ea is

```
Dn        -(An)
          d16(An)
(An)      d8(An,Xn)     nnn.W
(An)+                   nnn.L
```

This instruction subtracts the value of the eXtend flag and the operand from zero. It then places the result back into the operand.

NOP No Operation

Syntax: CCR: Unchanged

```
    NOP
```

This instruction does not perform a useful function except consume a clock cycle. Its most common uses are for timing loops and replacing questionable instructions while debugging.

NOT — Logical Complement

Syntax:

CCR: NZVC

```
    NOT         ea
```

where ea is

```
Dn          -(An)
            d16(An)
(An)        d8(An,Xn)     nnn.W
(An)+                     nnn.L
```

This instruction performs a logical bitwise complement of the operand. It turns each 1 into a 0 and each 0 into a 1. (This is the same as EOR'ing the operand with $FFFFFFFF.)

OR — Inclusive Logical OR

Syntax:

CCR: NZVC

```
    OR.s        ea1,Dn
    OR.s        Dn,ea2
```

where ea1 is

```
Dn          -(An)         d16(PC,Xn)
            d16(An)       d8(PC,Xn)
(An)        d8(An,Xn)     nnn.W
(An)+       #<num>        nnn.L
```

where ea2 is

```
            -(An)
            d16(An)
(An)        d8(An,Xn)     nnn.W
(An)+                     nnn.L
```

This instruction performs a bitwise logical OR with the source and destination operands and stores the result back into the destination.

ORI — Inclusive OR Immediate

Syntax: CCR: NZVC

```
ORI.s     #<num>,ea
```

where ea is

```
Dn        -(An)
          d16(An)
(An)      d8(An,Xn)     nnn.W
(An)+                   nnn.L
```

This instruction performs a bitwise logical OR with the immediate source operand and the destination operand, storing the result back into the destination.

ORI to CCR — Inclusive OR Immediate to Condition Codes Register

Syntax: CCR: NZVCX

```
ORI      #<num>,CCR
```

This instruction performs a bitwise logical OR with the immediate data and the CCR.

ORI to SR — Inclusive OR Immediate to Status Register (Privileged)

Syntax: CCR: NZVCX

```
ORI      #<num>,SR
```

This instruction performs a bitwise logical OR between the immediate data and the SR. This instruction may be executed only in supervisor mode.

PEA Push Effective Address

Syntax: CCR: Unchanged

```
    PEA        ea
```

where ea is

```
                          d16(PC,Xn)
             d16(An)      d8(PC,Xn)
    (An)     d8(An,Xn)    nnn.W
                          nnn.L
```

This instruction calculates an actual 32-bit address from the given effective address and then pushes the calculated address onto the stack.

NOTE: This instruction pushes the address onto the stack NOT the value at the address!

RESET Reset External Devices (Privileged)

Syntax: CCR: Unchanged

```
    RESET
```

This instruction tells the 68000 to raise its output signal, RESET, thus signaling external devices to perform a reset. This instruction does not interfere with the 68000. After the RESET instruction has executed, the processor moves to the next instruction. This instruction may be executed only in supervisor mode.

ROL and ROR Rotate Left and Right

Syntax: CCR: NZVC

```
    ROR.s    Dn,Dn
    ROR.s    #<num>,Dn
    ROR      ea
    ROL.s    Dn,Dn
    ROL.s    #<num>,Dn
    ROL      ea
```

where ea is

```
                -(An)
                d16(An)
    (An)        d8(An,Xn)    nnn.W
    (An)+                    nnn.L
```

These instructions rotate the bits in the operand either left or right. If the destination is a register, the shift may be from one to eight bits. If the destination is a memory location only one bit shifts are allowed.

ROXL & ROXR Rotate Left and Right With Extend

Syntax: CCR: NZVCX

```
ROXR.s   Dn,Dn
ROXR.s   #<num>,Dn
ROXR     ea
ROXL.s   Dn,Dn
ROXL.s   #<num>,Dn
ROXL     ea
```

where ea is

```
            -(An)
            d16(An)
(An)        d8(An,Xn)   nnn.W
(An)+                   nnn.L
```

These instructions rotate bits in the operand either left or right, just like ROL/ROR, except the eXtend bit is included in the rotatation allowing multiprecision shifts longer that 32 bits. The restrictions of ROR/ROL apply here as well.

