

Paul Overaa

A miiga assembler

INSIDER
guide
SERIES



The complete beginners guide for all Amigas

Amiga Assembler Insider Guide

Amiga Assembler Insider Guide

**An Introduction to 68000
Assembly Language Programming
on the Amiga**

Paul Overaa



Bruce Smith Books

© Paul Overaa 1993
ISBN: 1-873308-27-2
First Edition: October 1993

Editor: Peter Fitzpatrick
Typesetting: Bruce Smith Books Ltd

Workbench, Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc. UNIX is a trademark of AT&T. MS-DOS is a trademark of Microsoft Corporation. All other Trademarks and Registered Trademarks used are hereby acknowledged.

All rights reserved. No part of this publication may be reproduced or translated in any form, by any means, mechanical, electronic or otherwise, without the prior written consent of the copyright holder(s).

Disclaimer: While every effort has been made to ensure that the information in this publication (and any programs and software) is correct and accurate, the Publisher can accept no liability for any consequential loss or damage, however caused, arising as a result of using the information printed in this book.

E&OE

The right of Paul Overaa to be identified as the Author of the Work has been asserted by him in accordance with the *Copyright, Designs and Patents Act, 1988*.

Bruce Smith Books is an imprint of Bruce Smith Books Limited.
Published by: Bruce Smith Books Limited. PO Box 382, St. Albans, Herts, AL2 3JD. Telephone: (0923) 894355 – Fax: (0923) 894366.
Registered in England No. 2695164.
Registered Office: 51 Quarry Street, Guildford, Surrey, GU1 3UA.

Printed and bound in the UK by Ashford Colour Press, Gosport.

The Author

Paul Overaa initially qualified as an analytical chemist and spent two decades working in a field of physical chemistry known as gas-liquid chromatography. It was during this time that he became heavily involved with computerised data reduction techniques and computer programming. Nowadays he considers himself a programmer first and an analytical chemist second.

Paul has previously written books on low-level 6502 and Z80 assembly language programming, on Amiga programming in C and ARexx, on Amiga systems programming and on both Commodore Amiga and Atari ST program design. He is a proficient ARexx, C and 68000 assembly language programmer, and a very experienced Amiga programmer whose technical expertise is frequently used by a great many computer magazines including *Amiga Shopper*, *Amiga Format*, *Amiga User International*, *Amiga Computing*, *Program Now*, *Computing*, the *Amiga Buyer's Guide* and *Atari ST User*. In addition to this he provides expertise on MIDI programming for magazines such as *Sound on Sound* and *International Musician*. In the past he has written for many other publications including *ST World*, *Personal Computer World*, *Practical Computing*, *Laboratory Practice*, and the one time highly influential *Transactor Amiga* magazine.

Outside interests include Yoga, mathematics and badminton but his main passion nowadays is computer programming with his research interests having a strong bias towards the practical use of the Warnier diagram and other language-independent program design techniques.

	Preface	13
1	Setting the Scene	15
	Chips, Chips And More Chips	16
2	Creating a Program	19
3	A Model of the 68000 chip	25
	MicroProcessor Registers.....	27
	The Program Counter	28
	The 68000's Status Register	28
4	Addressing Modes & Instructions ...	33
	68000 Instruction Classes	34
	Data Movement.....	34
	Arithmetic and Logic Instructions.....	35
	Flow Control	35
	Other Instructions	36
5	Assemblers	39
	Comments	40
	Labels	40
	Label conventions.....	41
	Assembler Directives	42
	The EQU Equate Directive	42
	Storage Allocation Directives	42
	Operands and Addresses	43
	Macro Assembly.....	44
	Conditional Assembly.....	44
	If You're Having Trouble.....	44
6	Safety In Numbers	47
	Practising Safe Hex	48

7	Making a Start	53
	Data Transfer	54
	Data Transfer Using Address Registers	60
	Complementing a Value	61
	Addition.....	62
	Putting Some Pieces Together	63
	Quick Instructions	65
8	Amiga Libraries	67
	Run-Time Libraries	68
	Opening a Library.....	68
	A Sneaky Exec Trick.....	71
	Making a Library Call.....	72
	Library Vector Offset (LVO) Values	73
	Closing a Library	74
	Putting It All Together.....	74
9	Using A68k and Blink	79
	Step One – Opening a Shell Window.....	81
	Step Two – Creating the Source File	82
	Step Three – Assembling the Example Code.....	83
	Step Four – Linking.....	84
	Step Five – Preparing for the Worst	84
	Step Six – Go Go Go	84
	If Things Have Gone Wrong	85
	<i>End statement is missing</i> Error	85
	Undefined Symbol Errors	86
	Error In Operand Format	86
	Linking Errors	86
	Program Fails To Run As Expected.....	86
10	Devpac	89
	An Integrated Environment	89
	Step One – Starting Devpac	91
	Step Two – Creating the Source File	91
	Step Three – Assembling & Linking the Example Code...	93
	Step Four – Preparing for the Worst	94
	Step Five – Go Go Go	94
	If Things Have Gone Wrong	95

11 Macro Magic	97
Macro Definitions	98
The LINKLIB Macro	98
Macros Within Macros	101
The Inderlying Magic	101
Header Files	103
Asking the Assembler to Include Another File.....	103
12 DOS and the Shell.....	105
Writing Text	108
A First Coding Stage.....	109
Write()-ing The Message	110
Variety Is The Spice Of Life	112
13 System Include Files	117
The Snag For A68k Users	119
Why This Book has Avoided the System Includes.....	120
14 More on Intuition	123
Ringing the Changes.....	123
When New Windows Are Not!	124
Window Opening	125
Tag Lists	126
Open Sesame.....	128
Completing the Plan of Action.....	130
Adding the Screen Locking/Unlocking Code.....	131
Adding Some Tag Data.....	137
15 Amiga Graphics: A Start.....	143
Getting Graphics into Code.....	145
A Runnable Example	146
If You Haven't Got the Official Includes	148
The Official Approach	153
16 More Coding Practice.....	157
Subroutines.....	158
Subroutine Parameters	159
An Example Subroutine	159

Drawing a Row of Images162
Building a Test Framework.....167
The Official Alternative173
Reaping the Benefits178

17 Where To Go From Here 185

Appendices

A The 68000 Instruction Set..... 189

Effective Address190
Op-Codes190
Sign Extension190
Notes on An/Dn Name Conventions190
60000 Addressing Modes.....190
Data Movement Instructions194
Flow Control Instructions199
Logical Operations203
Shift and Rotate Operations207
Bit Manipulation Instructions209
Arithmetic Instructions210
To Get the Complete Story.....216

B Library Function Tables 217

Usage notes219

C Glossary Of Terms 221

D Bibliography 231

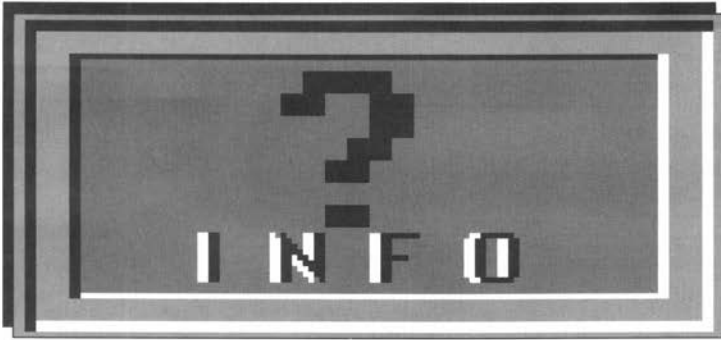
E Books for the Amiga 235

Index 249

Insider Guides

#1:	Wordprocessor ASCII Files	21
#2:	Program Crashes.....	22
#3:	Assemblers.....	23
#4:	Assemblers.....	24
#5:	Flags and Flag Bits.....	29
#6:	How a Computer Works	30
#7:	The 68000 Chip Itself.....	31
#8:	Branches and Jumps.....	36
#9:	Subroutines	37
#10:	ASCII CODES.....	43
#11:	Program Layout	45
#12:	Truth Tables	51
#13:	Additional Info	56
#14:	Word Storage	57
#15:	Long Word Storage.....	58
#16:	Where have all the functions gone?	69
#17:	Failed Open Library Calls.....	70
#18:	An Important Exec Function.....	70
#19:	Another Exec Masterpiece.....	71
#20:	The Importance of Being a6.....	72
#21:	A Below Average Score?	73
#22:	A Beeping Good Routine	74
#23:	Collecting The Standard Input Handle.....	106
#24:	Collecting The Standard Output Handle	106
#25:	Another Useful DOS Function	107
#26:	More Macro Help.....	113
#27:	Progress Report.....	115
#28:	System File Updates.....	119
#29:	Keeping Up To Date?	121
#30:	Window Opening – The Bottom Line	126
#31:	Lock Em Up!	131
#32:	Free At Last!	132
#33:	Open Up!.....	132
#34:	Closing Windows When You're Finished	133
#35:	A DOS Time Waster	133
#36:	If You Have The Official Amiga Include Files.....	138
#37:	Graphics The Easy Way.....	144
#38:	BitPlane Graphics Theory	145
#39:	Devpac to A68k Assembler Section Conventions....	147

The Amiga is an incredible computer already but, by learning how to program it using 680x0 Assembly Language, you can unleash some amazing additional power...



You've either bought this book already, or are giving it the once over in order to see whether it's likely to be useful to you or not. If you fall into the first category then thanks, I'm sure you won't be disappointed. For those of you in the second category I'll try and explain the general plan of the book in order that you can decide whether this particular offering is suitable for you or not.

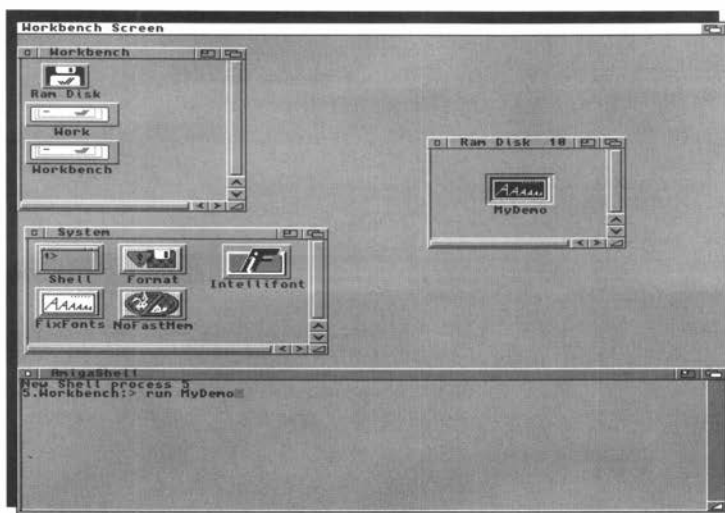
Firstly let me reassure you about one thing – learning 680x0 assembly language is not that difficult a task. There are in fact a great many books available that provide very good introductions to this subject. So why did I bother to write another? It's because many books, although perfectly well written, are about 680x0 coding in general rather than 680x0 coding on the Amiga. The difference is important because the 680x0 chips used in the Amiga do not work in isolation – they're just a small part of a system which involves a fairly complex surrounding shell of operating system software. If you are intent on programming the Amiga using assembly language then a lit-

tle knowledge about this operating system is needed early on. It is often the lack of this sort of Amiga specific material that provides a major stumbling block for potential 68k coders. It is just this sort of Amiga-specific info that I've made a point of providing in this Insider Guide.

The material in this book is up-to-date and this again is important because the Amiga's operating system has undergone quite a few changes in recent years as Workbenches 2 and 3 have arrived. Where relevant I've made a point of dealing with the changes in detail and you will for instance find explanations on the use of Intuition and the new Tag-List based function calls towards the end of the book.

I've attempted to introduce 680x0 assembly language specifically from an Amiga oriented viewpoint and my main aim has been to provide you with the necessary footholds to get into low-level Amiga programming as quickly as possible. I believe that I can show you a simple pathway to achieve this objective and even make the subject enjoyable. That, believe me, is over half of the battle. This book will not, by any stretch of the imagination, make you a 68k expert but it will get you started and prepare you for the things you'll read about when you feel ready to tackle more advanced Amiga books.

Many serious Amiga owners, including lots of new Workbench 2 and Workbench 3 based A600, A1200, and A4000/030 owners, are clamouring to learn the computer language that expert coders use. It's called 68000 Assembly Language and to start learning about it all you have to do is read on...



The heart of the Amiga is a silicon chip called the central processing unit or CPU. Although various Amiga models use different processors from the Motorola 680x0 family the basic device from this family is the unit known as the 68000 microprocessor. Later chips, such as the 68030 and 68040, are more powerful but since the 68000 provides a similar set of core facilities to those found in other members it is this chip that I'll refer to in this book.

To program the 68000 you use something called assembly language and these early chapters provide an overview of what assembly language programming is all about. I've made a point of trying to avoid all of the difficult stuff you usually find in assembly language books but there is bound to be the odd

topic that just doesn't appear to make sense at first. Don't get disheartened if you suddenly find you don't understand something, and don't for a moment think that you are expected to memorise everything in one sitting!

Certainly try to grasp things as you encounter them but when you do hit a topic that seems impossibly difficult to grasp then skip over it and make a note to return to it later on. You may actually find that areas, such as the 68000's addressing modes discussed in Chapter Four, make more sense once you've tackled a few of the small programs given in later chapters.

As you get more comfortable with the general ideas, refer back to the early chapters, as you'll get a little more from them each time you do. Work at your own pace and remember that no-one, and I really mean no-one, has ever learnt 68000 assembly language in a day. Take your time, enjoy the journey and, by the end of the book, you should be ready to tackle the more advanced texts that are available.

Chips, Chips And More Chips

It's the Motorola 68000 chip itself that is our main

concern. Like other microprocessors the 68000 has various hardware lines for communicating with the outside world along with a set of internal registers for storing data. These communications links are used to connect the 68000 to other components and very important they are too – before it can do any useful work the 68000 has to be connected to additional memory in order to provide additional data and program storage space. Two basic types of memory chips, called RAM and ROM chips, are in common use: RAM (Random Access Memory) chips can both be written to and read from by the microprocessor but have the disadvantage of losing their contents when the power is turned off. ROM (Read Only Memory) chips can keep their contents indefinitely, whether powered up or not, but they have to be pre-programmed with data and their contents cannot be changed.

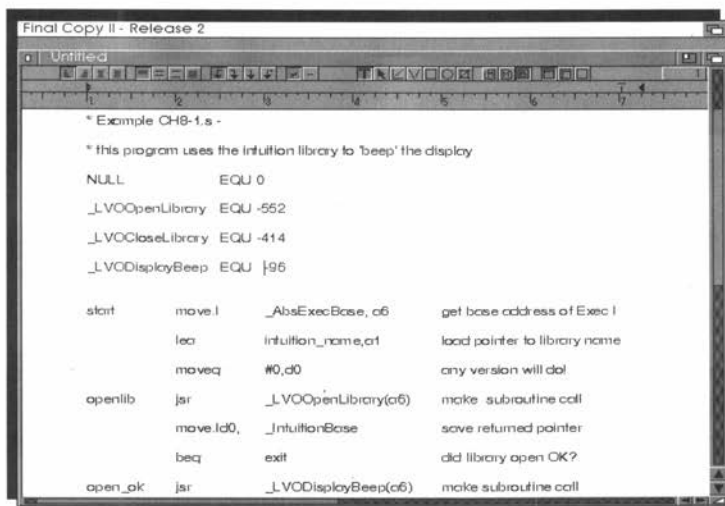
Because of the way it's been designed the 68000 microprocessor is able to perform various logical and arithmetic tasks and these are identified by numbers that represent particular microprocessor *instructions*. Computer programs are simply collections of these instructions put together in a meaningful way. One of the things that both RAM and ROM chips are used for is storing these programs whilst they are being executed, ie run, by the 68000 chip.

The *instruction sets* (the collection of available instructions) of most processors, including the 68000, are quite large but there is nothing inherently complex about the operations they perform. Each instruction carries out some elementary task, adding two values together perhaps or copying the contents of one memory location to another, but there is a minor snag – the language that the microprocessor understands is based on binary numbers. Given suitable hardware (a processor chip, memory, some input/output facilities, and all the associated electronic support) such a system could be programmed by entering suitable numbers directly into system memory and then getting the processor to execute the instructions. Trying to program a 68000 chip using the raw numbers in this way turns out to be a nightmare so, instead, Motorola give the instructions standardised and meaningful names such as ADD and MOVE. This makes it easier for programmers to remember the purpose of these instructions whose names are known as *mnemonics*. The process of converting mnemonics back into those numbers which represent real processor instructions is something that the computer itself can do quite easily and programs which do this are known as *assemblers*. The mnemonic form of the 68000 instructions are known as the *68000 Assembly Language* and it is precisely these instructions that you'll be learning about in this book.

Since the Amiga is a 680x0 based machine, it's pretty obvious that all Amiga languages end up generating 680x0 code. They have to because otherwise the final programs simply wouldn't be able to run on the Amiga's microprocessor. You may be wondering what it is then that makes code written by assembler programmers run faster than the equivalent 68000 code generated by other high-level languages. The answer is simply that assembler programmer's can fine tune their code to make sure that it is super-efficient and, for a number of reasons, high-level languages are unable to do this to the same degree.

Note: I've used the term 680x0 above quite a lot and if the meaning isn't clear it is simply a global way of referring to the family of 68000 chips, namely the 68020, 68030 and 68040.

Writing in 68000 Assembler requires a number of different preliminary stages before a runnable program can be produced. You can't just type in a program and then select "Run"...



The first step in writing an assembly language program is to use an editor to prepare something known as a *source code* file. It sounds good, but all it really means is that you have to produce a plain text file (commonly called an ASCII text file) which contains the required program instructions. You can list and print the contents of such a file just as you would a letter or any other piece of stored text.

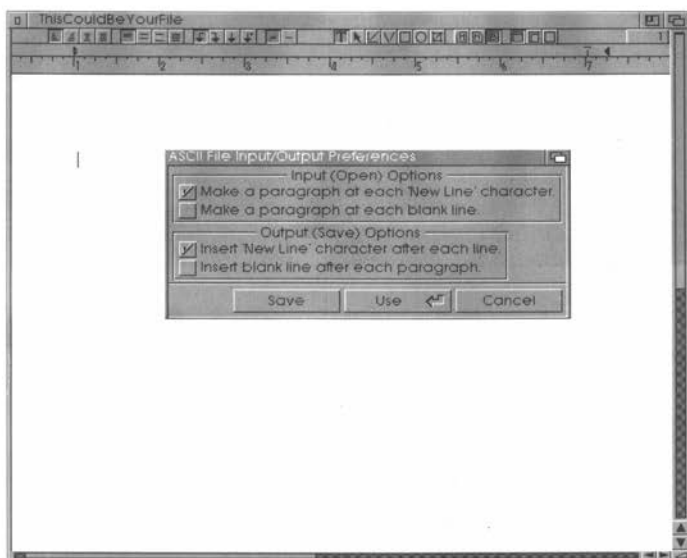
The Amiga provides one editor, called ED, as part of the Workbench system software but most commercial assemblers come with their own rather better editor programs. If preferred you can use an alternative editor or even your favourite word-processor program. The only proviso with the latter option is that it must be possible to stop the wordprocessor from inserting additional control characters because these are meaningless to an assembler and would cause it to come to a grinding halt if it tried to interpret them.

Once a source file is available the next step is to get the assembler program to convert it to the appropriate 680x0 instructions. In many cases the assembler has to be used first to create a standardised intermediate form known as an *object code* file. This is not a runnable program as such. Although the object file includes the translated 680x0 instruction-related material the code is not of the right format to be loaded by AmigaDOS. The program doesn't contain a sometimes important piece of Amiga specific front-end code known as the *start-up module* which is needed if the program is to be run from the Workbench. Finally, the file may still contain references to unresolved (unknown) items, such as library routines or variables that have been specified as being present in other object code modules. The third stage in producing an assembly language is known as *linking*, and it attempts to fill in the gaps created by these unresolved references. The Amiga linker (usually called *Blink*) is able to combine the code you have written, the start-up code, and any other modules or library code required, to produce a real Amiga program – ie a file that may be loaded and run!