RTE Return From Exception (Privileged)

Syntax: CCR: NZVCX

```
RTE
```

This instruction restores the state of the 68000 after an exception routine has been executed. At the very least the SR and PC are restored (other registers may be restored depending on the contents of the stack and internal processor state). This instruction may be executed only in supervisor mode.

RTR — Return and Restore Condition Codes

Syntax: CCR: NZVCX

```
    RTR
```

This instruction removes a word from the stack and then moves the lowest five bits of the word into the CCR. Then the 68000 pulls a longword from the stack and transfers that value into the PC. Program execution continues at the new PC.

RTS — Return From Subroutine

Syntax: CCR: Unchanged

```
    RTS
```

This instruction pulls a longword from the stack and moves it into the PC. Program execution then continues at the new PC. (This is the complement to the JSR and BSR instructions.)

SBCD — Subtract BCD With Extend

Syntax: CCR: NZVCX

```
    SBCD      Dn,Dn
    SBCD      -(An),-(An)
```

This instruction subtracts the value of the eXtend flag and the source operand from the destination and stores the result in the destination. The instruction uses BCD arithmetic.

Scc Set According to Condition

Syntax:

```
    Scc        ea
```

CCR: Unchanged

where ea is

```
Dn          -(An)
            d16(An)
(An)        d8(An,Xn)    nnn.W
(An)+                    nnn.L
```

The mnemonics for this instruction follow:

```
ST  -- True
SF  -- False
SHI -- High
SLS -- Same or low
SCC -- Carry Clear
SCS -- Carry Set
SNE -- Not Equal
SEQ -- Equal
SVC -- Overflow Clear
SVS -- Overflow Set
SPL -- Plus
SMI -- Minus
SGE -- Greater or Equal
SLT -- Less than
SGT -- Greater than
SLE -- Less or Equal
```

This instruction tests the state of a given condition. If the condition is TRUE, the processor sets the destination byte to all ones; if the condition is FALSE, the processor sets the byte to all zeroes.

STOP — Load Status Register and Stop (Privileged)

Syntax: CCR: NZVCX

```
    STOP      #<num>
```

This instruction loads the 16-bit immediate value into the status register and stops the further execution of instructions.

SUB — Subtract Binary

Syntax: CCR: NZVCX

```
    SUB.s     ea1,Dn
    SUB.s     Dn,ea2
```

where ea1 is

```
Dn        -(An)        d16(PC,Xn)
An        d16(An)      d8(PC,Xn)
(An)      d8(An,Xn)    nnn.W
(An)+     #<num>       nnn.L
```

and where ea2 is

```
          -(An)
          d16(An)
(An)      d8(An,Xn)
(An)+                  nnn.W
                       nnn.L
```

This instruction subtracts the source operand from the destination and stores the result back into the destination. At least one of the operands must be a data register. If an address register is used as an operand, the only valid data sizes are word and longword.

SUBA Subtract Address

Syntax: CCR: Unchanged

```
SUBA.s    ea,An
```

where ea is

```
Dn        -(An)        d16(PC,Xn)
An        d16(An)      d8(PC,Xn)
(An)      d8(An,Xn)    nnn.W
(An)+     #<num>       nnn.L
```

This instruction subtracts a word or longword from an address register and stores the result back into the address register. The result is always 32-bits wide.

SUBI Subtract Immediate

Syntax: CCR: NZVCX

```
SUBI.s    #<num>,ea
```

where ea is

```
Dn        -(An)
          d16(An)
(An)      d8(An,Xn)    nnn.W
(An)+                  nnn.L
```

This instruction subtracts the immediate source data from the destination and stores the difference back into the destination.

SUBQ Subtract Quick

Syntax: CCR: NZVCX

```
    SUBQ      #<num>,ea
```

where ea is

```
        Dn      -(An)
        An      d16(An)
        (An)    d8(An,Xn)   nnn.W
        (An)+               nnn.L
```

This instruction subtracts the immediate source data, which can have a value from 1 to 8, from the destination and stores the result back into the destination.

SUBX Subtract With Extend

Syntax: CCR: NZVCX

```
    SUBX.s    Dn,Dn
    SUBX.s    -(An),-(An)
```

This instruction subtracts both the value of the eXtend flag and the source operand from the destination. The result is stored in the destination.

SWAP Swap Register Halves

Syntax: CCR: NZVC

```
    SWAP      Dn
```

This instruction exchanges the high and low words of a data register.

TAS — Indivisible Test and Set

Syntax: CCR: NZVC

```
    TAS       ea
```

where ea is

```
        Dn        -(An)
                  d16(An)
        (An)      d8(An,Xn)    nnn.W
        (An)+                  nnn.L
```

This instruction tests a byte sized operand, sets the N and Z flag accordingly, and sets bit 7 of the operand to 1.

TRAP — Trap Through the Exception Table

Syntax: CCR: Unchanged

```
    TRAP      #<vector>
```

This instruction pushes the PC, SR and exception table vector onto the stack, in that order, and calls the exception routine. After the TRAP has executed the processor is in supervisor mode.

TRAPV — Trap on Overflow

Syntax: CCR: Unchanged

```
    TRAPV
```

This instruction traps through exception number 7 if the oVerflow flag is set. If the V bit is not set the program executes normally.

TST — Test an Operand

Syntax: CCR: NZVC

```
    TST.s      ea
```

where ea is

```
Dn        -(An)        d16(PC,Xn)
          d16(An)      d8(PC,Xn)
(An)      d8(An,Xn)    nnn.W
(An)+                  nnn.L
```

This instruction tests the operand (by comparing it to zero) and sets the Negative and Zero flags accordingly. The operand is not altered in any way.

UNLK — Unlink and Deallocate Stack

Syntax: CCR: Unchanged

```
    UNLK       An
```

This instruction is the complement of the LINK instruction. This instruction first loads the SP with the value in the given address register. A longword is then pulled from the new stack and stored back into the original address register. (This process removes the frame created by the LINK instruction.)

INDEX

A

Absolute addressing 36, 52
abcd 219
add 63, 219
adda 220
addi 65, 220
addq 48, 221
addx 221
Addresses 8
Addressing modes 36-37
Address registers 33
Address register direct addressing 36, 48
Address register indirect addressing 36
Address register indirect addressing with displacement 36, 122-123
Address register indirect addressing with index 36, 123-124
Address register indirect addressing with post-increment 36, 77-78
Address register indirect addressing with pre-increment 36, 45
AND 88-89, 113
and 222
andi 88, 89, 222-223
Arrays 72-73
ASCII 49, 61, 62, 63-65, 90-92, 93-94
asl 224
asr 224
Assembler directives 15

B

BASIC 1, 2-3
beq 50
Binary numbers 24, 25-26, 27-29
Bit 22
bcc 225
bchg 226
bclr 226
bcs 225
beq 225
bge 149, 225
bgt 62, 225
bhi 225
ble 225
bls 225
blt 62, 225
bmi 64, 225
bne 225
bpl 225
bra 50, 225
bset 227
bsr 73, 227
bss 41-42, 86
btst 227
bvc 225
bvs 225
Buffers 86, 153
Byte 22

C

Carry bit 34
Cconin 35-36, 48-49
Cconrs 95-96
Cconws 47, 63
chk 228
clr 91, 228
cmp 49-50, 62, 229
cmpa 229
cmpi 95, 230
cmpm 230
Comments 15, 80
Comparisons 49-51
Compiled languages 2, 3
Compiling 3
Condition codes 34, 49-50
Conditional assembly 146-147
Constants 42-44, 88, 149
Converting numbers 26-31,
  60-61, 63-65, 90-92, 93-94
Cprnos 149
Cprnout 151-152