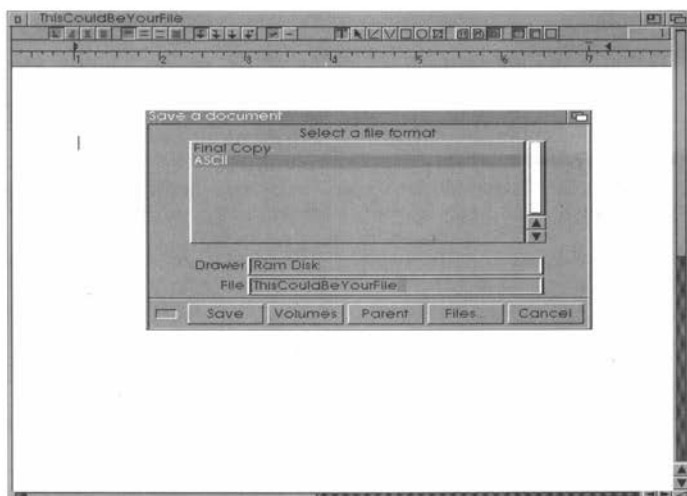
On occasion things may not go well and you may find that, as the assembler attempts to translate your source file, it reports any number of errors. Whatever the cause (syntax errors, illegal instructions etc) these faults have to be corrected and this can mean that, in the early days, you have to pass through the edit-assemble cycle quite a few times before you succeed in creating a program that even assembles successfully. Once through that stage you may then find that the linker reports additional errors such as misspelling a library routine name or not specifying the correct location of library files. These errors must also be found and eliminated before a runnable version of the program can be created.

As you might guess, there is no guarantee, even once a program is up and running, that it is free from errors. Assembly language programmers are, unless they are very careful, likely to spend far more time looking for hidden errors – commonly called *bugs* – than their high-level language counterparts. Many programmers frequently use a piece of software called a *debugger* – a system tool that is able to execute a program on a step-by-step basis – in order to help them to trace program execution and identify faults. It's worth mentioning that debugging tools are by no means essential because there are plenty of other ways of locating program bugs.

Insider Guide #1 – Wordprocessor ASCII Files

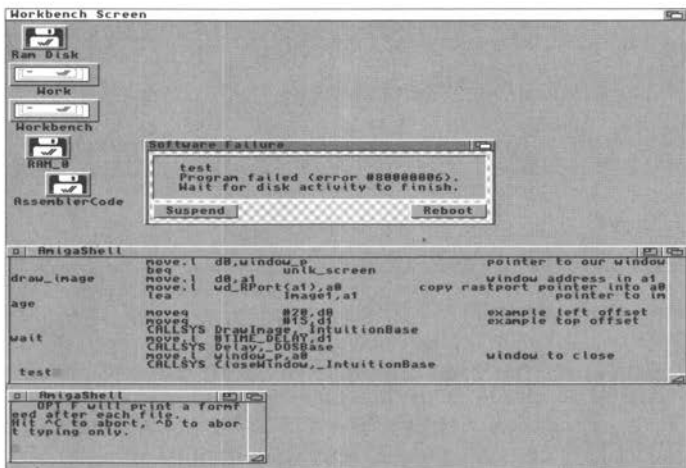


Almost all wordprocessors nowadays provide options for writing plain ASCII text files but, since this will not be the default text file format, it's up to you to explicitly select this type of file output. With Final Copy II, for instance, you have to use the Project Menu's 'Save As' Option and select ASCII using the file requester's Export gadget.



Insider Guide #2 – Program Crashes

When you execute an assembly language program that contains one or more bugs it may, if you are lucky, just fail to work as expected but otherwise be relatively harmless as far as your Amiga's operating system (O/S) is concerned. More often however, seemingly small errors can cause your Amiga to display a Guru message or even seize up completely so that you have to re-boot. The danger here is that, if you were not prepared for such an eventuality, you may have unsaved files present in your Ram Disk (most program editing and assembly operations are done in Ram because it is quicker).

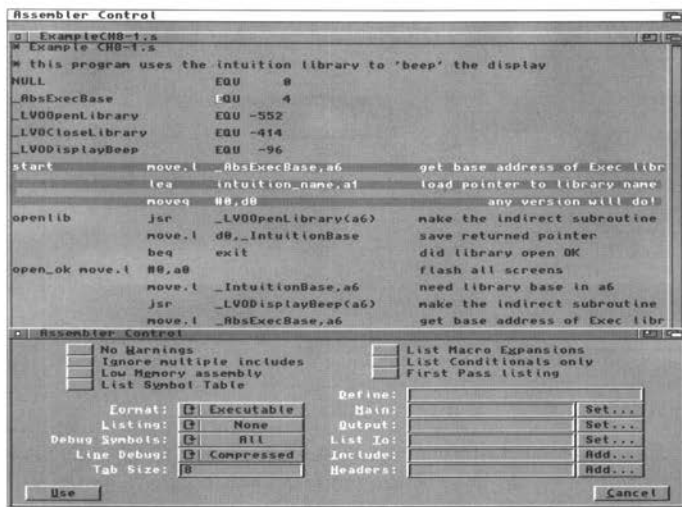


Sometimes the Amiga's O/S can stop the faulty program from running and let you recover Ram Disk files by switching to a working Shell window and saving any important files. At other times the faulty code might have overwritten important portions of the Amiga's O/S causing your Amiga to re-boot without warning. Because of this you should get into the habit of saving your source files frequently just in case something does go wrong when you run a newly created, or recently modified, program for the first time.

Insider Guide #3 – Assemblers

Two of the most popular Amiga assemblers are HiSoft's Devpac and Charlie Gibb's A68k. Devpac is a commercial offering which has been around for a long time and there is no doubt at all that it's a very well supported program. It comes with an easy-to-use Intuition styled front end, lots of extra utilities, and all of the necessary Amiga system files.

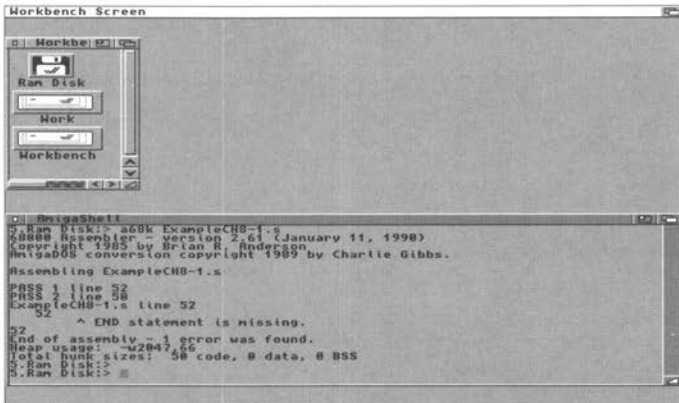
Devpac 3, the latest version, is very highly thought of and used by a great many professional programmers.



One of the most popular Amiga assemblers is the HiSoft Devpac package

Insider Guide #4 – Assemblers

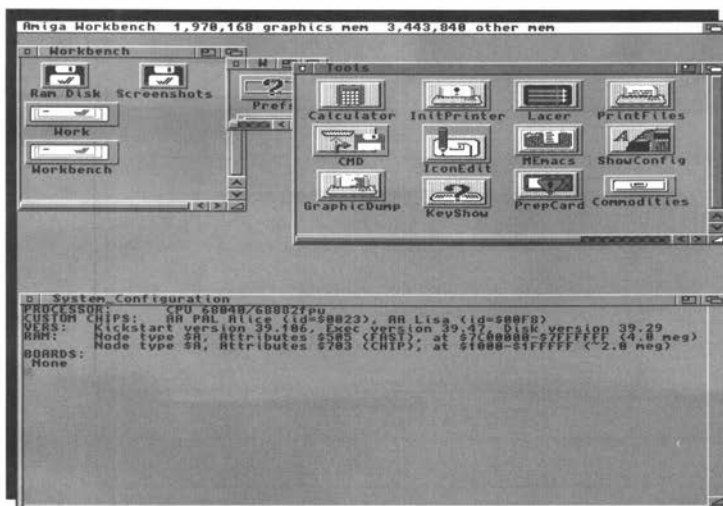
A68k is a freely distributable public domain assembler available from most PD libraries. It is an extremely well programmed piece of software although not having an Intuition style front end – it requires Shell based commands instead – and it's not as easy to use as Devpac.



Charlie Gibb's A68k public domain assembler is also perfectly adequate.

For convenience, A68k has been included, along with the Blink linker, on the accompanying Insider Guide disk that is obtainable free from the publishers of this book – see Appendix E. The disk therefore contains everything you need to assemble and run all of the examples in this book. You'll find more details about actually using assemblers and linkers later.

Learning about the microprocessor hardware and electronics is a nightmare for the newcomer but this chapter provides an easy way around the problems...



To write assembly language programs all you need is a simple conceptual model of the processor so, initially, there's no need to appreciate either the hardware or the associated electronics. However, it is important to get an understanding of the general characteristics of the 68000 – such as what sort of bits of information (data) it can store internally – and of some restrictions imposed by the overall design of the chip.

Figure 3.1 shows a schematic diagram of a 68000 processor. What I'll be doing for the rest of this chapter is building a conceptual model of the 68000 chip, a simplified picture of the chip and its facilities, and this will allow me to discuss the features which are relevant to the writing of assembly language programs without having to get involved with the rather awkward hardware issues.

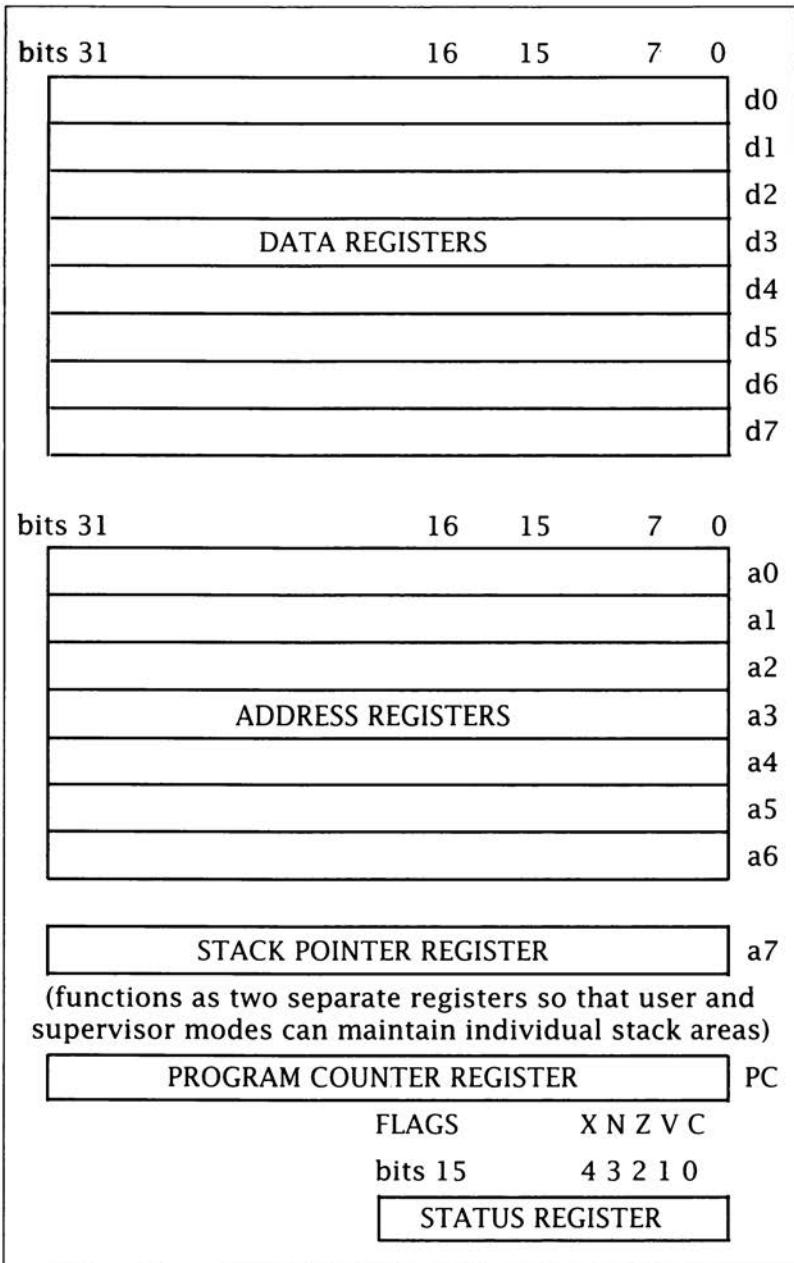


Figure 3.1. A schematic model of the Motorola 68000 microprocessor.

MicroProcessor Registers

The 68000's internal registers are split into two groups known as address registers and data registers respectively. Registers of each group are numbered from 0 to 7 with data registers being labelled as d0, d1, d2... d7 (or D1, D2... etc.), and address registers as a0 (or A0) and so forth. Each 68000 register can hold a four byte (32 bit) number and amongst its other facilities the processor is able to move such numbers between its internal registers or between a register and a memory location (and vice versa). The 68000 can also move external data held in memory from one location to another.

Address register a7 has a special purpose in that it serves as the micro-processor's *stack register*. This holds the address of, or *points to*, an area of memory used to store and retrieve information on a last-in-first-out basis. The 68000 uses this area to store things like subroutine addresses which we'll discuss later.

There are in fact two different 68000 stack pointers and this stems from the fact that the processor can operate in two modes known as *user mode* and *supervisor mode*. In some situations it is convenient for each mode to have its own stack and so the 68000 was designed so that register a7 behaved like two separate registers. We're not going to be involved with these mode related issues at all and, for the purposes of this book, just regard register a7 as a single register holding a single stack pointer.

One of the nicest features of the 68000 is the flexibility of its registers. Although they can hold 32 bit (long word) values the chip can for many operations use the address registers to work with 16 bit values (words) and the data registers can in fact work with 32 bit, 16 bit or 8 bit values. Similarly there are few restrictions on what you can, or cannot, use the contents of such registers for. If, for instance, you wish to copy the contents of a data register into an address register the 68000 lets you do it although having said that it is usually better to use address registers for storing and working with memory addresses and data registers for data oriented operations because each group is best suited to its design-chosen purpose.

When working with instructions that may involve byte, word or long word values it is often necessary for the assembly language programmer to identify the size that should be assigned to a given value. As you'll see later the 68000 conventions are based on placing .b, .w or .l after the instructions. The 68000, because of its design does however

have a limitation in that when accessing word or long word addresses the address must be even. These even addresses are conventionally said to be *word aligned* but the good news here is that assemblers take care of much of the word-alignment problems automatically.

The Program Counter

The 68000 also contains a 32 bit program counter which is a register used by the microprocessor to determine the address of the next instruction to be executed. Under normal conditions the program counter is automatically *incremented* as instructions are read and acted upon, hence instructions contained in memory are executed in sequence, ie one after another. An important part of microprocessor programming, however, involves the use of a number of instructions which can alter the contents of the program counter and the result of doing this has far reaching implications. By changing the address held in the program counter it is possible to cause the microprocessor to get its next instruction from anywhere in memory, as opposed to getting the instruction sequentially next in memory. The result is that the execution of the program can *jump*, or *branch*, from one part of the program to another.

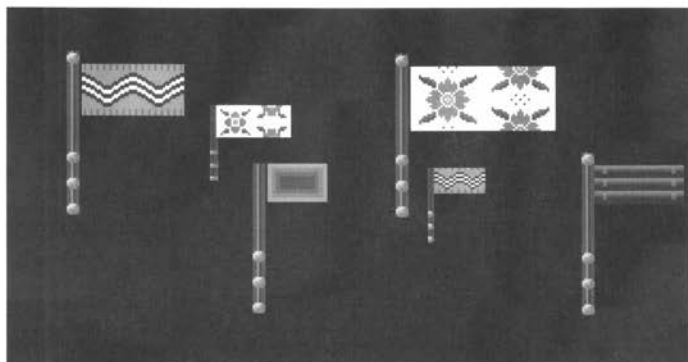
The fact that these jumps can be made conditional on the state of various processor flags means of course that the processor can make intelligent flow control decisions based on the data with which it is working. A program might for instance compare two numbers and, on the basis of the result, execute (or perhaps not execute) a particular group of instructions.

The 68000's Status Register

Another important 68000 register is the status register which is actually divided into two eight bit registers known as the *system byte* and the *user byte*. We won't be concerned with the system byte as it is only accessible in supervisor mode. The user byte, on the other hand, is going to be important because it contains flag bits whose values are set and cleared according to the results of particular instructions.

Five flags – out of a possible total of eight – have been implemented in the 68000's user byte and these provide single bit true/false type detection of the processor conditions known as carry (C), overflow (V), zero (Z), negative (N), and extend (X). The carry bit holds the carry from

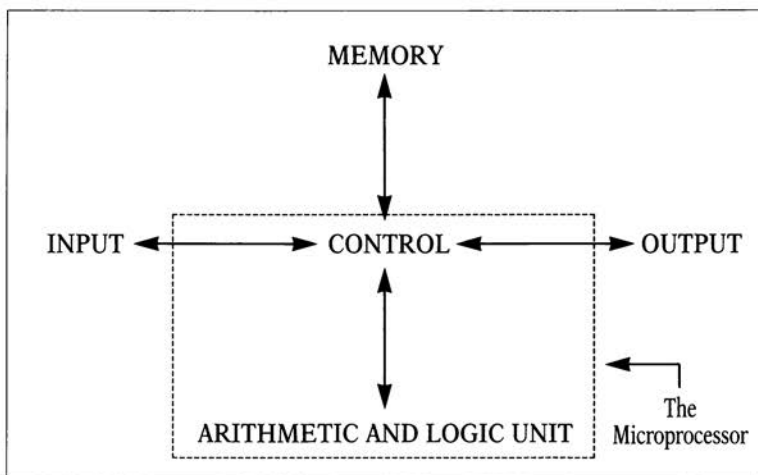
Insider Guide #5 – Flags and Flag Bits



In the computing world, flags are simply those bits present in a variable, or hardware register, which have been assigned some specific meaning. The term is normally reserved for yes/no (true/false) type indicators which only require a single bit of storage space. A byte-sized hardware register, since it is a register containing 8 bits, can therefore act as a store for up to eight different flag values. By convention if a flag bit has the value 1, then it is said to be set (or true), and if the bit has the value 0 it is said to be clear (or false).

the most significant bit produced by bit shifting or arithmetic operations. The zero flag is set high (ie set to 1) when an operation produces a zero result. If, for example, the result of adding two numbers together produced a zero then the 68000's zero flag would be set to 1. The negative bit (sometimes called the sign bit) always takes the value of the most significant bit of the result and, along with the overflow and extend flags, is primarily used for arithmetic applications. Not all instructions affect all flags as you'll see when we look at typical instructions.

Insider Guide #6 – How a Computer works

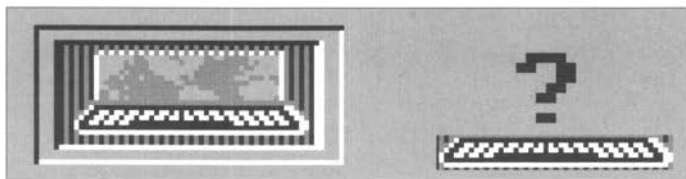


Above: Block diagram of a computer

A computer consists essentially of a microprocessor (containing a control unit and an arithmetic/logic unit) coupled to memory chips and various input/output devices such as a keyboard, VDU screen, disk drives, and printer. The chip is built in such a way so that it can understand a particular set of simple instructions known as the instruction set. It is able to be programmed to perform different tasks by providing it with different instructions. Computer programs consist of a list of such instructions arranged in a suitable order for the task being carried out. Programs are stored in memory and are executed by the microprocessor using a loop consisting of four basic steps:

- 1 Fetch the next instruction from memory and place it in the control unit.
- 2 Decode the instruction (ie figure out what must be done).
- 3 Obey the instruction.
- 4 Go to step 1

The first two steps are called the fetch cycle, the second two are known as the execution cycle.