D

data 16, 41
Data registers 33
Data register direct addressing
  36, 51
Data sizes 21-23
dbcc 231
dbcs 231
dbeq 76-77, 231
dbf 231
dbge 231
dbgt 231
dbhi 231
dble 231
dbls 231
dblt 231
dbmi 231
dbne 231
dbpl 231
dbra 76-77, 231
dbt 231
dbvc 231
dbvs 231
dc 16, 42, 43, 86-87
dcb 87
divs 232
divu 232
ds 86, 87
Decimal numbers 24, 28-29, 30
DEGAS files 187-188
divu 89

E

Effective address 63
endc 146
endm 111
eor 233
eori 233-234
equ 42
even 72
exg 234
ext 235
Extended bit 34, 35

F

fail 146
Fclose 151
Fetching 9
Fields, instruction 15
Fcreate 167
Fopen 148-149
Fread 150-151
Function calls 35-36
Fwrite 167, 171

G

Getrez 189-190

H

Hexadecimal digits 26
Hexadecimal numbers 24, 26, 27
High byte 23
High level languages 1, 2
High word 23

I

ifc 146, 147
illegal 235
Immediate addressing 36, 45
Immediate values 44
Instruction fields 15
Instruction suffixes 37-38
Interpreted languages 2
Integers 60, 61, 63, 64, 90-92, 93-94
Interpreting numbers 59-60

J

jmp 235
jsr 236

L

Labels 7, 15-16
lea 236
link 237
Least significant byte 23
Local labels 92-93
Logical screen 192, 196-197
Long word 23
Looping 76-77
Low byte 23
Low word 23
lsl 118, 237
lsr 114-115, 237

M

Machine language 1, 3-4
macro 111
Macros 110-112, 146-147
mexit 146
Mnemonic 5
move 38, 44, 238, 239-240
movea 238
movem 75, 79, 241
movep 241
moveq 92, 241
Most significant byte 22
muls 242
mulu 94, 242

N

nbcd 243
neg 243
Negative bit 34, 35
negx 244
nop 244
not 245
null character 42
Number systems 24-26

O

Opcode 15-16
Operand 15-16
or 119-120, 245
ori 246
OR 119
Overflow bit 34

P

Pallette 193-195
pea 166, 247
Physbase 193
Physical screen 192-193, 196-197
Printers 149-150, 151-152
Program counter 8-11, 34, 73-74
Program counter addressing with displacement 36
Program counter addressing with index 36
Programming style 65-66
Pterm0 53

R

Random 88
Random numbers 87-90
Registers 7-8, 33-35, 63, 74-76
reset 247
Restoring registers 79
rol 248
ror 248
roxl 249
roxr 249
rte 249
rtr 250
rts 74, 76, 250

S

Saving registers 63, 74-76
sbcd 250
scc 251
scs 251
seq 251
sf 251
sge 251
sgt 251
shi 251
sle 251
sls 251
slt 251
smi 251
sne 251
spl 251
st 251
stop 252
sub 252
suba 253
subi 253
subq 254
subx 254
svc 251
svs 251
swap 254
Scan codes 49
Screen flipping 195-197
Screen memory 190-193
Screen resolution 189-190, 196
Setcolor 193-195
Setpalette 195
Setscreen 195-197
sp 16, 33
Stack 11-14, 35-36, 46-47, 48, 73-74, 75
Stack pointer 12, 33, 45, 46-47, 48, 74
Status register 34-35
Status register addressing 36
Strings 42-43, 51-52, 72-73, 86
String arrays 72-73
Structured programming 79
subi 63, 64
Subroutines 73-76, 79-80
swap 90

T

Tables 72-73, 124-125
tas 255
text 15, 41
Tgetdate 112-113

Tgettime 116
Top-down programming 168-169
trap 35-36, 47, 88, 255
trapv 255
Tsetdate 121
Tsettime 121
tst 91