Insider Guide #7 – The 68000 Chip Itself

The Motorola 68000 is one of a family of '68' processors ranging from an eight bit oriented 68008 to a fairly recently announced super chip called the 68060. All the processors are essentially object code compatible, which means that they execute the same base level instructions, although chips higher up the family – like the 68020, 68030, 68 040 and so on – all have more powerful instruction sets than the basic 68000. As far as the physical details of the 68000 chip itself is concerned the logical layout of the pins looks like this:

Clock	CLK	a1-a23	Address Bus
		d0-d15	Data bus
Reset	RESET		
Halt	HALT		
Processor Status	FC0	AS	Asynchronous data bus control
	FC1		
	FC2	UDS	
		LDS	
		R/W	
		DTACK	
Interrupt Inputs	IPLO		
	IPL1		
	IPL2		
Bus arbitration control		VMA	Synchronous data bus control
Bus error	BR	E	
		VPA	
	BG		
	BGACK		
	BERR		

Above: The Motorola 68000 processor

An external clock signal causes the 68000 microprocessor to step through its fetch/execute cycle at a specified rate. The processor collects data from memory, and stores data in memory using pins d0-d15 which are connected to a common electronic pathway called a bus (pins a1-a23 are used to provide address information for the 68000 chip). The remainder of the pins are power and control signals – the R/W line for instance informs the memory chip whether the processor is doing a read or a write operation. For more details of the electronics involved you'll need to consult a 68000 hardware reference book.

The 68000 instruction set is powerful and a variety of addressing modes are available for many commands. There's a lot of new jargon to be learnt but this chapter should put you straight as far as the basic ideas are concerned...

```

Workbench Screen
-----
Run Disk
Work
Workbench
ASSEMBLER

M68000 Assembler
-----
expunge MOVE.L    D2/A5-A6, -(A7)
MOVE.L    A6, A5
MOVE.L    #16, EXECBase(A5), A6
TST.W    LIB_OPENCNT(A5)
BEQ      #LIB_DELEXP, m1b_Flags(A5)
BSET     #0, D0
MOVEQ    #0, D0
ERR.S    L2

E1 MOVE.L    m1b_SeaList(A5), D2
MOVE.L    A2, A1
JSR      _L00Remove(A6)
MOVE.L    A2, A1
MOVEQ    #0, D1
MOVEQ    #0, D0
MOVE.W    LIB_REGSIZE(A5), D1
MOVE.W    LIB_POSSIZE(A5), D0
ADD.L    D1, A1
JSR      _L00FreeMem(A6)

E2 MOVE.L    D2, D0
MOVE.L    (A7)+, D2/A5-A6
RTS

extfunc MOVEQ    #0, D0
RTS

; following test function increments the value in register a0
;         ULONG = mvfunc(ULONG x)
; registers    d0          a0

```

Most processor instructions work on a piece of data called the *operand* and this data has to be stored somewhere. Many instructions use some real or implied source address, do something, and then transfer the result to its destination address. It is the processor's addressing modes which enable these source and destination addresses to be specified. With the 68000 there are eleven basic addressing schemes and for completeness here are the names... Inherent, Register, Immediate, Absolute, Address Register Indirect, Address Register Indirect with Displacement, Address Register Indirect with Postincrement, Address Register Indirect with Predecrement, Address Register Indirect with Index and Displacement, Program Counter Relative with Displacement, and Program Counter Relative with Index and Displacement. Now, I'm not going to explain all of these addressing modes in

detail because you are unlikely to need more than a few of them during your early coding days. For the moment, then, here are just a few brief descriptions to set the scene.

Inherent addressing means that the instruction itself implies the location of the data it is going to work with. Register addressing implies that the operand resides in one of the 68000's internal registers. Absolute addressing means that the address of the operand is stored just after the instruction in memory whereas Immediate addressing implies that the operand value itself (not its address) is located just after the instruction in memory.

Indirect addressing is a very powerful concept and on the 68000 a variant called register indirect addressing is used. In short an address register is used to specify the address of the operand. In addition to these straightforward addressing modes it is possible to specify displacements, to auto-increment or auto-decrement an address by 1, 2, or 4 bytes and write something called program counter relative code. Later on, when I do need to provide more details about certain addressing modes, I'll do it within the context of some example code because this makes the ideas easier to absorb.

68000 Instruction Classes

The 68000 instruction set, as we've mentioned, is reasonably large and because almost all sensible addressing modes can be used with any instruction the full number of variations available is actually quite substantial. As was the case with the 68000's addressing modes it is not a useful exercise, either now or later, to list or discuss each instruction as the basic details alone would fill a complete book by themselves. Luckily all we need to start with is a general understanding of the types of things the 68000 can do. So, before we start looking at actual programs, the following sections provide suitably brief overviews of the type of instructions available.

Data Movement

The 68000 has a large number of instructions which allow the transfer of data to and from memory and/or the 68000 microprocessor's internal registers. For example, the instruction

```
move.b    d0,d4
```

transfers the lower eight bits of data from register d0 to register d4. This is an example of register addressing. On the other hand

move.l #0,d1

places a zero value in register d1. The hash # sign indicates an operand source addressing mode known as Immediate addressing – in terms of the final 68000 instruction this means that the operand (in this case a long word, ie 32 bit, zero value) is stored immediately after the move.l instruction code.

Data can also be moved to memory locations so, to move the full 32 bit contents of register d0 to a memory location which has been given the symbolic name `_DOSBase` you would use this instruction

move.l d0,_DOSBase

Arithmetic and Logic Instructions

The
68000
supports

a standard set of logic and arithmetic operations which allow it to perform addition, subtraction, multiplication and division. In addition to this it also supports all of the common logic operations (such as AND and OR etc.) As an example, the instruction

add.l d0,d1

adds the full (32 bit) contents of data register d0 to those of data register d1.

Flow Control

Without flow control instructions a processor would only be able to execute program instructions sequentially. The ability to execute different parts of a program under different input/data conditions is fundamental to the nature of computing so the 68000, like all other processors, provides a number of useful mechanisms.

The 68000 provides a number of conditional branch type instructions for transferring control from one part of a program to another. One such instruction is called `beq` (Branch on Equal to zero) and this is a flow control branch which is only taken if the 68000's zero flag is set. To use this instruction to branch conditionally to a symbolic address called `EXIT` one would write

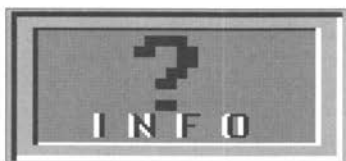
beq EXIT

Unconditional branch/jump and subroutine oriented branch/jump instructions are also available on the 68000. The main differences between ordinary branches or jumps and the subroutine oriented variety are that the subroutine forms automatically store a return address

on the stack. After a subroutine call has been executed, this return address can be retrieved and used to transfer control back to the main part of the program.

Insider Guide #8 – Branches and Jumps

The terms branch and jump tend to get used interchangeably and this is understandable because both types of instructions have similar end results – the program counter register gets loaded with a new value which causes the 68000 chip to get the next instruction from somewhere other than the next sequential instruction in memory. Branches and jumps, however, do work in slightly different ways because whereas jump instructions use real addresses branch instructions use displacements. Thus a jump instruction effectively tells the processor to go to location XYZ for its next instruction while a branch instruction supplies offset values from the current value of the program counter register.



It's a bit like someone asking you where Mr Jones (a neighbour) lives. You may live at number 30 and Mr Jones 3 doors down at number 36. You could say Mr Jones lives at number 36 – giving his absolute address – or you could say "Oh, he lives three doors away", pointing the caller either up or down the road as appropriate. In the latter case you've provided a relative address – a positive or negative displacement from a known anchor point.

Other Instructions

Instructions are provided which allow the 68000 to test, set, and clear individual bits and to rotate and shift operands. There are powerful address calculation instructions, automated loop instructions, and even instructions which allow data areas to be allocated within stack space as subroutine calls are made. A variety of instructions are also available for comparing particular operand values – these set the appropriate status register flags.

Insider Guide #9 – Subroutines

There are frequent cases in programming where the same sequence of instructions is needed in more than one place in a program. Instead of duplicating those instructions (which is wasteful of memory) microprocessors are provided with special instructions that allow a section of code to be re-used. These code sections are themselves mini-programs written to do well-defined jobs. Since they represent routines which may be called by other parts of a program, they are called subroutines.

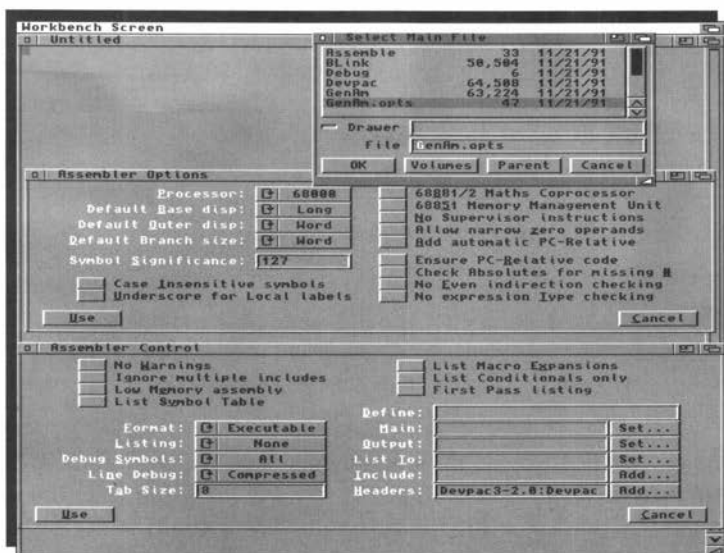
The 68000 provides two basic methods for transferring control to a subroutine. The jump-to-subroutine instruction, whose mnemonic is jsr, causes an unconditional jump to a



specified memory address. This instruction behaves just like the unconditional jump (jmp) instruction but, in addition to placing the specified jump address into the program counter, it also saves a return address on the stack. By placing a return-from-subroutine instruction (rts) at the end of a subroutine this address can be retrieved and placed into the program counter so the net result is that the processor having jumped to, and executed, a piece of suitably written subroutine code, returns to the instruction immediately following the original subroutine call.

A further instruction, called branch-to-subroutine (bsr), provides a relative addressing form of the subroutine call mechanism and in this case either an 8 or 16 bit displacement can be provided.

Assemblers, with the help of a linker, can translate your source code files into programs which can be run on the Amiga. Such assemblers have certain rules and conventions ...



An assembly language program consists of a number of statements. Some statements correspond directly to 68000 instructions, others are assembler-oriented directives known as *pseudo-operations* or *pseudo-ops*.

Program lines may contain as many as four fields – a label, a mnemonic, an operand or address field, and a comment. The mnemonic represents an instruction op-code while the operand, if present, is the data that the instruction acts on.

Here are some typical assembly code lines to illustrate the format. Don't worry about what the instructions are doing, it's the general layout of the program lines that is important not the details.

* -----
 ; an example assembly language code fragment
 * -----

```

OpenLib  move.l    library_name,a1    get library name
         move.l    _IntuitionBase,a6  get library base value
         rts
  
```

Not like BASIC is it? But don't worry too much because I'll let you into a secret – each line of assembly language is actually far simpler than a typical line of BASIC because it only involves the one operation. Assembly language instructions perform far simpler tasks than high-level languages commands and this will become apparent as we look at the various fields present in the above example code.

Comments

Comments are optional, ie they do not need to be present. They are added for the same reasons that REM statements are added to BASIC programs – to provide in-line documentation, lines to separate routines etc.

Assemblers vary in how they delimit comments but usually those lines which begin with an asterisk are treated as a whole line comment, any characters after a semicolon are similarly ignored, and any text after the operands field is usually also be treated as a comment providing it is separated by one or more spaces.

Labels

Labels do not have to be used but if they are, they normally have to be placed at the start of the line – some assemblers are quite fussy about field placement.

Many 68000 assemblers adopt a convention which allows white space to signify the end of the label – as in the above example – but also allow the label to start at a position other than the first character of the line providing it is terminated with a colon (:).

Each byte of each instruction or data item in an assembler program has, by virtue of its position in the program, an address by which it can be identified. Internally the assembler keeps track of this numerical position information by using a *location counter*. Referring to places within a program using such numbers is awkward as it means

the programmer has to remember the lengths of each instruction so labels can make life a lot easier. It also leads to far more readable code – in the above fragment the programmer can use ‘OpenLib’ rather than having to work with some relatively meaningless numeric value.

Labels can also appear in the operand fields and this, as the exit label in the following fragment illustrates, is commonly used to specify a location to jump or branch to.

```
move.l  _IntuitionBase,a6  get library base value
beq     exit                test result for success
CALLSYS CloseLibrary,_AbsExecBase
exit    rts                logical end of program
```

Programmers use labels to identify space set aside for variables and static program data, the starts of both the program and particular routines, entry and exit points, jump/branch positions etc. Given the purpose of labels in an assembly language program it should be obvious that it is best to use labels that are meaningful as OpenLib, exit, and library_name in the above example should show. Labels like HOWZAT or ICUR2Y4ME are less than useful.

Label conventions

The conventions which assemblers expect vary and sometimes vary considerably. Many assemblers, for

instance, place restrictions on the lengths of labels and on the characters which may be used within them. For example, the leading character often has to be a letter and usually only a few non-alphanumeric characters are allowed. Some assemblers allow long labels, others may not or may truncate them without warning.

An assembler, since it has to equate each label to a specific address, cannot allow the same label to be defined twice within a program. With older assemblers it was the programmer’s responsibility to ensure that duplicate label names were avoided. If, for instance, you had three routines similar to our last example fragment within the same program it was necessary to use, say, exit1, exit2, exit3 to avoid causing *duplicate label* errors. Modern day assemblers now provide something known as *local label support*. Here the assembler builds up the internally unique identifier by adding the local name to some previously supplied base name. Devpac, for instance, adopts a convention whereby a label beginning with a period (or optionally an underline) is attached to the last non-local label.

Assembler Directives

Assembler Directives are the pseudo-ops mentioned earlier and are used to define symbols, designate areas of memory for data storage, place fixed values in memory and so on. Directives also exist for more mundane operations such as controlling the listing and error reporting facilities of the assembler. Once again, conventions are going to vary from assembler to assembler but the detailed specifics are fully documented in your assembler manuals. Having said that, a few pseudo-ops do need to be dealt with because they are used extensively within this book.

The EQU Equate Directive

This allows the programmer to define a label with a specific numerical value. For instance:

NULL	EQU	0
TRUE	EQU	1
FALSE	EQU	0
SPACE	EQU	32

Most assemblers even allow you to define one label in terms of another, or in terms of a numeric expression:

BASE	EQU	10
STRUCT	EQU	4+BASE

None of these EQU type definitions cause the assembler to create any code. All that happens is that the definition supplied gets noted internally and from that point on the programmer is free to use the label wherever they would otherwise have needed to use the appropriate numerical value. Other advantages, in terms of program maintenance, also exist because if you alter a label at the front of a program that new definition is then automatically updated wherever the label has been used.

Storage Allocation Directives

All assemblers recognise a set of directives which allow you to reserve specified amounts of memory and initialise locations, or sets of locations, to particular values. It is usually possible to specify bytes, words or long word allocations by appending the appropriate *.b*, *.w*, or *.l* suffix to a directive. A *ds* (define storage) directive, when written as *ds.l*, allocates space for a number of four-byte (long

word) values. To reserve four bytes of uninitialised space for a variable called `_IntuitionBase` we could use:

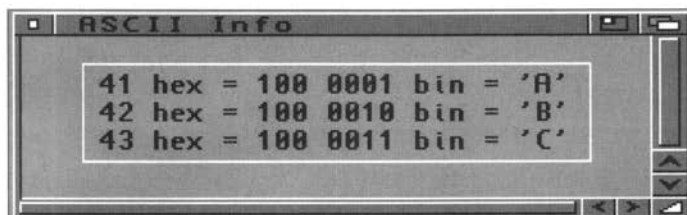
```
_IntuitionBase ds.1 1
```

Directives are also available for placing constant values in memory. The following statement uses `dc.b`, the byte form of a define constants directive, to store the numerical equivalents of the characters "intuition.library" plus a terminal NULL (zero) character in a set of memory locations whose start address has been labelled as "intuition_name".

```
intuition_name dc.b 'intuition.library',NULL
```

Insider Guide #10 – ASCII CODES

All microprocessor data is represented by numbers and so, to develop text-oriented programs, it has been necessary to devise codes whereby each character is represented by a number. Several schemes have been developed but the one used more than any other is called the American Standard Code for Information Interchange (ASCII). ASCII text files are so called because they use the ASCII code values to represent text characters.



Operands and Addresses

Most assemblers assume that all numbers are decimal numbers unless otherwise stated but can accept binary, octal, and hexadecimal numbers if suitably identified. The \$ sign, for instance, is frequently used to specify hexadecimal numbers. Modern assemblers offer great flexibility in terms of the complexity of the numeric expressions they accept and many provide multiplication, division, addition, subtraction, logical operations, use of parentheses etc. Assemblers which support the generation of floating point co-processor code also provide for the use of floating point constants.

ASCII character constants, as illustrated in the previous section's `dc.b` directive example, are also allowed with quotes or double quotes being used to delimit the start and the end of the set of characters.

Macro Assembly

You frequently find that particular sequences of instructions crop up again and again. Macro 68000 assemblers, such as Charlie Gibb's A68k and HiSoft's Devpac, allow you to assign names to such instructions sequences so that when the name is encountered the assembler automatically expands it to produce the original set of instructions. Nowadays this facility is not restricted to predefined, absolutely fixed, instruction sequences – macros can be used which contain parameter placeholder markers. When the macro is used the parameters provided for that particular use instance are inserted into the code that is generated. Macros allow assembly language programming to be done at a significantly higher level than was previously possible and they are in fact an essential part of Amiga assembly language programming. A great many pre-defined macros have been made available to the programmer in the Amiga system header files and you'll find a number of examples of how such macros are used later in the book.

Don't, incidentally, make the mistake of thinking that macros are the same as subroutines because they aren't. The big difference is that each time a macro is used the corresponding code is inserted at that point in the source file. A subroutine, on the other hand, only physically exists in the one place within the source file but the code itself may be called many times.

Conditional Assembly

Most assemblers provide directives which allow specified parts of a program to be assembled, or not assembled, depending on specified conditions. For instance the single standard start-up code source file provided by Commodore includes changeable constant declarations which allow the automatic generation of a number of different start-up module versions. Programmers often include debugging code in their programs but conditionally remove the relevant sections of code in the released versions of their programs.

If You're Having Trouble

I've tried to protect you from as many of the technical nasties as possible but in case you are finding things hard going let me stress that you do not need to either remember, nor understand, everything that has been dealt with. Indeed some of the topics will only really begin to make sense once you have some practical experience under

Insider Guide #11 – Program Layout

Assemblers only really need single spaces to be able to distinguish between the various fields present in an assembly language program line and so it is quite permissible to type in code lines in this format:

OpenLib move.l library_name,a1 get library name

Most programmers do however use either tabs or extra spaces in order to make the listings look tidy and you should do the same. If the listings look neat they'll be easier to examine, and you'll make fewer mistakes. This is how program lines are displayed in this book:

```
OpenLib  move.l library_name,a1  get library name
          move.l _IntuitionBase,a6  get library base value
          ↑                          ↑
          tab to here                 tab to here
```

By default the assembler expects the label to occur at the start of the line and if the above lines were laid out in the following fashion:

```
<space>OpenLib move.l library_name,a1 get library name
<space>move.l _IntuitionBase,a6      get library base value
```

the assembler would think that OpenLib was meant to represent a 68000 instruction and report an error. Similarly if a program line which does not use a label is written so that the instruction itself starts at the beginning of a line like this:

```
OpenLib move.l library_name,a1    get library name
move.l _IntuitionBase,a6         get library base value
```

the assembler thinks that move.l represents a label and again generates an error message. Shifting the second line of the above example to the right either by one or more spaces or by a tab so that the move.l instruction no longer starts at the beginning of the line eliminates the error.



your belt. So, just carry on with the subsequent chapters and refer back to these early chapters, to consolidate or add to the things you've already picked up, as and when necessary.

the letters A-F. Each column in a base 16 number therefore represents some power of the base. For example the decimal number 16 itself is written as 10 hex, because:

$$\begin{aligned} 10 \text{ hex} &= 1 \times 16^1 + 0 \times 16^0 \\ &= 16 + 0 = 16 \text{ decimal} \end{aligned}$$

Similarly 1F hex would be:

$$\begin{aligned} 1F \text{ hex} &= 1 \times 16^1 + 15 \times 16^0 \\ &= 16 + 15 = 31 \text{ decimal} \end{aligned}$$

The fact that the bases of the binary and hexadecimal numbering systems are power related (2 to the power of 4 equals 16) allows one hexadecimal digit to represent four binary digits. Best of all, the binary-to-hex conversion process is very easy to understand once you've learnt the following table:

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

To convert a hexadecimal number into binary form you just replace each hexadecimal digit with its group of four binary digits. To convert a binary number to its hex form you peel off (from right to left) groups of four bits and replace them with the corresponding hex digit!

So, to convert CF hex to the binary equivalent you'd replace each of the two hexadecimal symbols with the binary equivalents:

CF hex = C F
1100 1111 = 11001111 binary

To go the other way you take groups of four bits from the binary number and replace them with the corresponding hex digits. The binary number 1111000010101010, for example, could be translated to hexadecimal form as follows:

1111000010101010 = 1111 0000 1010 1010
F 0 A A = FOAA hex

Using (and converting between) binary, hex and decimal number systems is not that difficult but it does take practice. Familiarity with hex and binary number forms is also essential for understanding how the bitwise logical operations provided by both microprocessor instructions and high-level languages work. For instance, logical AND and OR instructions for instance, which I'll assume you know about from languages such as BASIC, perform operations based on these two truth tables:

X	Y	X AND Y
0	0	0
1	0	0
0	1	0
1	1	1

Logical AND Operation

X	Y	X OR Y
0	0	0
1	0	1
0	1	1
1	1	1

Logical OR Operation

Being able to picture in your mind what these tables mean is a big advantage. If you AND two operands together then only those bit positions where both operands have a bit set to 1 will produce a 1 in the result. With the OR operation you'll get a 1 in the result when either (or both) of the bits in that position in the corresponding operands are set to 1.

The bit pattern for F0 hex for, instance, is 11110000 so ANDing any value with F0 hex forces the lower four bits of the result to zero. The value F0 hex is called a *mask* because it masks out certain bit positions. The OR operation is equally useful because it can force bit positions to take particular values.

Insider Guide #12 – Truth Tables

You don't need to panic because a truth table is just the possible input and output values of some, usually simple, logical operation, laid out in an easy-to-use table form. Let's take an example – the logical AND operation as applied to the bits of binary numbers. As you know, each digit of a binary number, since it is a base 2 number system, can only take one of two values: 0 or 1. ANDing is a function that effectively says: "Take any combination of these possible input values (which we'll label as X and Y) and if both are true (1) then my output value is true (1), otherwise my output value is false (0)."

How many different possibilities present themselves for applying this AND operation and what results do you get with each case?

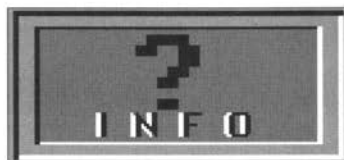
If you experiment you will find that there are just these four possible combinations:

INPUTS		OUTPUT
X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

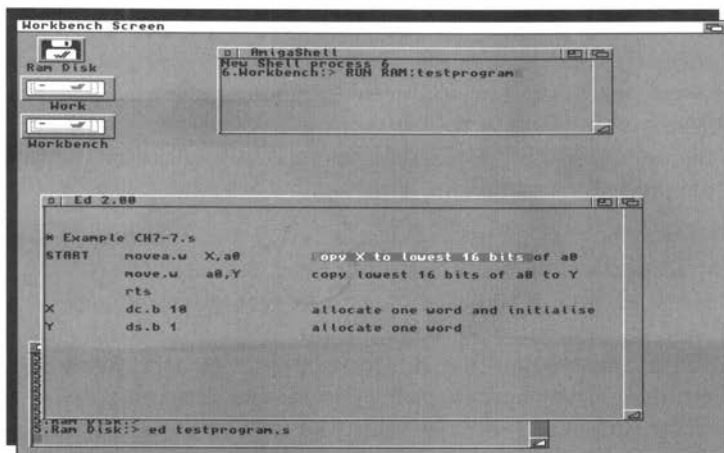
These values represent the AND operation's truth table.

By convention, however, truth tables are usually provided in a rather different form known as a matrix representation. In this case the inputs are shown as row and column headings with the results forming the entries in the main body of the table like this:

		Input X		
		0	1	
Input Y	0	0	0	← Results of Logical ANDing
	1	0	1	



One of the easiest ways to come to terms with 68000 assembly language programming is to look at some programs and so this is exactly what we shall be doing in this chapter.



The good news, as far as this chapter is concerned, is that it brings us to the point where we can actually start looking at some assembly language programs. Before we do this however I'd better give you a few words of warning, just in case you are expecting to dive straight into the world of Amiga graphics and multitasking.

The plain truth of the matter is that to explain the purposes of a lot of the 68000 instructions we are going to need to start with very simple examples containing only a few instructions. Unfortunately such simple programs, by definition, tend not to do very much. In fact the programs that we'll be dealing with in this chapter don't even have any visible output when they are run.

On the face of it, the prospect of spending time examining programs that add two numbers together, or copy a few bytes from one set of memory locations to another is hardly likely to

instill a burning desire to learn about the 68000. Nevertheless this chapter useful because it illustrates a number of important 68000 instructions. Granted they may not seem important within the context of the programs in this chapter but be patient – these examples have been chosen so as to illustrate the operations that you'll be expected to know about once we get into proper Amiga 68000 programming.

Since the examples provided have no output there is, to be honest, little point in assembling and running the programs. Because of this I've left discussions of the practical issues of assembly until Chapter Nine where we *do* create a program that does something. The best idea for the moment is to work through the material provided, and just think about the examples in relation to the things you've picked up so far. You may also find it useful to refer to Appendix A because this lists a selection of commonly used instructions, along with additional details of the 68000 processor's addressing modes etc.

Despite the simplicity of the examples it is however quite possible to run the programs in this chapter from a Shell window. Users who have access to an Amiga 68000 monitor/debugger program such as Devpac's MonAm and, more to the point, are familiar with using it, might find it useful to enter and run the odd example in single-step mode. The program may have no visible output, but it is still possible to see how the various instructions affect the state of the processor's registers and flags.

I wouldn't recommend anyone to start struggling with a debugger program just for the sake of it. Debuggers are invariably most unfriendly beasts, and you really need some coding experience under your belt before you try using one. With that warning is out of the way, let's look at some 68k code:

Data Transfer

Data movement on the 68000 can be achieved with move instructions. A number of variants exist but the basic format

is:

move.<size> source destination

Size values can be b (byte), w (word) or l (long word) but if the object size is not specified then a word size (16 bit) is assumed by default.

You may remember from Chapter Five that I said that labels can be used to identify memory locations and that this saves having to deal with meaningless numeric addresses. Suppose then that we have

asked the assembler to set aside one byte of RAM and label it as location X. To move the contents of this location to the lowest 8 bits of register d0 we write:

```
move.b X,d0 copy byte X to lowest 8 bits of d0
```

Similarly, to move the lowest 8 bits of register d0 to a location which has been labelled Y we could write:

```
move.b d0,Y copy lowest 8 bits of d0 to Y
```

One way of initialising and/or allocating the above X and Y variables would be to use the byte forms of the assembler's "define constant" and "define storage" pseudo-ops, dc.b and ds.b like this:

```
X dc.b 10 allocate one byte and initialise it to 10  
Y ds.b 1 allocate one byte but do not initialise it
```

If we put these fragments together we can build a program which copies the pre-initialised 1 byte value held in location X to location Y. Notice the overall layout of the program – it starts with some instructions which are followed by assembler pseudo-ops telling the assembler that some storage space for variables is needed:

*** Example CH7-1.s**

```
START move.b X,d0 copy byte X to lowest 8 bits of d0  
move.b d0,Y copy lowest 8 bits of d0 to Y  
rts end of program  
X dc.b 10 allocate one byte and initialise it to 10  
Y ds.b 1 allocate one byte but do not initialise it
```

The program starts with X holding the value 10 and Y being undefined. After it has been run byte X still contains the value 10 but byte Y also contains 10.

There was no particular reason why register d0 was chosen – any of the 68000's data registers (d0-d7) could have been used instead.

There is in fact a much easier way to achieve the above copy operation because the 68000 allows you to transfer data directly from one memory location to another like this:

```
move.b X,Y copy byte X to byte Y
```

This means that it's possible to eliminate the use of d0 as a temporary storage register in the above program and write this simpler version:

Insider Guide #13 – Additional Info

Nowadays most assemblers initialise *ds.x* statements to zeros but, for consistent documentation, it is best to assume that such initialisation is not done. If you really want to initialise byte *Y* to zero choose the *dc.b 0* pseudo-op.

```

clr.l d0
rts          end of program
MyData      dc.b 18,2,4,3,5,6,24,79,8
MyText      dc.b 'Just some example text',NULL
MySpace     ds.b 5
MyZeroSpace dc.b 8,8,8,8,8

```

The *rts* (return from subroutine) instruction at the end of the code is used in these examples to return control back to the Amiga's operating system. Don't worry at the moment about understanding what it does, as we'll deal with those issues later. Strictly speaking even these simple programs should terminate with register *d0* set to zero (achieved by using a *move.l #0, d0* or a *clr.l d0* instruction just before the *rts*). But, for simplicity's sake, this Amiga-orientated operation has not been included in these, otherwise general, discussions.

*** Example CH7-2.s**

```

START  move.b  X,Y  copy byte X to byte Y
      rts

```

X **dc.b 10** **allocate one byte and initialise to 10**

Y **ds.b 1** **allocate one byte but don't initialise**

When *move* is used to copy a piece of data the instruction, providing the destination is not an address register, generally affects the flags in the user-byte 68000 status register. These flags are variously called the *user-byte flags*, *condition codes*, or the *status byte flags*. In this book I've used the term *status byte flags*. With *move* instructions the Zero (Z) and Negative (N) flags are set to an appropriate state whilst the Overflow (V) and Carry (C) flags are cleared.

Now that you've seen how to move 8 bit values you'll be pleased to know that you can move word (16 bit) and long word (32 bit) values just as easily. The following version performs a word (two byte) copy.

*** Example CH7-3.s**

```

START  move.w  X,Y  copy word X to word Y
      rts

```

X **dc.w 10** **allocate word and initialise to 10**

Y **ds.w 1** **allocate word but don't initialise**

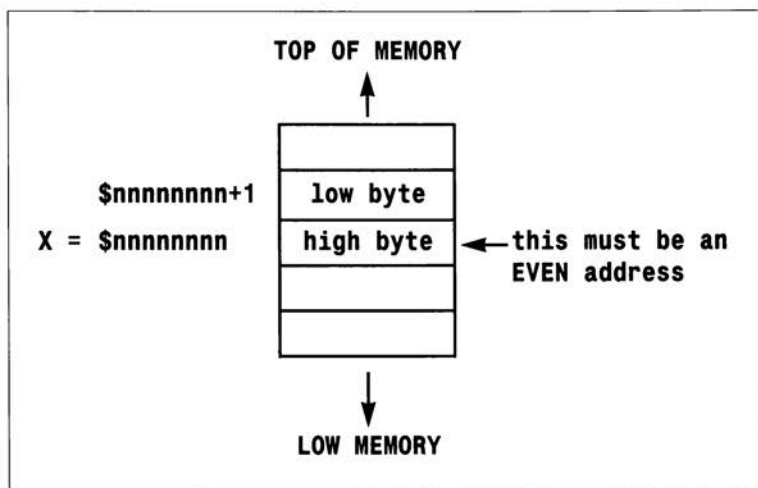
Since instructions assume a word size by default it is not necessary to include the 'w' size indicator on the move instruction. Example CH7-3.s could therefore just as easily have been written as:

*** Example CH7-4.s**

```
START  move   X,Y   copy word X to word Y
        rts
X      dc.w 10      allocate word and initialise to 10
Y      ds.w 1       allocate word but do NOT initialise
```

Insider Guide #14 – Word Storage

Since two bytes are needed to store a word value, and since each byte has an individual address, you might be wondering what address the assembler assigns to the word variables. On the 68000 Amiga system words are stored in memory as shown below.



Above: 68000 storage of words in memory

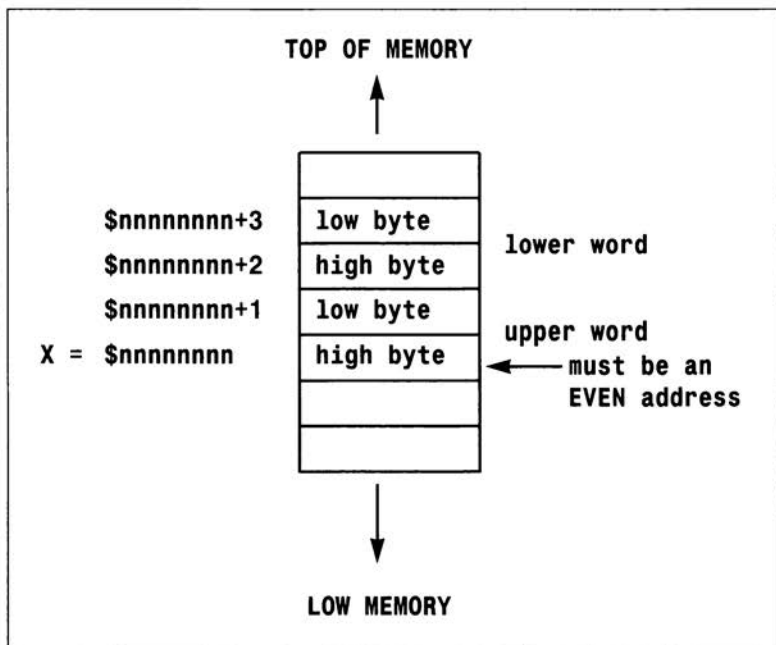


Insider Guide #15 – Long Word Storage

Four bytes are needed to store a long word value and, on the 68000, these items are again stored in a particular order. Just as a word can be expressed in terms of an upper and lower byte so we can consider a long word as containing an upper and lower word:

$$\begin{array}{rcl}
 32 \text{ bits} & = & 16 \text{ bits} \quad 16 \text{ bits} \\
 \langle \text{long word value} \rangle & = & \langle \text{upper word} \rangle \quad \langle \text{lower word} \rangle
 \end{array}$$

The 68000 stores the word components of long words in the same way as it stores the byte components of ordinary (16 bit) words, ie it stores the bytes of the most significant word first. The net result is that long words are stored in memory as shown below.



Above: 68000 storage of long words in memory

BEFORE YOU LOOK at the following solution, try to change program Example CH7-4.s to produce a long word version. Here's the result you should obtain:

*** Example CH7-5.s**

```

START  move.l  X,Y    copy long word X to long word Y
      rts
    
```

X	dc.l 10	allocate one long word and initialise
Y	ds.l 1	allocate uninitialised long word

In transferring data from one set of locations to another, Example CH7-5.s used absolute addressing – remember the X and Y labels used in the move.l X, Y instruction represent numerical addresses. Another way of writing the programs that we've just been looking at is to reserve uninitialised memory space for both the X and Y variables and then explicitly initialise the X variable when the program is run. The following example uses an additional immediate addressing move instruction to load variable X with the value decimal 10. By convention immediate addressing on the 68000 is signified by placing a hash (#) sign in front of the operand:

*** Example CH7-6.s**

START	move.l #10, X	initialise long word X to 10
	move.l X, Y	copy long word X to long word Y
	rts	
X	ds.l 1	allocate uninitialised long word
Y	ds.l 1	allocate uninitialised long word

You can see from the instruction code summaries provided in Appendix A that the move instruction is unable to transfer data *to* an address register. In actual fact a specialised form of the move instruction – called movea (move address) – is available for this purpose and there are a number of differences between move and movea.

Firstly, like most direct address register instructions, movea can only operate on word or long word values. Secondly, movea does not affect any of the processor's flags – for address-orientated operations, this is a convenience not a limitation. Lastly, movea sign-extends any word values it is working with. This means that the uppermost bit (bit 15 of the word) is propagated throughout the upper 16 bits of the address register. Sign extension was introduced on the 680x0 series to allow a form of absolute addressing based on word addressing to be used (as opposed to a full long word address) and you can find additional details in Appendix A.

Although it is not a good idea to use address registers for such purposes we can write a word (16 bit) version of our original Example CH7-1.s data copying program like this:

*** Example CH7-7.s**

```

START  movea.w  X,a0    copy X to lowest 16 bits of a0
        move.w   a0,Y    copy lowest 16 bits of a0 to Y
        rts
X       dc.w    10       allocate one word and initialise
Y       ds.w    1        allocate one word

```

Data Transfer Using Address Registers

As it happens most 68000 assemblers *do* allow you to use the move mnemonic when specifying an address register. Program Example CH7-7.s actually can be re-written as:

*** Example CH7-8.s**

```

START  move.w   X,a0    copy X to lowest 16 bits of a0
        move.w   a0,Y    copy lowest 16 bits of a0 to Y
        rts
X       dc.w    10       allocate one word and initialise
Y       ds.w    1        allocate one word

```

In this last example the assembler automatically inserts a movea instruction for loading register a0 and this means that, unlike data register loading operations, the address register loading operation does *not* affect the processor's status flags. More subtle differences can also occur as this example shows:

*** Example CH7-9.s**

```

START  move.w   X,a0    copy X to lowest 16 bits of a0
        move.w   a0,Y    copy lowest 16 bits of a0 to Y
        rts
X       dc.w    $FFFF   allocate one word set to FFFF hex
Y       ds.w    1        allocate uninitialised word

```

Here we are using a word data value which includes a 1 in the uppermost position (FFFF hex = 1111 1111 1111 1111). Because the first instruction is really a movea, and because the sign bit (bit 15) of the word \$FFFF is set high then the value that movea transfers to register a0 is FFFFFFFF hex, and not FFFF hex.

Since the program only copies the lower 16 bits of the register back to location Y this doesn't affect the result in this case. But the instruction has, of course, affected the upper 16 bits of the a0 register in a way that the related data register version of the program would not do.

Most 68000 coders soon get used to the flag and sign extension implications of address register usage, use the move mnemonic for both data and address orientated instructions, and let their assemblers decide on the correct object code instruction.

Complementing a Value

Complementing a number means turning all the 1s present in the number, to

0s and turning all the 0s to 1s. If, for example register d0 contained the value:

```
d0 = 0000 0000 0000 0000 0000 0000 0000 0000 binary
    =  0   0   0   0   0   0   0   0   hex
```

then the complemented value is:

```
d0 = 1111 1111 1111 1111 1111 1111 1111 1111 binary
    =  F   F   F   F   F   F   F   F   hex
```

Try and confirm for yourself that, if d0 = 1F01 hex, then after a long word (32 bit) complement operation d0 contains E0FE hex. Write out each hex digit in the binary form as above, invert all the bits, and then translate the answer back to hexadecimal form.

The 68000 instruction which performs this operation is called NOT and, like many other instructions, it exists in byte, word and long word forms. Here's a short program which uses immediate addressing to load d0 with the byte value 0F hex, inverts it, and then stores the result in a location whose symbolic name (ie its label) is RESULT

*** Example CH7-10.s**

```
START  move.b  #F,d0      initialise low 8 bits of d0 to F hex
        not.b   d0        invert lower 8 bits
        move.b  d0,RESULT copy inverted d0 to RESULT
        rts

RESULT ds.b 1           allocate one byte
```

As was the case with the earlier examples the 68000 allows us to eliminate the use of a temporary storage register by using the not.b instruction directly on a memory location:

*** Example CH7-11.s**

```

START   move.b   #$F,RESULT   store value directly in RESULT
        not.b    RESULT       invert value
        rts
RESULT  ds.b 1                allocate one byte

```

In the above example the not.b instruction is using absolute addressing. In the previous example the register addressing form was used.

Addition

The 68000's basic addition instruction uses the syntax:

add<.size> source, destination

where the result of the "source plus destination" addition gets placed in the destination register. This feature is common to a great many 68000 instructions that work with two operands.

So far the instructions we have looked at have allowed source and destination operands to be either in registers or memory. Not all 68000 instructions are that flexible and in fact the 'add' instruction only allows one of its operands to be in memory. You may add the contents of a register to a memory location, or do the reverse – add the contents of a memory location to a register. What you cannot do, however, is add the contents of one memory location directly to the contents of another.