U

Uninitialized data 86-87
unlk 256

V

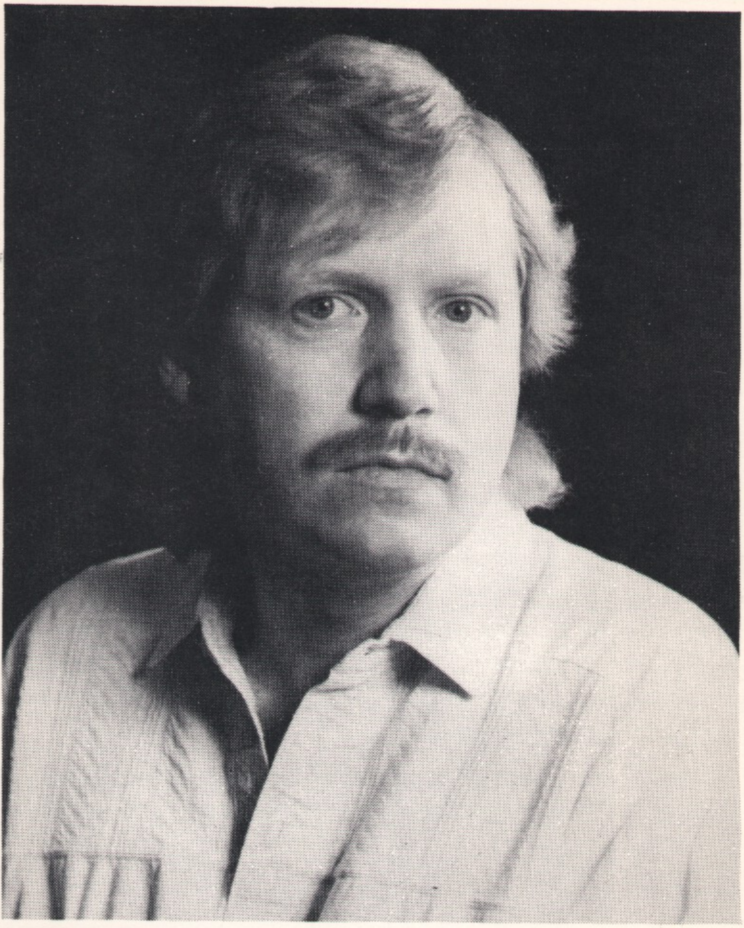
Verifying input 62

W

Word 22

Z

Zero bit 34



Clayton Walnum, the former editor of *ST-Log* and *ANALOG Computing*, has been programming Atari computers for a decade. In addition, he has sold more than 300 articles to such magazines as *The Writer*, *Atari Explorer*, *Compute*, *VideoGames & Computer Entertainment*, and, of course, *ST-Log* and *ANALOG Computing*. Now a full-time, free-lance writer, he has published several books, including *Beyond the Nintendo Masters* (Hayden Books), *Master Populous* (Sams), *The First Book of Works for Windows* (Sams), and *C-manship Complete* (Taylor Ridge Books). Mr. Walnum lives in Connecticut, with his wife, Lynn, and their three children, Christopher, Justin, and Stephen.

The ST Assembly Language Workshop

Clayton Walnum's first programming book, *C-manship Complete*, was praised as one of the best books of its kind. A complete tutorial in both the basics of C and advanced GEM programming, *C-manship* taught thousands of ST programmers how to make their computers jump through the hoops. Now, Taylor Ridge Books is proud to present Mr. Walnum's newest work, *The ST Assembly Language Workshop, Volume 1*, the first volume of a three-volume set.

In *The ST Assembly Language Workshop, Volume 1*, you will learn the basics of programming in 68000 assembly language, everything you need to know to produce full-length programs on your Atari ST. You'll learn the most commonly used 68000 instructions, as well as how to handle disk files, printers, graphics, and more. Also included in the book is a quick reference to the complete 68000 instruction set, providing you with the information you need to explore the language more fully.

If you want to tackle assembly language programming on your ST, let *The ST Assembly Language Workshop* be your guide.

Praise for C-manship Complete

"Within its genre, it has all the earmarks of a classic text."

--John Jainschigg, Atari Explorer

"C-manship is a terrific way to learn the C language in general and ST programming in particular. Highly recommended!"

--Charles F. Johnson, CodeHead Software

"For learning C programming and the VDI, AES and GEM libraries on the Atari ST, this book cannot be matched."

--Stephen D. Eitelman, Current Notes

TRB

Taylor Ridge Books
P.O. Box 78
Manchester, Connecticut 06045
(203) 643-9673

U.S. $24.95
Includes Disk!