This limitation means that for this instruction we need to use a temporary register much as we did with our early data copying examples. The following example loads register d0 with a number contained in NUMBER1 and then adds it to the contents of the memory locations represented by the label NUMBER2. After the program has been run the variable NUMBER2 contains the value 7.

*** Example CH7-12.s**

```

START   move.l   NUMBER1,d0    load 1st number into d0
        add.l    d0,NUMBER2    add contents of d0 to NUMBER2
        rts
NUMBER1 dc.l 3                set initial value to 3
NUMBER2 dc.l 4                set initial value to 4

```

Until now I've mentioned the byte, word and long word forms of variables but have not said anything about when the various forms should be used. As far as data items are concerned the unwritten rule for the assembler programmer is the same as for the programmer working in any other language, namely to conserve as much memory as possible and not to waste it by allocating unnecessary space.

Have a look at the internal contents of the two four byte numbers used in the previous example:

	byte 3	byte 2	byte 1	byte 0	
NUMBER1	00000000	00000000	00000000	00000011	decimal 3
NUMBER2 (before)	00000000	00000000	00000000	00000100	decimal 4
NUMBER2 (after)	00000000	00000000	00000000	00000111	decimal 7

Both NUMBERS *and* the final result fit comfortably into an eight bit byte so, in all honesty, we did not need to use long word size variables, bytes would have done. Here then is an improved version:

*** Example CH7-13.s**

```
START    move.b  NUMBER1,d0    load 1st number into d0
         add.b   d0,NUMBER2    add contents of d0 to NUMBER2
         rts
NUMBER1' dc.b 3                set initial value to 3
NUMBER2  dc.b 4                set initial value to 4
```

Only two bytes of variable storage space are needed instead of the eight used previously and the byte-orientated forms of the instructions execute more quickly as well. Programmers would therefore say that this new version of the program was "more memory efficient", or just "more efficient" than the previous one.

Putting Some Pieces Together

Now let's try something a little more complicated. We'll set up some space for a long word variable called NUMBER1, initialise it using immediate addressing to some arbitrary value (I've used 1FFFFFF hex), increment it by 1, complement the result, and then store it in a variable called RESULT. Here's one program that does the job:

*** Example CH7-14.s**

```
START    move.l  #$1FFFFFF,NUMBER1  initialise number
        move.l  #1,d0              load d0 with value 1
        add.l   NUMBER1,d0         increment d0 copy of NUMBER1
        not.l   d0                 complement result
        move.l  d0,RESULT
        rts

NUMBER1  ds.l 1                   space for number
RESULT   ds.l 1                   space for result
```

Depending on what was actually required there are many ways that a program similar to the above could have been written. It might, for instance, have been appropriate to place the original value directly in the locations assigned for the result, and do the addition and complement operations on the result locations like this:

*** Example CH7-15.s**

```
START    move.l  #$1FFFFFF,RESULT  initialise number
        addi.l  #1,RESULT          increment value
        not.l   RESULT            complement result
        rts

RESULT   ds.l 1                   space for result
```

In the above example a special form of the add instruction, `addi`, is being used. This allows an immediately addressed source operand (in this case 1) to be added directly to the destination operand. If you take a sneak preview of the add addressing mode details in Appendix A you'll find that the normal add instruction couldn't have been used in Example CH7-15.s because, to use immediate addressing, the destination needs to be a data register.

However, as is the case with a number of instructions, most 68000 assemblers do let you write statements such as:

```
add.l    #1,RESULT    increment value
```

and then automatically translate the instruction to

```
addi.l   #1,RESULT    increment value
```

so program Example CH7-15.s can be re-written as:

*** Example CH7-16.s**

```
START    move.l    #$1FFFFFF,RESULT    initialise number
         add.l     #1,RESULT           increment value
         not.l     RESULT              complement result
         rts
RESULT   ds.l     1                    space for result
```

Quick Instructions

For immediate operands within limited ranges the 68000 offers a number of quick instructions.

Instead of using real immediate addressing – where the operand is placed immediately after the op-code in memory – these instructions have a data value buried in the instruction op-code itself.

The `moveq` instruction, for example, uses a data register as the destination and allows 16 bit operands to be specified. It does, however, sign extend the data to long word size.

To load register `d2` with the value 23 for instance we can write:

```
moveq    #23,d2          load d2 with value 23
```

Add and subtract quick instructions also exist although these only allow immediate data in the range 1-8 to be specified. To increment by 4 the contents of a memory location whose address has the symbolic name `RESULT` we might, using absolute addressing, write:

```
addq     #4,RESULT
```

If we choose to load the address of `RESULT` into register `a1` we can use the 68000's indirect addressing scheme instead to specify the destination address:

```
move.l   #RESULT,a1     load a1 with address of RESULT
addq     #4,(a1)        add 4 to contents of the byte
                                "pointed to" by register a1
```

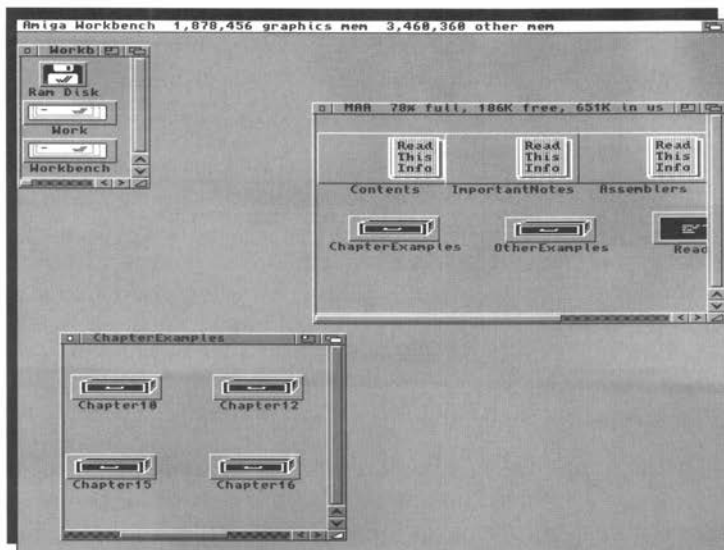
where the destination operands (`An`) notation is the 68000 assembly language form for specifying an indirect address.

Another method of loading register `a1` with the address of the `RESULT` variable is to use the more specialised Load Effective Address (`lea`) instruction. If this is done with the above fragment the code ends up looking like this:

lea	RESULT,a1	load a1 with address of RESULT
addq	#4,(a1)	add 4 to contents of the byte "pointed to" by register a1

The earlier loading of the address of the RESULT operand into a1 using an immediate addressing move instruction served us well enough but, in general, the lea instruction is a far more flexible alternative and much more use is made of it later on in the book.

In this chapter you will, believe it or not, be writing your first assembly language Intuition program. In order to do this it is essential to learn a little about the Amiga libraries first.



This chapter contains another subject that can be difficult on first encounter but though previously I've suggested you skip any issues that seem awkward and return to them later – with this chapter you *must* persevere. Read it, think about it, sleep on it, read it again, but whatever you do... don't give up! The material in this chapter is, at least as far as the Amiga assembly language newcomer is concerned, absolutely vital.

A library, in the conventional programming sense, is just a collection of pre-written routines. The idea is that by supplying a set of ready-made routines for all commonly needed tasks the programmer is saved time and effort because he or she doesn't need to re-invent those routines.

It sounds straightforward but, for the Amiga programmer, libraries can be the source of much confusion simply because the term is used in a number of different contexts. Your assembler, for instance, may have its own libraries of standard functions which are used at the linking stage of program assembly. These *linker libraries* are just disk files of useful functions arranged in a special, easily linker-accessible, format. When a reference to one of these functions is used within a program it causes the construction of an equivalent *unresolved* reference in the intermediate object code file. At link time the linker must, with some guidance from the programmer, find the library file that contains the function and physically copy it into the program being created. One example of a linker library which you may have already heard of, is the `amiga.lib` library.

Run-Time Libraries

The Amiga also uses another type of library based on a dynamic Exec library system. Exec incidentally is the part of the Amiga operating system that handles a lot of the housekeeping jobs such as multi-tasking. We won't be looking inside Exec at all in this book – it is an extremely complicated piece of software – but we will be using its library facilities. Exec style libraries do have one thing in common with linker libraries in that they exist quite separately from the applications programs which use them.

That is where the similarities end because whereas linker library code gets added onto the assembled program at the linking stage, ie before the program is turned into runnable form, these Exec-style run-time libraries exist separately and never form part of the real program code at all. The libraries are written in a way which allows any number of different programs to use them simultaneously (or at least appear to do so within Exec's multi-tasking framework) and this obviously makes them much more flexible and efficient. It is the use of these "run-time libraries" that forms the subject matter for this chapter.

At the risk of causing temporary confusion I now have to tell you that Exec itself is also effectively organised as a run-time library. Naturally it offers a set of library functions which programmers can use. In fact some of these Exec routines are actually used to open and close other run-time libraries.

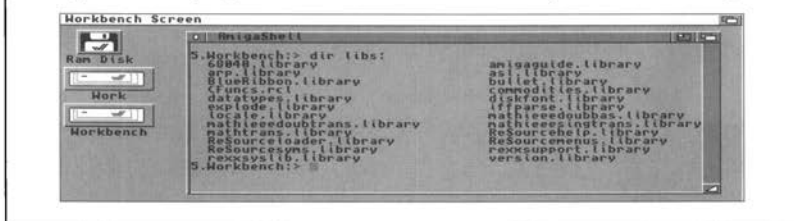
Opening a Library

Programs tell Exec that a library is needed by attempting to open that library using an `OpenLibrary()` function. When such a call is made Exec does several things. It search-

es its lists of libraries which are already open and available. If the library is found then Exec simply returns the address of the library and makes an internal note that another program is now using it.

Insider Guide #16 – Where have all the functions gone?

On less sophisticated computers, especially early eight bit machines, the location of various system functions was fixed because the routines were pre-programmed into a ROM chip and their addresses were therefore static. On the Amiga things have changed and to all intents and purposes you won't know, until the time you come to use a library, whereabouts its functions are. Some libraries are currently positioned in read only memory (ROM), others may be available in RAM because they've been loaded during system start-up, but many actually remain on disk until the first applications program indicates that it needs a particular library.



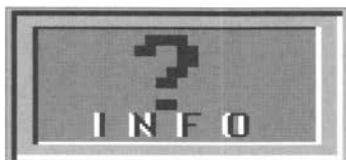
If the library is not already open, Exec passes on the request to AmigaDOS asking it to look for, and then load, the specified library. AmigaDOS looks in the LIBS: logical device – if you boot from the Workbench disk, for instance, then this logical device will have been assigned to SYS:LIBS, ie the LIBS directory of the WorkBench disk. If AmigaDOS finds that library it loads it and tells Exec where it has been placed.

Exec then records the fact that the library is now available by adding it to its list of available libraries. Exec never attempts to remove these library modules whilst they are in use. Should the last user of a particular active library indicate that they no longer need access to the routines – which they do by executing a CloseLibrary() function – Exec's library manager may then remove the memory copy of library and release the associated memory so that it is free for other use.

As all this happens a lot of complex operations get carried out but the good news is that you don't need to worry about this at all. As far as an applications program is concerned, most of these operations are transparent and this is so even at the assembly language programming level. All a program has to do to use a given library is open it using the

Insider Guide #17 – Failed Open Library Calls

Why does a library fail to open? The system might not have been able to find it on disk, the specified version might not be available, the programmer might simply have spelt its name wrong within the program, or the system might even be running out of memory and have insufficient space to load a new library.



The important point is that you must not make any library function calls unless you have got a valid base pointer or you will doubtless get a visit from the Amiga guru!

Insider Guide #18 – An Important Exec Function

Function Name: `OpenLibrary()`

Description: Open a run-time library

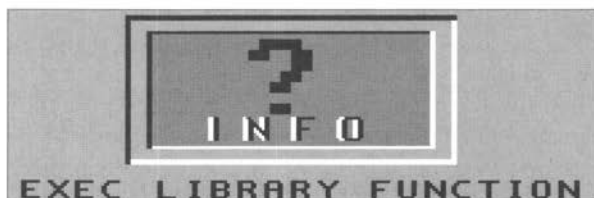
Call Format: `base_address =
OpenLibrary(library_name, version);`

Registers: D0 A1 D0

Arguments: `library_name` – the address of a null terminated string
`version` – a library version number

Return Value: `base_address` – the address of the base of the library. If the library could not be opened a NULL value is returned.

Notes: User must not attempt to use any library functions if this function did not succeed.



Exec `OpenLibrary()` function, and then use the library routines in much the same way that the `OpenLibrary()` function was itself used.

Insider Guide #19 – Another Exec Masterpiece

Function Name: CloseLibrary()

Description: Close a previously successfully opened library

Call Format: CloseLibrary(base_address);

Registers: A1

Arguments: base_address – the library base address

Return Value: None

Notes: User must not make library calls to a library after it has been closed.

The only thing which the applications program must do is ensure that the OpenLibrary() call was successful and it does this by checking that the address returned is non-NULL – ie not zero. If the address returned has a zero value then the system has failed to open the library.

If an applications program follows this protocol it never needs to concern itself with where the routines are in memory, nor with the fact that other programs may also be using the same routines. This makes for an extremely powerful and flexible library system and there's no doubt that much of the Amiga's power has stemmed directly from its run-time library arrangements.

A Sneaky Exec Trick

I've already mentioned that the first stage in using a library is to open it by using the Exec OpenLibrary() function. You may now be wondering how it is possible to open the Exec library in the first place. The simple answer is that you do not need to because the Exec library never has to be opened. Exec's base address, known conventionally as SysBase, is permanently available because it is stored in the long-word memory location whose first byte is at location 4. The four bytes which make up this long word location are called AbsExecBase and, because this is loaded with a pointer to the Exec library during system start-up, the Exec library is always alive and kicking from the word go.

Making a Library Call

By convention the base address of the library is placed in register a6. An indirect subroutine call is made using the appropriate library vector offset (LVO) value to specify the routine to be executed. Indirect subroutine calls of this type are very important on the Amiga and they're used because the arrangement is connected with the way the Amiga library functions are accessed internally (the explanations of which involve some pretty advanced topics including the use of things called *jump tables* which are not going to be discussed).

Insider Guide #20 – The Importance Of Being a6

You might be forgiven for thinking that any register can be used to perform indirect library routine calls. This is most definitely not the case and there is a strict system convention which says that register a6 must be used. Why? It's because many library functions call other library functions in order to carry out their work.



When this is done the function doing the nested library call must also follow the system conventions and provide a library base address. By convention it expects it to be present in register a6. Exceptions to the a6

rule do exist but, to be honest, it is safer if you forget about any special cases and regard the a6 rule as absolute!

What happens, as far as the indirect subroutine call with displacement is concerned, is that the address in the specified address register gets added to the specified LVO function call displacement and this produces a destination subroutine address that leads us to the right library function. Regard it as magic, if you like, but don't go looking too hard for in-depth explanations until you have a good understanding of the material covered in this chapter!

As far as writing library opening code is concerned we are virtually there. I've already mentioned that in the case of the Exec library the base address is already available – it can be loaded directly from memory using AbsExecBase. The bare bones code for an OpenLibrary() Exec call can therefore be written like this:

```
move.l  _AbsExecBase,a6      get base address of Exec library
jsr     _LV0OpenLibrary(a6)  make the indirect subroutine call
```


Before this sort of code can be executed it is necessary to set up any parameters which the library function needs. If you look back at the `OpenLibrary()` function you'll see that it needs a pointer to a library name in register `a1`, and a version number in `d0`. For the moment we'll be setting the `d0` to zero because this tells Exec that any library version will do.

Library Vector Offset (LVO) Values

L V O
offset
values

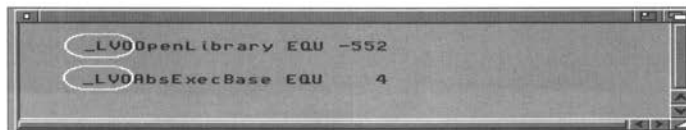
can be acquired in a number of ways but for the moment we'll be putting LVO definitions at the start of our programs because it is easiest. You will find an abbreviated set of tables in Appendix B and from the Exec entries you'll see that the LVO value for the Exec `OpenLibrary()` function is `-552` or `-0228` hex. The assembly language programmer is therefore quite at liberty to define the displacement in this fashion:

```
move.l  _AbsExecBase, a6  get the base address of Exec library
jsr     -552(a6)          make the indirect subroutine call
```

The trouble with this latter approach is that you lose the inherent documentation that the LVO references provide. Let's face it, the number `-552` will not tell you what library call is being made unless you've memorised all of the LVO tables. The reference `_LVOOpenLibrary` is much more meaningful.

Insider Guide #21 – A Below Average Score?

You may have noticed in the code fragments used that `AbsExecBase` and the LVO value have underscore prefixes. This stems from an internal C language convention and the underscore used in all assembly language forms has been introduced simply to provide compatibility between C and assembler header files and code. Not all programmers use these underscore arrangements but it's a good habit to cultivate because it'll be useful when you come to more advanced coding.



Closing a Library

Closing a Library is just as easy as opening one. You use the same type of indirect subroutine call, but specify

the `CloseLibrary()` function instead:

```
move.l  _AbsExecBase, a6      get base address of Exec library
jsr     _LVOCloseLibrary(a6) make the indirect subroutine call
```

Putting It All Together

The Intuition Library provides a function called `DisplayBeep()` which,

when supplied with a null (ie zero) address, causes all visible Amiga screens to be flashed. What we're now going to do is put together all the things we've learnt over the last few chapters and produce a program which causes your Amiga to flash its screen. We need to set up the LVO definitions somewhere near the start of the program and this means using some EQUate definitions:

```
_LVOpenLibrary   EQU   -552
_LVOCloseLibrary EQU   -414
_LVODisplayBeep  EQU   -96
```

Insider Guide #22 – A Beeping Good Routine

Function Name: `DisplayBeep()`

Description: Causes a screen to flash

Call Format: `DisplayBeep(screen_address);`

Registers: a0

Arguments: `screen_address` – address of screen to flash

Return Value: None

Notes: Intuition flashes all screens if a NULL screen address is supplied



We also need to set up a text string representing the name of the intuition library. This name string needs to have a NULL (zero) at the end of the real text characters because the system routines being employed use that zero value to identify the end of the string (this is a very common convention so you should get used to it). We also need a labelled long word location to store the base address of the library in once it is open. Here are the sort of pseudo-ops which do the trick...

```
intuition_name dc.b 'intuition.library',NULL  
_IntuitionBase ds.l1
```

I'll be placing these at the end of my program. The real code – the stuff that the assembler turns into executable instructions – comes between these directives and the initial EQUate definitions. Talking of real code let's identify a suitable plan of action. We've got to load the address of the Exec library into register a6, set up the intuition library name pointer and version details, and then make an `OpenLibrary()` call as explained earlier. If the value returned in d0 is not zero then the intuition library is open.

How do we test d0 to check whether it contains a zero or not? Simple, we use a move instruction to copy the contents of d0 to the location that we've set up to hold the intuition library pointer. If the library does open successfully we need this pointer in order to perform the `CloseLibrary()` routine before the program terminates.

It's important to realise that if, for some reason, the library doesn't open then we can't use the Exec `CloseLibrary()` function because there'll be no library to close. Similarly we can't make any intuition library calls if the library didn't open. As you might guess this calls for a bit of *conditional testing* and this is done as follows: We place a `beq` instruction immediately after we stored the `OpenLibrary()` return value and branch in such a way that if the `OpenLibrary()` return value is zero then we avoid executing both the `DisplayBeep()` routine and the `CloseLibrary()` routine. As with previous examples this program is terminated with an `rts` instruction. Since we are now talking about a truly runnable Amiga with visible output, we'd better start following another convention – clearing register d0 before returning to system level. The terminal d0 state is actually used by system programs to return an error code but this is another one or those areas which we shall not be getting too involved with.

Well, you've had the theory and some explanation. All you need now is the code itself, so here it is:

* Example CH8-1.s

* uses the intuition library to 'beep' the display

```
NULL            EQU 0
_AbsExecBase    EQU 4
_LV00OpenLibrary EQU -552
_LV0CloseLibrary EQU -414
_LV0DisplayBeep EQU -96

start    move.l  _AbsExecBase, a6    get base address of
                                             Exec library
        lea    intuition_name, a1    load pointer to
                                             library name
        moveq  #0, d0                any version will do!
openlib   jsr    _LV00OpenLibrary(a6) make the indirect
                                             subroutine call
        move.l d0, _IntuitionBase    save returned pointer
        beq    exit                  did library open OK?
open_ok   move.l  #0, a0              flash all screens
        move.l  _IntuitionBase, a6    need library base in
                                             a6
        jsr    _LV0DisplayBeep(a6)    make the indirect
                                             subroutine call
        move.l  _AbsExecBase, a6    get base address of
                                             Exec library
        move.l  _IntuitionBase, a1    library to close
        jsr    _LV0CloseLibrary(a6)    make the indirect
                                             subroutine call
exit      clr.l  d0
        rts                            logical end of program

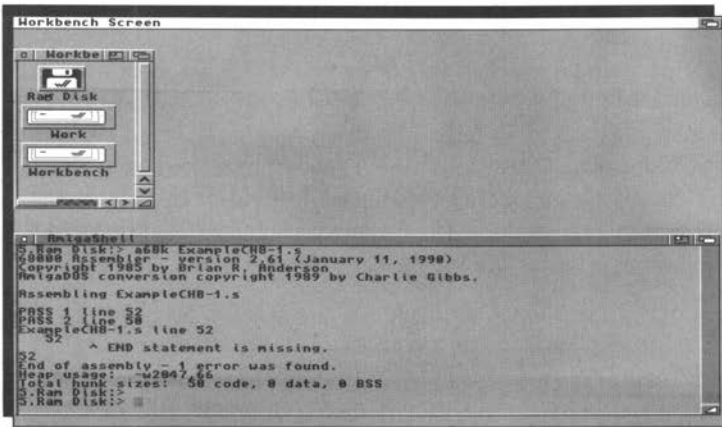
_IntuitionBase  ds.l 1
intuition_name  dc.b 'intuition.library', NULL
```

This is the smallest example of an Intuition program that can actually do anything visible. Needless to say, most Intuition programs are much larger and much more complex than this.

Having seen the code in this chapter you might be wanting to reach for your assembler program, type out the code (or just copy the ready-made file from the book's accompanying disk,), assemble, link and then run this little masterpiece for yourself. For all I know you might have tried already with either this offering, or some similar program!

It is almost inevitable, the first time an assembler package is used, that some snags crop up – call it life! To the newcomer these practical problems often seem immensely difficult to solve, usually because there's no one around to ask. This being so, perhaps a little practical assembling help wouldn't go amiss and have no fears – it's exactly this sort of advice that you'll find in the next chapter...

Now is your chance to enter the world of assembly language programming for just a couple of quid. This chapter looks at an assembler system that costs little more than the price of a disk...



To write an assembly language program you need three things: a text editor to create the source program, an assembler to convert the 68000 statements such as ADD and MOVE into the numerical form which the Amiga's processor needs, and a linker which adds the final touches by creating files that are loadable (runable). The linker can attach the WorkBench start-up code, add any library routines specified in the source code and is generally responsible for producing the final, executable, program.

ED and MEMACS are two text editors which have been provided as part of the Amiga system software for some time so all Amiga users have access to at least these text editors. On the assembler front the Amiga community has a program called A68k and if you've obtained any public domain assembly language environments, or use NorthC/PDC or other public domain C compiler offerings, then the chances are that you already have A68k. With Steve Hawtin's NorthC compiler for

example you'll find A68k and Blink, and their documentation files, in the *bin* directory. If you haven't got a copy of A68k and Blink then just get in touch with your local Amiga PD library.

Now I know what you're thinking. If this assembler environment is that cheap then there are bound to be problems with it. Right? Wrong – both A68k and Blink are excellent pieces of software and, what's more, they've been written by dedicated, professional, coders who have gone out of their way to provide robust, well-supported, products. For this we owe a debt of thanks to a number of people starting with Brian Anderson and Charlie Gibbs.

The A68k story actually starts way back in the mid 80s when the Dr Dobbs Journal published the source for a 68000 cross assembler (called X68000) written in Modula 2 by Brian Anderson. Charlie Gibbs took the X68000's ideas, translated them into C, and then used them as the basis for an Amiga assembler. After adding many enhancements (including macro and include file support and the difficult job of adding relocation information) the package we now know as A68k was born!

Nowadays A68k is available from almost all public domain libraries and you are legally entitled to copy it for free. A68k, as just mentioned, is found both as a separate package and as a component of many public domain high-level languages. Blink, the Amiga linker program, is another brilliant piece of software that stands in the Amiga's freely distributable *Hall of Fame*. It was written by a group of programmers known as the Software Distillery whose members include the likes of John Toebes and other famous names of the Amiga world.

The net result is that if you want to dive into some Amiga assembly language programming you can do it virtually for free. There is a minor stumbling block in that many of the code examples that you'll find in magazines and books will have been created using Devpac but, without detracting from the fact that Devpac is a superb 680x0 programming environment for serious users, there's no doubt that you can make a start in 68000 coding without it.

One good thing about assembly language programming is that you don't usually have much hassle in getting a published piece of code to assemble properly. Unlike the environments for languages like C the good news is that if a program can be assembled without errors on one assembler the chances are that it can be assembled by other

assemblers with little or no change – providing of course that no assembler specific statements have been used.

With A68k any changes that are necessary will in fact be minor. A68k, unlike some other assemblers, requires source files to contain an explicit END statement at the end of the source code so this may need to be added to a published listing. This is easily done by reading the source file into any available ASCII text editor, moving to the end of the text file and inserting a terminal END statement as the last line of the source code.

Both A68k and Blink are sophisticated programs and the documentation that comes with them is quite extensive – it is provided via document files which are always distributed along with these programs. There are a great many command options available but it is worth stressing that, on many occasions, only simple assembling and linking command lines are needed.

To illustrate this I'm now going to work through the steps needed to produce and run an A68k version of the program we created at the end of Chapter Eight. Rather than clutter the initial explanations with details of problems which may, in your particular case, not arise I've chosen to deal with the potential snags and pitfalls separately. If during the following assembling and linking stages you do encounter a problem just skip forward to these later sections for some additional help.

Step One – Opening a Shell Window

ED, the Amiga text editor that we'll use to create the source file, A68k, Blink and the final program that is going to be created, are all Shell based programs so before we can do anything a Shell window must be opened. Open your Workbench system drawer and double click on the Shell icon. When the window opens you see this sort of prompt:

Workbench:>

indicating that Workbench is the current directory. For convenience we want to use the Ram disk for our assembling operations so reset the current directory by typing:

cd ram:

after the prompt. This now changes to:

Ram Disk:>

At this stage it is convenient to copy the A68k and Blink programs to the Ram Disk so that they are ready for use.

Step Two – Creating the Source File

To start ED from the Shell window type ED followed by the name of the program to be edited or created. Assembler source files, by convention, always have a .s filename extension so I'm going to call the file "ExampleCH9-1.s" and enter this command at the Ram Disk:> prompt:

```
Ram Disk:>ED ExampleCH9-1.s
```

Feel free to call the program "test.s", or something similarly short in order to save yourself some typing – I've only used ExampleCH9-1.s for consistency since this is the name of the equivalent program as stored on the accompanying Insider Guide disk.

With ED up and running all you now need to do is type in the source code given at the end of Chapter Eight but add an additional END statement to the listing. Keep to similar field placements but again, to save typing, you do not need to include the comment lines or end of line remarks. You should end up with a file looking something like this:

```
* Example CH9-1.s
```

```
_AbsExecBase EQU 4  
NULL EQU 0  
_LV0OpenLibrary EQU -552  
_LV0CloseLibrary EQU -414  
_LV0DisplayBeep EQU -96  
start move.l _AbsExecBase,a6  
lea intuition_name,a1  
moveq #0,d0  
openlib jsr _LV0OpenLibrary(a6)  
move.l d0,_IntuitionBase  
beq exit  
open_ok move.l #0,a0  
move.l _IntuitionBase,a6
```

```
        jsr        _LV0DisplayBeep(a6)
        move.l    _AbsExecBase,a6
        move.l    _IntuitionBase,a1
        jsr        _LV0CloseLibrary(a6)
exit    clr.l     d0
        rts
_IntuitionBase ds.l 1
intuition_name dc.b 'intuition.library',NULL
        end
```

Check that you've entered the instructions correctly, select Save from the ED Project Menu, and then quit the program (which returns you to the Shell prompt). It is a good idea at this stage to make a permanent copy of the newly created source code file on a floppy or hard disk just in case something goes wrong when you run the finished program.

Step Three – Assembling the Example Code

The created source file above is called ExampleCH9-1.s and is present in the Ram Disk. The Shell command line needed to assemble the program is

```
Ram Disk:>A68k ExampleCH9-1.s
```

The case of the letters is not important and you could just have well have entered

```
Ram Disk:>a68k examplech9-1.s
```

Either way A68k looks for the source code file, assembles it, and creates an intermediate object code module. By default this has the same name as the specified source file but it with a '.o' (for object file) extension. If you want to produce an object file with a name different to its source file – eg test.o – then it is necessary to add an additional command line parameter using a -o prefix like this

```
Ram Disk:>A68k ExampleCH9-1.s -otest.o
```

One way or the other then A68k produces an object file on the Ram disk which can subsequently be linked using Blink.

Step Four – Linking

In the simple case where no start-up code or linker libraries need to be specified on the

Blink command line we just have to type

Ram Disk:>blink ExampleCH9-1.o

and this results in a Shell executable program called ExampleCH9-1 being placed on the Ram disk. Blink, by default, creates an executable program whose name is the same as the supplied object code module but with the '.o' suffix removed. If you want to specify some alternative name Blink recognises a TO keyword that can be used in conjunction with a suitable destination filename. So

Ram Disk:>blink ExampleCH9-1.o to test

produces an executable file called test.

Step Five – Preparing for the Worst

At this stage of the proceedings a runnable program is sitting in the Ram disk. Any number of trivial slips might result in this program crashing your machine so, before you run it, take these precautions:

- 1 Check that you have a backup copy of the source code – on floppy or hard disk.
- 2 Take out any floppies from your disk drive unless they are write protected. Writable disks have been known to get corrupted during a crash and it is better to be safe than sorry.

Step Six – Go Go Go

Well, it's now or never. Having taken the above mentioned precautions type the name of your

masterpiece at the Shell window

Ram Disk:>ExampleCH9-1

With luck the Workbench screen will flash – nowadays there will also be an audible noise if this has been selected in the Workbench Preferences settings. Now I know that, as programs go, this is hardly what you'd call an earth shattering piece of code. Nonetheless, if you were able to write, assemble and link this example first time you should congratulate yourself – you've been far luckier than some!

If Things Have Gone Wrong

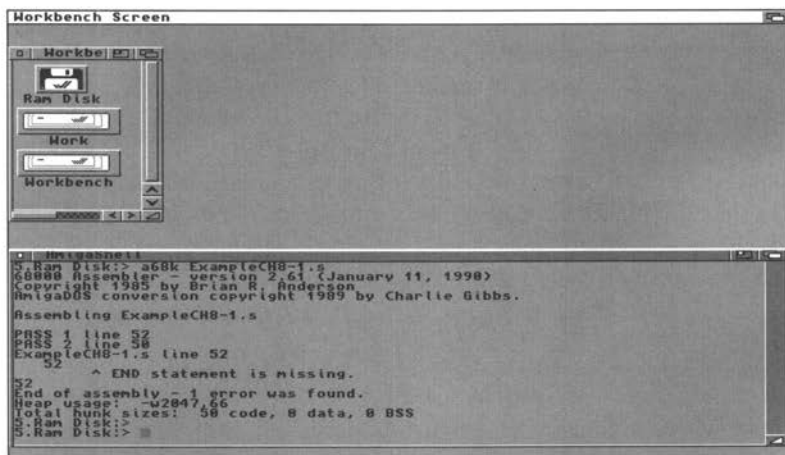
Ignoring the trivial slips which can occur, such as forgetting

to copy A68k and Blink to your Ram Disk it is at the assembling stage where error messages are first seen. Always look very carefully at the first error message because it always indicates a fault in the code. With the second and subsequent errors this may not always be the case – rogue error messages are occasionally produced because an earlier error caused the assembler to misinterpret later, valid, instructions. When an error is found A68k, (where appropriate) displays the source line along with an error position ^ indicator and error message. Most of the time these error messages are self explanatory as you'll see from the following examples.

END statement is missing Error

This means exactly what it

says. A68k, as mentioned earlier, expects to see an explicit END statement at the end of a source file and you've forgotten to include one. Re-edit the source code, move to the end of the file and insert an END (or end) statement remembering to indent the command – placing it in the instruction field – otherwise a68k will think you have added another label to your program!



```
Horkbench Screen
Horkbe
Ram Disk
Work
Workbench
-----
AmigaDOS
C:\Ram Disk:> a68k ExampleCH8-1.s
68000 Assembler - version 2.61 (January 11, 1990)
Copyright 1985 by Brian R. Anderson
AmigaDOS conversion copyright 1989 by Charlie Gibbs.
Assembling ExampleCH8-1.s
PASS 1 line 52
PASS 2 line 58
ExampleCH8-1.s line 52
52      ^ END statement is missing.
52
End of assembly - 1 error was found.
Heap usage: -w2847,66
Total disk sizes: 58 code, 8 data, 8 BSS
C:\Ram Disk:>
C:\Ram Disk:>
```

Undefined Symbol Errors

Undefined symbol errors suggest that you've either forgotten to define a required symbol completely, or have defined but mis-spelt it. If, in our example program, you typed `AbsExecBase` instead of `_AbsExecBase`, forgetting to include the initial underscore, the A68k assembler would rightly complain.

Error In Operand Format

This tells you that the instructions operand format is wrong but as well as true faults of this kind (which obviously you must examine and correct) you'll also get this error message if additional blanks have been included in the operand field. These latter slips are often harder to find because there is nothing really wrong with the overall code line format. As an example a line which reads

```
move.l.  _AbsExecBase, a6
```

assembles correctly but a single blank separating the two operands "`_AbsExecBase`" and "`a6`":

```
move.l.  _AbsExecBase, a6
```

causes A68k to complain. As always the remedy is to re-edit the source code so as to correct the mistake, and then reassemble it.

Linking Errors

With only one source code file and no include files to worry about the only linker error likely to be found is "Cannot find object ExampleCH9-1.o". This is most

likely to occur because you've either got the filename wrong or because the object file was never created in the first place. This latter case would occur if A68k found an error but you failed to spot the error message on the screen.

Program Fails To Run As Expected

In getting to the point where a runnable program was produced A68k and Blink have confirmed that the source code file is *syntactically correct*. Unfortunately the program can still be a far cry from being bug free.

If, for example the program line that reloads `a6` with the Exec library base address prior to making the final `CloseLibrary()` function call

```
move.l  _AbsExecBase,a6
```

was inadvertently missed out so that the section that starts with "open_ok" reads as

```
open_ok  move.l  #0,a0
         move.l  _IntuitionBase,a6
         jsr    _LVODisplayBeep(a6)
         move.l  _IntuitionBase,a1
         jsr    _LVOCloseLibrary(a6)
```

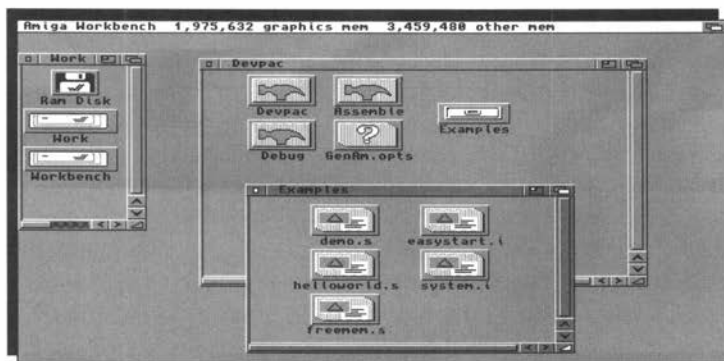
then some very serious problems arise because now, at the time the CloseLibrary() function is executed, a6 is holding the Intuition library base rather than the Exec library base. In short the code ends up trying to execute some wholly inappropriate Intuition library function (obviously this is the one which has the same LVO value). In addition to this the Intuition library never ever gets closed.

The result is that the program performs the required DisplayBeep() operations and then crashes the machine immediately afterwards, requiring you to re-boot. Similar errors, such as using the wrong 68000 registers when making library calls, cause other problems. In all cases the solution is easy – look at the code you've written, compare it with what you should have written, correct the differences and repeat the assembly/linking processes again.

Finding errors in this way is something that comes with practice. Not to worry, all assembler programmers get plenty of practice with this, especially in their early days.

It's worth mentioning at this stage that two of the most frequently made slips are firstly failing to set up properly those parameters needed by a function call (ie not loading the appropriate 68000 registers with the data needed by the library routine), and not checking that the returned values are valid (especially important with things like library bases).

HiSoft's 680x0 Devpac Amiga Assembler package has been around for a long time forming a large user-base. It is popular because it does the job that it is supposed to do and it is stable and well supported.



The latest version of Devpac – Devpac 3 – has been designed with Workbench 2/3 users in mind. The package is, however, currently shipped with an additional 1.3 based version as well. Quite simply Devpac 3 is just about as good an assembler package as you could wish for.

An Integrated Environment

Devpac comes with its own editor and this, for most operations, acts as a main controller for the whole of the Devpac development environment. The editor offers multiple-file editing with full mouse-controlled cut-&-paste facilities and you can open individually scrollable multiple windows on the same file. This means that whilst working on one program you can open other files and copy pieces of existing code (such as standard routines, text notes and so on). There are a host of other useful extras including bookmark set and locate facilities and macro recording facilities for memorising complex key press sequences.

Devpac
comes with
its own edi-

There are some powerful assembler/debugger options and one of the big advantages with the Devpac editor is that it not only integrates these tools into the editor environment but provides facilities for the automatic location of errors in the source after assembly. Create the source code using the editor and select "Assemble" from the program menu. Edit/assemble until the assembly process is error free and you can then run the code directly from the editor's program menu. In short it is possible to create, assemble, debug, run and save your code all from the same environment!

An editor settings menu allows you to set the editor and assembler controls and define the usual types of global settings for tab size, end-of-line behaviour, auto indenting, automatic back-up creation and so on. The assembler options themselves are grouped into three separate requesters which are called up by selecting one of three items on the assembler settings sub-menu. A control requester provides control over basic assembler operation, source and destination file paths, listing control etc. The Options requester gives access to the large number of more technical assembler settings – identifying processor, coprocessor and MMU types, ensuring PC-relative code, producing local label underscoring and so on. The third requester provides a range of assembler optimisation settings.

Devpac supports the 68000-68040, 68332, 68881/2 and the 68851 memory management unit (MMU) chips. The assembler has all the *bells & whistles* expected of a modern day offering – it's a macro assembler which provides comprehensive expression handling and supports *, /, +, -, =, bitwise and/or/xor/not, left and right shifting and all the usual inequality operators. Like many assemblers it allows decimal, hex, octal, binary and character constants but it also offers some more specialised facilities such as floating point constants for 68881/2 coprocessor applications. Devpac allows the use of local labels and, by default, all label names are significant to 127 characters.

Devpac's debugger is called MonAm and it's a low-level debugger able to step through a program displaying code instructions, 68000 register contents, processor status, and memory contents in hex or ASCII form. If you have included debug info in your program then MonAm can use that to display your original program labels. The debugger can also be used to look at compiler written code and if the package that produced the code included line number debug data it is even possible to view the original source code. MonAm is very powerful but, having

said that, it does take a bit of time to learn how to use it effectively and it is not quite as user-friendly as the rest of the Devpac package.

As well as the editor, assembler and debugger the Devpac 3 package includes Blink – the Amiga's de facto standard linker – a program called SRSplit, which is an S-record splitter utility, and a utility called FD2LVO which converts Commodore FD files into include files containing direct *library vector offset* data (LVO values). You also get those all-important Commodore assembly language include files which are covered in Chapter 13, link libraries and some example programs to get you started.

This book is in no way restricted to Devpac users but it must be said that, if you have yet to get an assembler package, Devpac 3 is worthy of serious consideration. As software goes it provides good documentation along with some superb facilities so newcomers get an assembler environment which will help make learning about, and using, assembly language just about as easy as it ever could be!

The following sections provide a step-by-step account of assembling and linking the same ExampleCH8-1.s program dealt with in the last chapter. This time we're using the Devpac environment rather than A68k's Shell based methods.

Step One – Starting Devpac

Devpac works perfectly well with either floppy or hard disk based systems. Like most packages it comes with an automatic installation program for hard disk users. Once initially installed all you have to do is double-click on the Devpac icon to bring up the editor display.

Step Two – Creating the Source File

As in the previous chapter we've now got to either type in the source code or, if you prefer, load the pre-written disk form from the Insider Guide disk. If you choose the latter option then all you now have to do is select Load from the editor's Project menu. Then, when the standard Amiga requester appears, identify and load the appropriate source.

If you prefer to type the program in (which helps you to get a feel for what assembly language coding is all about) do it now and remember that, as with the Chapter Nine material, you can save some typing by

leaving out the comments. I've repeated the code here so that you don't have to refer back to the earlier chapters.

*** Example CH10-1.s**

```
NULL          EQU    0
_AbsExecBase  EQU    4
_LV0OpenLibrary EQU   -552
_LV0CloseLibrary EQU  -414
_LV0DisplayBeep EQU   -96

start         move.l   _AbsExecBase, a6
              lea     intuition_name, a1
              moveq   #0, d0

openlib       jsr     _LV0OpenLibrary(a6)
              move.l   d0, _IntuitionBase
              beq     exit

open_ok       move.l   #0, a0
              move.l   _IntuitionBase, a6
              jsr     _LV0DisplayBeep(a6)
              move.l   _AbsExecBase, a6
              move.l   _IntuitionBase, a1
              jsr     _LV0CloseLibrary(a6)

exit          clr.l   d0
              rts

_IntuitionBase ds.l 1
intuition_name dc.b 'intuition.library', NULL
```

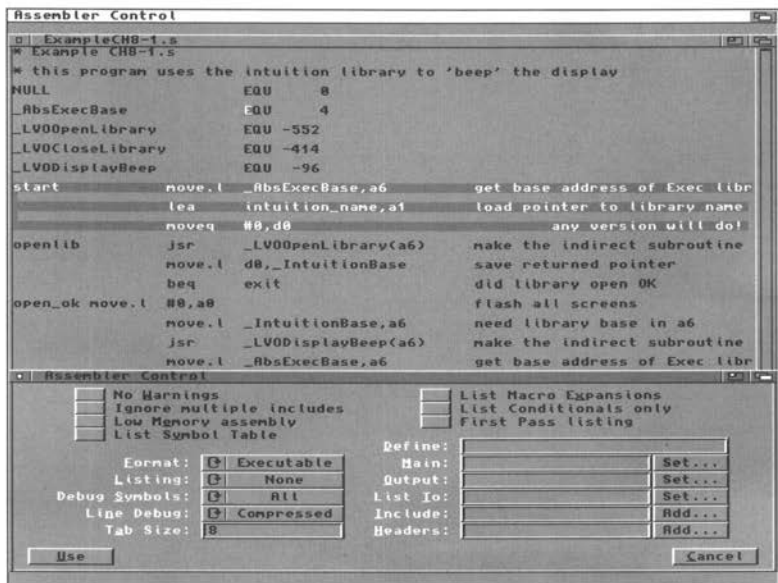
At this stage you'll want to save the source code and you can do this first time by selecting "Save As" from the project menu which brings up a file save requester. Assembler source files, by convention, always have a ".s" filename extension so call the file something like ExampleCH10-1.s. Again, feel free to call the program test.s, or something similarly short in order to save yourself some typing - I've only used ExampleCH10-1.s for consistency. It is the name of the equivalent program as stored on the accompanying Insider Guide disk. It is a good idea at this stage to make a back-up copy of the newly created

source code file, just in case something goes wrong when you finally run the finished program.

Step Three – Assembling and Linking the Example Code

Go to the editor's Settings menu and toggle select the Assemble To Disk option.

Then, from the same menu, select the "Assembler Control" option to open a window showing the current assembler settings. A Format box should be showing Executable – if it isn't change it so that it does. By clicking on this field it can be changed between Executable, Linkable or S-Records.



Having done that move to the other side of the requester to the near-bottom right field called Headers. This specifies the header files that Devpac uses during assembly. Move the mouse to within the text box and then keep hitting the back-space key until the default name is removed as this header isn't needed for our example.

By selecting Executable you've told Devpac to create an executable program directly so it is not necessary to link the assembled program explicitly. All you have to do to produce a runnable program is to select "Assemble" from the editor's Program menu. Do it now!

Step Four – Preparing for the Worst

At this stage of the proceedings a runnable program is available. As mentioned before any number of trivial slips might result in this program crashing your machine so, before you run:

- 1 Check that you have a backup copy of the source code on floppy or hard disk.
- 2 Take out any floppies from your disk drive unless they are write protected.

Step Five – Go Go Go

It's now or never. Having taken the above mentioned precautions go to the Editor's

Program menu and select Run. With luck the Workbench screen will flash.

```

Workbench Screen
ExampleCH8-1.s
Example CH8-1.s
* this program uses the intuition library to 'beep' the display
NULL EQU 0
_AbsExecBase EQU 4
Assembling...
Hem! Macro Assembler Copyright © HiSoft 1985-91
All Rights Reserved - version 3.81
Assembling ExampleCH8-1.s
Pass 1
Pass 2
0 errors found
49 lines assembled into 86 bytes, Amiga executable relocatable code
23376 bytes used

; ... _intuition.library; ... make the 'intuition' sub-object
exit clr.l d0
rts
intuition_name dc.b 'intuition.library', NULL
_IntuitionBase ds.l 1
  
```

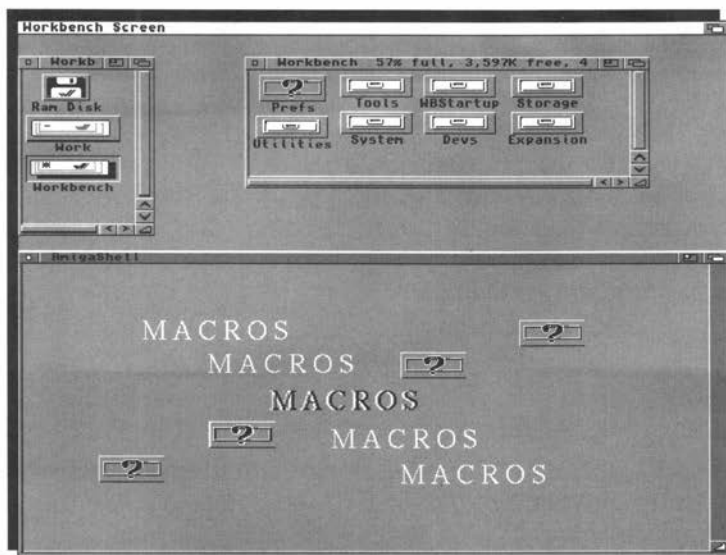
- By default Devpac gives the runnable program the same name as your source code but with the .s extension removed. You can, if you wish, specify some alternative output file name.

If Things Have Gone Wrong

If Devpac doesn't like, or cannot understand your source file it displays one or more error messages. The key is again to look very carefully at the first error message because it invariably indicates a genuine fault. With the second and subsequent Devpac flagged errors this may not always be the case. As with other assemblers Devpac occasionally generates rogue messages produced because some early error caused the assembler to misinterpret later, valid, instructions. When an error is found Devpac (where appropriate) displays the source line along with an error message. With this environment when you cancel the Assembly Details display window you are returned to the editor at the point where the first error occurred. This can be corrected immediately and using the Program menu's Find Error, Next Error and Previous Error options, you can quickly identify (and therefore correct) any errors in the source.

The types of errors that can occur are of course exactly the same as those mentioned in Chapter Nine so, if you do hit any error message snags, have a look at that material.

To create code that fits the Amiga 68000 programming style it's as well to know a little about how macros are created and used. Why? Because they can hide some of the complexity of assembly language programming!



In 68000 assembly language, as with any other computer language, you frequently find that similar sequences of instructions crop up again and again. With sequences that are identical one solution is to write the instructions as a subroutine rather than waste space by having the same instructions duplicated in various places throughout the program. The subroutine approach reduces program size and has a number of benefits as far as program structure is concerned but there are still times when inserting duplicate sections of code is necessary - eg to eliminate the time in calling the subroutine. Subroutines are often inappropriate simply because the various sequences of instructions are only similar and not completely identical.

Macros provide a solution to this dilemma because they allow the programmer to assign symbolic names to sets of instruction sequences and, when the name is encountered, the assembler automatically expands it to produce the original set of instructions. This facility is not restricted to predefined, absolutely fixed, instruction sequences. Macros which contain parameter placeholder markers can be created so that, when the macro is used, parameters provided with each particular use instance are inserted into the code that is generated. This makes it possible for the macro programmer to generate a variety of code fragments from each macro definition.

Macro Definitions

A68k, Devpac and most other Amiga assemblers tend, for obvious reasons, to support the standard

Motorola style 68000 macro definitions. These start with a label followed by the **MACRO** keyword and end with the **ENDM** keyword. Lower case **macro** and **endm** are also accepted but, to my mind, the upper case versions mark the macro segment more clearly. The basic 68000 macro takes this type of form:

```
my_macro_name    MACRO
                   <main body of macro code>
                   ENDM
```

Parameters are specified using the backslash(\) character followed by an any alphanumeric character. The best way of coming to terms with these, extremely useful, code units is to see a couple of examples. I've chosen two that are directly related to some of the assembler code you've seen earlier in the book.

The LINKLIB Macro

Let's start by giving you a macro definition to look at:

```
LINKLIB    MACRO
            move.l    a6, -(sp)
            move.l    \2, a6
            jsr       \1(a6)
            move.l    (sp)+, a6
            ENDM
```

The first and last real code instructions are exactly that, just perfectly ordinary 68000 move statements which copy a 32 bit long word value from, and to, register a6. The “sp” term means stack pointer and most assemblers allow this to be used as another way of specifying register a7. Thus the above macro could equally have been written as

```
LINKLIB   MACRO
           move.l      a6, -(a7)
           move.l      \2, a6
           jsr         \1(a6)
           move.l      (a7)+, a6
           ENDM
```

The first and last move instructions are doing something which I've talked about in general terms but have yet to provide an example of – pushing and retrieving a data item from the 68000's user stack. Very often you will want to protect a microprocessor register from being inadvertently altered whilst you execute some other series of operations. One easy way to do this is to copy temporarily the item in question to the 68000's stack, and retrieve it when it is safe to do so. It is common practice for a subroutine to save the original contents of any registers it intends to use and re-instate them just before it returns to the main program (ie just before the terminal rts instruction).

The 68000's stack pointer register always points to the last item stored in the stack area. These stacks grow downwards in memory and so each time a number of bytes are copied to the area of memory being used for the stack, the stack pointer has to be updated. The important thing here is to recognise that the pointer must be decreased *before* any new data is stored on the stack otherwise the existing last item would be overwritten with the new data. When getting data from the stack the reverse convention has to be followed – data is retrieved *after* the stack pointer has been updated.

The 68000 actually has special addressing modes, called indirect pre-decrement and postincrement addressing, which allow these adjustments to be done automatically. Don't worry if the ideas seem a bit strange at first – just mimic the way the instructions are used in your own code and the understanding will doubtless come in due course.

The predecrement mode, which is always used when storing values on the stack, uses an initial minus (-) sign

```
move.l    a6, -(a7)
```

The postincrement form uses a trailing plus (+) sign

```
move.l    (a7)+, a6
```

These types of preserve/restore operations must always be done in pairs otherwise both you, and the 68000 processor will lose track of the data that is coming onto, or being taken off, the stack.

If you take a look at the two innermost macro instructions you will see that they are almost like real 68000 instructions but, in both cases, there is an operand missing. Instead there is a backslash followed by a number

```
move.l    \2, a6
jsr       \1(a6)
```

These represent parameter placeholders and what happens is that these operands get filled in at assembly time using values that you've supplied. If, for instance this macro was to be used in conjunction with the following program line

```
LINKLIB  _LVODisplayBeep, _IntuitionBase
```

the assembler would automatically generate this sequence of instructions:

```
move.l    a6, -(sp)
move.l    _IntuitionBase, a6
jsr       _LVODisplayBeep(a6
move.l    (sp)+, a6
```

The net effect is that register a6 is preserved on the 68000's stack whilst the register is re-loaded with some specified value – which is in practice a library base. Then the specified library call is made using the now familiar indirect subroutine technique. Lastly register a6 gets re-instated with its original value. Note that the macro protects the user by preventing a6 from being inadvertently changed!

This particular macro, with some additional error checking code, is actually already present in one of the Amiga system's include files and it is used to generate library access code.

Macros Within Macros

Macro definitions can even be nested – ie a macro definition can include other

macro definitions. Here's an example which tags on the extra `_LVO` characters to the function name.

```
CALLSYS   MACRO
            LINKLIB      _LVO\1,\2
            ENDM
```

You can see from the above definition that `CALLSYS` adds an `_LVO` Prefix onto the first of the two required parameters. Instead of using `LINKLIB` like this

```
LINKLIB _LVOOpenLibrary,_AbsExecBase
```

you can write

```
CALLSYS OpenLibrary,_AbsExecBase
```

and most programmers regard this latter form as being easier to read.

The Underlying Magic

If you include the above `LINKLIB` and `CALLSYS` macros in your code you'll

be able to create the appropriate library opening code using these types of simplified statements:

```
CALLSYS OpenLibrary,_AbsExecBase
```

To appreciate the benefits of macros you need to see exactly what results these macros can produce relative to a piece of code that doesn't use them. I should apologise for re-using the Chapter Eight *Intuition Beeping* example again but by now you should be familiar with it so the changes made in this chapter should be both easy to follow and easy to understand

* **Example CH11-1.s**

* **a macro style version**

```
LINKLIB   MACRO
            move.l    a6,-(a7)
            move.l    \2,a6
            jsr       \1(a6)
            move.l    (a7)+,a6
```

```
                ENDM
CALLSYS        MACRO
                LINKLIB _LV0\1,\2
                ENDM

NULL          EQU 0
_AbsExecBase  EQU 4
_LV0OpenLibrary EQU -552
_LV0CloseLibrary EQU -414
_LV0DisplayBeep EQU -96

start        lea    intuition_name,a1    load pointer to
                                                library name
                moveq #0,d0              any version will do!
openlib      CALLSYS OpenLibrary,_AbsExecBase
                move.l d0,_IntuitionBase  save returned pointer
                beq    exit                did library open OK?
open_ok      move.l #0,a0                flash all screens
                CALLSYS DisplayBeep,_IntuitionBase
                move.l _IntuitionBase,a1  library to close
                CALLSYS CloseLibrary,_AbsExecBase
exit         clr.l  d0
                rts

_IntuitionBase ds.l 1
intuition_name dc.b "intuition.library",NULL
```

The result is simpler, cleaner, source code and this is exactly the type of benefit which macros provide along with a certain amount of standardisation. Since macro code tends to get re-used and, since code that is frequently re-used quickly becomes bug free, programmers who make maximum use of macro facilities tend to make fewer coding errors.

Macros resemble subroutines in the sense that they provide a shorthand reference to a frequently used set of instructions. However, it

should be clear from the above discussion that macros are *not* subroutines. The code for a subroutine only occurs once within a program, and program execution branches to the subroutine. On the other hand, each time a macro is used the assembler inserts a copy of the appropriate instructions with any parameter-specified alterations.

Header Files

There's nothing wrong with putting your macros in the same file as your source code but most macros, by their very nature, are used over and over again. Because of this it's actually convenient to put these units in their own separate file, called a 'header file' or 'include file', and usually given a .i filename extension - i for include. You can then ask the assembler to include this file when it assembles the main program code.

*** Example CH11-2**

*** macros.i - a typical macro definitions file**

```
LINKLIB    MACRO
            move.l    a6, -(a7)
            move.l    \2, a6
            jsr      \1(a6)
            move.l    (a7)+, a6
            ENDM
CALLSYS    MACRO
            LINKLIB  _LVO\1, \2
            ENDM
```

Asking the Assembler to Include Another File

The assembler can be instructed to include some other file, such as the macro definition file

mentioned above, by adding an INCLUDE directive statement to your main program:

```
INCLUDE macros.i
```

Both A68k and Devpac support this type of statement and both provide additional facilities for defining the directories to be searched when looking for such files. With A68k there is a -i<include directory list> option. With Devpac several options are provided including the

addition of directory names via the Editor menu's Assembly requester. I'll be talking more about include files later (in Chapter 13) but the benefits of having standard, and often used, definitions kept separate in this way should already be reasonably obvious.

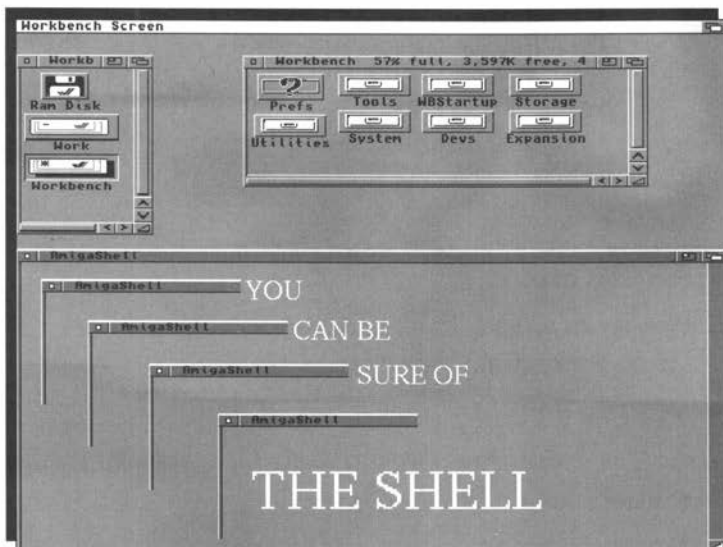
To finish this chapter here is our, now somewhat overworked, example program with a newly added INCLUDE statement

*** Example CH11-3.s**

```

                                INCLUDE macros.i
NULL                            EQU 0
_AbsExecBase                    EQU 4
_LV0OpenLibrary                 EQU -552
_LV0CloseLibrary                EQU -414
_LV0DisplayBeep                 EQU -96
start    lea    intuition_name,a1    load pointer to
                                library name
                                moveq #0,d0        any version will do!
openlib  CALLSYS OpenLibrary,_AbsExecBase
                                move.l d0,_IntuitionBase    save returned pointer
                                beq    exit            did library open OK?
open_ok  move.l #0,a0                flash all screens
                                CALLSYS DisplayBeep,_IntuitionBase
                                move.l _IntuitionBase,a1    library to close
                                CALLSYS CloseLibrary,_AbsExecBase
exit     clr.l  d0
                                rts
_IntuitionBase ds.l 1
intuition_name dc.b "intuition.library",NULL
```


Writing programs which interact with a Shell window can provide an easy way into the world of larger assembly language programs. Best of all, a lot can be achieved with a few ready made DOS library function calls.



In order to read data from, or write data to, a Shell window a program obviously needs to know something about the DOS I/O environment. More specifically it needs to obtain the two I/O handles which C programmers conventionally call *stdin* and *stdout*. When used within assembly language programs these handles are more usually defined as *_stdin* and *_stdout*.

Regard what happens at the DOS level as magic for the moment and just accept that, in order to collect these I/O handles, the Shell program just has to open the DOS library, and then make calls to two DOS functions known as *Input()* and *Output()*. Opening, and using, the DOS library is no different to opening any other run-time library and so the code required will follow the same pattern as we've used already.

***Insider Guide #23 –
Collecting The Standard Input Handle***

Function Name: Input()

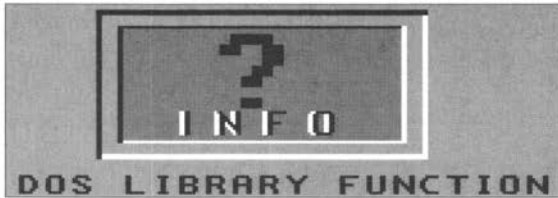
Description: Identify a program's initial input file handle

Call Format: file_handle = Input()

Registers: d0

Arguments: None

Return Value: file_handle - the program's initial input file handle



***Insider Guide #24 –
Collecting The Standard Output Handle***

Function Name: Output()

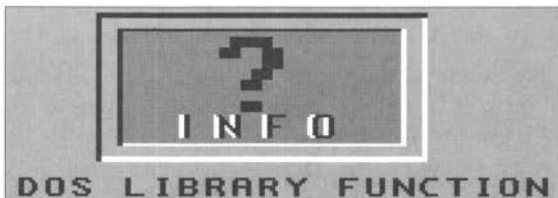
Description: Identify a program's initial output file handle

Call Format: file_handle = Output()

Registers: d0

Arguments: None

Return Value: file_handle - the program's initial output file handle



Insider Guide #25 – Another Useful DOS Function

Function Name: Write()

Description: Write data to a file

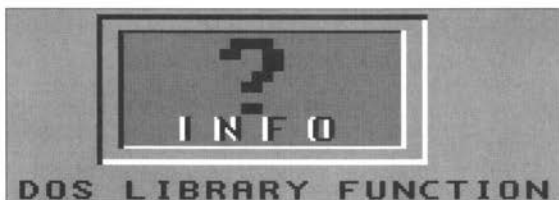
Call Format: length_written =
Write(file, buffer_p, data_length)

Registers: d0
 d1 d2 d3

Arguments: file – file handle
 buffer_p – pointer to buffer
 holding the data
 data_length – length of the
 data

Return Value: length_written – number of bytes
 actually written

Notes: A length_written value of -1
 indicates an error



The standard name for the DOS library base is `_DOSBase`, and so the required library opening code takes this form:

```
lea      dos_name,a1      library name start in a1
moveq   #0,d0            any version will do
CALLSYS  OpenLibrary,_AbsExecBase
move.l  d0,_DOSBase     store returned value
```

In the final runnable example program, we need to check that the returned library base is valid. As with previous library code examples this is done by checking the zero flag after the library base (which comes back in register `d0`) has been copied to a suitably defined storage location.

If `OpenLibrary()` was successful then the `DOS Input()` and `Output()` functions can be used to identify the program's input and output handles. For the purposes of the example developed in this chapter I'm going to deal only with the task of outputting messages, so only the output handle needs to be obtained. This is achieved using this sort of code:

```
CALLSYS   Output,_DOSBase   get default output handle
move.l   d0,_stdout        store output handle
```

Again we need, in the final program, to check that the returned `d0` value is valid (ie is non zero).

Writing Text

Writing Shell text messages is an easy task because there is a general DOS function, called `Write()`, which can do the job. You can

see from the description of the routine that we need to know how long each text string is. There is a sneaky way of getting this without having to count the characters manually!

Static program text is, as we've seen in earlier chapters, usually set up using `define byte (dc.b)` assembler directives:

```
message dc.b "Just an example message"
```

All 680x0 assemblers, and most others come to that, allow the programmer to use an asterisk (*) to represent the current value of the assembler's location counter – remember that this is how the assembler keeps track of its current position whilst assembling a source file. By placing an additional label at the end of the text, and using another `EQUate` directive to set it to a value based on the current assembler location counter value we can get the assembler to work out the length of the text string:

```
message dc.b "Just an example message"
message_SIZEOF EQU *-message
```

The result is that the assembler automatically sets the second label to the size of the preceding string. I adopt a convention whereby the sizes of all message strings are represented by a label formed by taking the original string label and adding `_SIZEOF` to it. Why? Because it is then possible to create a macro that, given the string label, can form the size label automatically. I won't be doing this because, at the moment, it is probably more instructive to see the new function call being explicitly set up and used.

A First Coding Stage

The following example deals with issues which should, in the main, either be familiar to

you or relatively straightforward to understand given the previous discussions. The DOS library is opened and `_stdout` is set up then the DOS library is closed. For simplicity I've placed the `LINKLIB` and `CALLSYS` macros in the same file as the main source code. If however you are now happy about using a separate macro include file then by all means rearrange the source accordingly.

*** Example CH12-1.s**

```

LINKLIB      MACRO
              move.l    a6, -(a7)
              move.l    \2, a6
              jsr       \1(a6)
              move.l    (a7)+, a6
              ENDM

CALLSYS      MACRO
              LINKLIB  _LVO\1,\2
              ENDM

NULL        EQU      0
_AbsExecBase EQU      4
_LV0OpenLibrary EQU    -552
_LV0CloseLibrary EQU   -414
_LV0Output  EQU      -60
_LV0Write   EQU      -48

start       lea        dos_name, a1      load pointer to library
                                                name
              moveq    #0, d0           any version will do!

openlib     CALLSYS   OpenLibrary, _AbsExecBase
              move.l   d0, _DOSBase     save returned pointer
              beq      exit            did library open OK?

dos_open    CALLSYS   Output, _DOSBase  get _stdout handle

```

```

        move.l    d0,_stdout      store it
        beq      close_dos      check _stdout is valid
*
        HERE'S WHERE THE MESSAGE
*
        WRITING CODE WILL FINALLY GO
close_dos move.l    _DOSBase,a1    library to close
        CALLSYS  CloseLibrary,_AbsExecBase
exit     clr.l    d0
        rts
_DOSBase ds.l 1
_stdout  ds.l 1
dos_name dc.b 'dos.library',NULL

```

Write()-ing The Message

The DOS Write() function needs quite a lot of parameters. Namely an output handle in register d1, the address of the start of the message in register d2, and the size of the message in register d3. These aren't difficult to set up and simple move.l instructions can do the job

```

move.l    _stdout,d1          standard output handle
move.l    #message,d2       start of message
move.l    #message_SIZEOF,d3 message length

```

Having set up all the necessary function parameters we make the library call in the usual fashion:

```

move.l    _DOSBase,a6      set a6 to DOS library base
jsr      _LVOWrite(a6)    make DOS Write() call

```

All that remains to be done is to put these new ideas into the framework we've already established. Here's the final result:

```

* Example CH12-2.s
LINKLIB  MACRO
        move.l    a6,-(a7)
        move.l    \2,a6
        jsr      \1(a6)
        move.l    (a7)+,a6

```

```
                ENDM
CALLSYS        MACRO
                LINKLIB _LVO\1,\2
                ENDM

NULL          EQU    0
LF            EQU    10
_AbsExecBase EQU    4
_LV0OpenLibrary EQU   -552
_LV0CloseLibrary EQU  -414
_LV0Output    EQU   -60
_LV0Write     EQU   -48

start        lea    dos_name,a1      load pointer to library
                                                name
                moveq #0,d0          any version will do!

openlib      CALLSYS OpenLibrary,_AbsExecBase
                move.l d0,_DOSBase   save returned pointer
                beq    exit          did library open OK?

dos_open     CALLSYS Output,_DOSBase  get _stdout handle
                move.l d0,_stdout    store it
                beq    close_dos     check _stdout is valid

write_text   move.l _stdout,d1        standard output handle
                move.l #message,d2    start of message
                move.l #message_SIZEOF,d3 message length
                CALLSYS Write,_DOSBase

close_dos    move.l _DOSBase,a1      library to close
                CALLSYS CloseLibrary,_AbsExecBase

exit         clr.l  d0
                rts

_DOSBase     ds.l  1
_stdout      ds.l  1
dos_name     dc.b 'dos.library',NULL
```

```
message dc.b 'Just an example message',LF
message_SIZEOF EQU *-message
```

Variety Is The Spice Of Life

You've had the details. Now it's your turn – so try

and sketch out a program which will print these three lines of text:

```
2YUR2YUBICUR2Y4ME
```

or in other words...

Too wise you are, too wise you be, I see you are too wise for me!

You need to set up three text messages (call them message1, message2 and message 3) and make three DOS Write() calls. Place a Linefeed (use a LF EQU 10 expression) at the end of each text string so that each piece of text is printed on a separate line.

On the Amiga registers a0, a1, d0 and d1 are designated as *scratch registers* and library functions can, and often do, overwrite these without warning. Because of this you need to reload register d1 with the `_std` output handle every time you perform a Write(). However, the DOS library base pointer can be safely re-used as register a6 is not altered by any library calls.

Here's the sort of result you should have obtained:

* Example CH12-3.s

```
LINKLIB      MACRO
              move.l    a6, -(a7)
              move.l    \2, a6
              jsr      \1(a6)
              move.l    (a7)+, a6
              ENDM

CALLSYS      MACRO
              LINKLIB   _LVO\1, \2
              ENDM

NULL          EQU      0
LF            EQU      10
_AbsExecBase EQU      4
```


Insider Guide #26 – More Macro Help

Macro names are labels and, just like any other labels, they must be placed in such a way that the assembler recognises the name as a label. MACRO and ENDM on the other hand are assembler directives and these must be placed in the source code's instruction fields. Problems arise if you're not careful with these type of field placements. For example, the definition:

```
LINKLIB    MACRO
            move.l    a6, -(a7)
            move.l    \2,a6
            jsr       \1(a6)
            move.l    (a7)+,a6
```

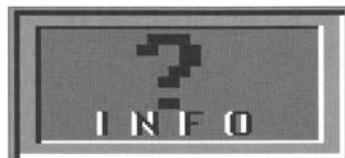
ENDM

will be thrown out by the assembler because ENDM is not indented. Similarly:

```
LINKLIB    MACRO
            move.l    a6, -(a7)
            move.l    \2,a6
            jsr       \1(a6)
            move.l    (a7)+,a6
```

ENDM

will also be thrown out because the indented LINKLIB text is not recognised as representing a label.



```
_LV00openLibrary    EQU    -552
_LV00closeLibrary   EQU    -414
_LV00output          EQU    -60
_LV00write           EQU    -48
start    lea        dos_name,a1    load pointer to library
                                             name
```

```
        moveq    #0,d0          any version will do!
openlib  CALLSYS  OpenLibrary,_AbsExecBase
        move.l   d0,_DOSBase    save returned pointer
        beq     exit           did library open OK?
dos_open CALLSYS  Output,_DOSBase get _stdout handle
        move.l   d0,_stdout     store it
        beq     close_dos      check _stdout is valid
write_text move.l  _stdout,d1    standard output handle
        move.l   #message1,d2   start of message
        move.l   #message1_SIZEOF,d3 message length
        CALLSYS Write,_DOSBase
        move.l   _stdout,d1     standard output handle
        move.l   #message2,d2   start of message
        move.l   #message2_SIZEOF,d3 message length
        CALLSYS Write,_DOSBase
        move.l   _stdout,d1     standard output handle
        move.l   #message3,d2   start of message
        move.l   #message3_SIZEOF,d3 message length
        CALLSYS Write,_DOSBase
close_dos move.l  _DOSBase,a1    library to close
        CALLSYS CloseLibrary,_AbsExecBase
exit     clr.l   d0
        rts
_DOSBase ds.l 1
_stdout  ds.l 1
dos_name dc.b 'dos.library', NULL
message1 dc.b '2YUR2YUBICUR2Y4ME',LF
message1_SIZEOF EQU *-message1
message2 dc.b 'or in other words...',LF
message2_SIZEOF EQU *-message2
```

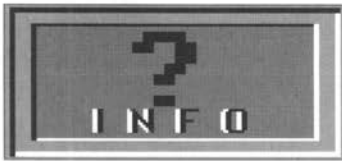
```
message3 dc.b 'Too wise you are, too wise you be, I see you  
are too wise for me!',LF
```

```
message3_SIZEOF EQU *-message3
```

There are more efficient ways of writing the above code using a program similar to BASIC's FOR/NEXT arrangements. These are important when dealing with a lot of text but, if you were able to get anywhere near the above arrangement, then give yourself a pat on the back because you're doing fine!

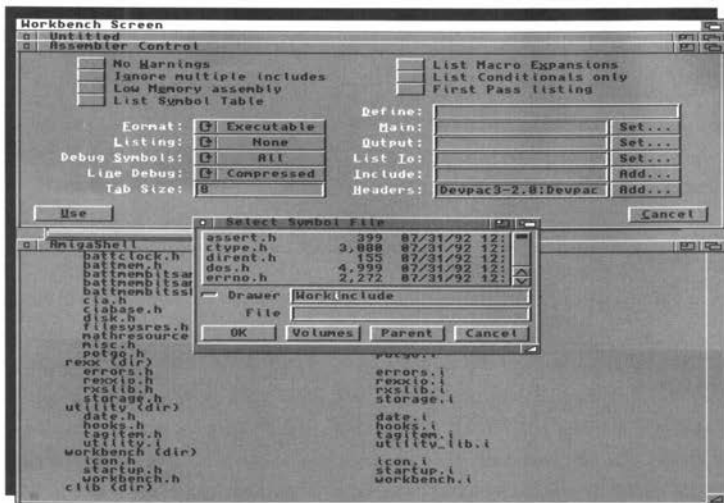
Insider Guide #27 – Progress Report

You have come quite a way down the Amiga 68000 programming road by now. As well as learning what assemblers and linkers are, and about the 68000 chip itself, you will have picked up tips on bits and bytes, hex and binary, macros and addressing modes and so on.



Most importantly, you have seen how with just a few simple instructions, those important Amiga libraries can be opened and used and this is the magic key which will eventually take you into more advanced Amiga programming!

The Amiga has a large collection of system include (or header) files. This chapter explains what they are, where to get them, and a little about how they are used...



As programs get larger it becomes useful to split the source code into any number of smaller units. The distinction between definition type items, such as macros and EQUates, and real code provides one useful dividing line. As we saw in Chapter 11, separate ASCII type text files containing such definitions are called *include files*. These files have usually been written as separate entities in order to make it easier for any number of programs to use the data they contain.

Include files do not usually contain real 68000 code and it isn't normal practice to include such things as subroutine definitions. The normal approach for including standard routines of this nature is either to assemble the subroutine separately and then attach this pre-assembled code unit (the object code module) to the assembled program at link time or, more commonly,

actually place the routine in a linker library along with all of your other frequently used standard routines. Similarly the code present in larger programs can often be spread over several different files. Again the normal approach for putting the various segments together is to assemble the files individually and then get the linker to combine them. The linker-based approach is used for practical reasons. Suppose you have a program that has many files and you wish to edit one of them in order to modify the program. It makes sense to be able to edit that single file, re-assemble it in isolation, and then just create the new version of the program by relinking the new module with the existing unchanged modules.

If the source files are arranged so all of the various code sections are brought in via separate include files – and there's no reason in theory why this can't be done – then, after any editing any of those files, the *complete* file set has to be re-assembled to produce a new working program.

To ease the burden on Amiga programmers, Commodore have made a variety of system files available containing not just hundreds, but thousands, of EQUate definitions, macros definitions, system structure templates and so forth that have been found useful for the Amiga software developer. Commercial assembler packages, such as Devpac, always come with a set of these include files but public domain assemblers, *a la* A68k etc, do not. In this latter case the files (if needed) must be purchased separately from Commodore. These official system files relate to things like the serial and other devices, DOS, Exec, graphics and Intuition libraries etc, and always come grouped in directories whose names provide the broad general use category. Exec headers, for instance, are found in the exec directory.

There are in fact two distinct sets of system files available. C programmers use a set of header files containing, as expected, C-style system definitions. These files can be easily recognised not only by their contents but by their .h filename extensions. Assembler programmers have a similarly arranged set of system definitions written in ways usable to the 680x0 coder. These include files are again always instantly recognisable because they have .i filename extensions.

These Amiga system files provide C and assembly language programmer with broadly parallel universes. A C programmer might, for instance, use definitions taken from the devices/serial.h header file. Someone coding a similar application using 68000 assembler would use the devices/serial.i include file.

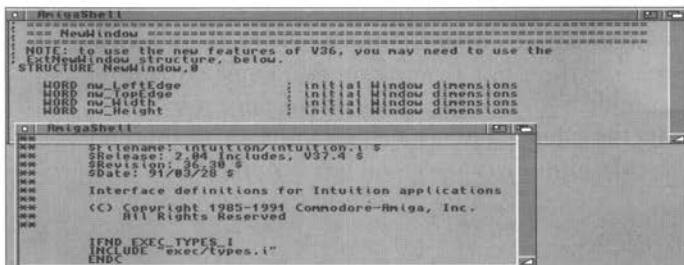
The existence of these files helps in two ways. Firstly, it eliminates the need for programmers to create their own definitions – a job which is both time consuming and error prone. Secondly, it promotes standardisation. All or most Amiga programmers soon get into the habit of using the ready made definitions provided in the Commodore supplied includes.

Insider Guide #28 – System File Updates

As each new version of the Amiga's operating system has appeared so updated header/include files are released in a package officially known as the Amiga Native Developer Update disks. At the time of going to press these cost £25 a set.

If you are one of the thousands of users who have now moved to Workbench 2 based machines (actually version 2.04 or later) then you may already be aware that a lot of new facilities have been added with this release: ARexx, the Gadtools library (which provides a whole range of new style, and easy to use, intuition gadget/menu building blocks), the ASL file/font requester library and many other goodies. The version 2.04 update disks provided the header/include file support for all of these new operating system additions and the more recent Workbench 3 headers/includes will be available by the time this book goes to print. For details of the Native Developer Update Disks write to:

The Developer Support Liaison Manager
Commodore Business Machines UK Ltd
Commodore House
The Switchback
Gardner Road
Maidenhead
Berks, SL6 7XA



```
AmigaShell
-----
NewWindow
NOTE: to use the new features of V36, you may need to use the
STRUCTURE NewWindow, 0
WORD nw_LeftEdge          initial Window dimensions
WORD nw_TopEdge           initial Window dimensions
WORD nw_Width             initial Window dimensions
WORD nw_Height            initial Window dimensions
-----
AmigaShell
-----
$Filename: intuition/intuition.1 $
$Release: 2.94 includes, 037.4 $
$Revision: 86.39 $
$Date: 91/03/28 $
-----
Interface definitions for Intuition applications
-----
(C) Copyright 1985-1994 Commodore-Amiga, Inc.
All Rights Reserved
-----
IFND EXEC_TYPES_1
INCLUDE "exec/types.1"
ENDC
```

The Snag For A68k Users

If, by following the A68k/Blink pathway, you've been thankful for being able to build yourself an assembler environment for next to

nothing then getting the official include files may present something of a dilemma. Relative to the cost of A68k/Blink they'll seem expensive. Although these files are worth their weight in gold for the serious programmer, not everyone wants to, or can afford to, invest in these official includes when they first start experimenting with Amiga assembly language.

In the early days, your programs are unlikely to need more than a few system definitions so it is easily possible to add these items to your own source code explicitly. Some coders even create their own *mini system headers* containing just those items which are regularly needed.

Why this Book has Avoided the System Includes

In the main I've chosen to create normal Amiga-flavoured code by

adding suitable explicit macro and EQUate definitions to the code examples rather than use the official Commodore include files. There are several reasons for this. Firstly, it would have been grossly unfair on those A68k users without the official files. Secondly, much of the material present in the official files is not particularly relevant for the newcomer to assembly language. Thirdly it is probably more instructive in the early stages to see the various EQUate and macro definitions being used rather than have those definitions buried away in a large, and relatively complex, hierarchy of system files. Lastly, it turns out to be very easy for those programmers who do have access to the official includes to modify their programs if they wish to do so.

Let me give you an example: a macro called LINKLIB (similar to the Insider Guide LINKLIB macro discussed and used in earlier chapters) is present in the Amiga exec/libraries.i system file. In any of the Insider Guide examples which use this macro it would be possible to eliminate the explicit definition that we've used in this book:

```
LINKLIB      MACRO
              move.l    a6, -(a7)
              move.l    \2, a6
              jsr       \1(a6)
              move.l    (a7)+, a6
            ENDM
```

and instead just add the following line near the top of the source code

INCLUDE exec/libraries.i

This causes the assembler to bring in the libraries.i include file which, amongst other items, contains the equivalent, Commodore defined LINKLIB macro.

In the next chapter I'll be taking a look at some more Intuition library functions. The example programs require access to quite a number of system include file based items and this provides another opportunity to illustrate the difference between the *home cooked* definitions and the official system file coding approaches.

Insider Guide #29 – Keeping Up To Date?

System file updates are usually done in such a way that any changes involving existing structures and library routines are backwards compatible. This means, for example, that a 1.3 based programmer could use more up-to-date system files providing they stuck to using only those library routines which were in fact available with the 1.3 operating system release.

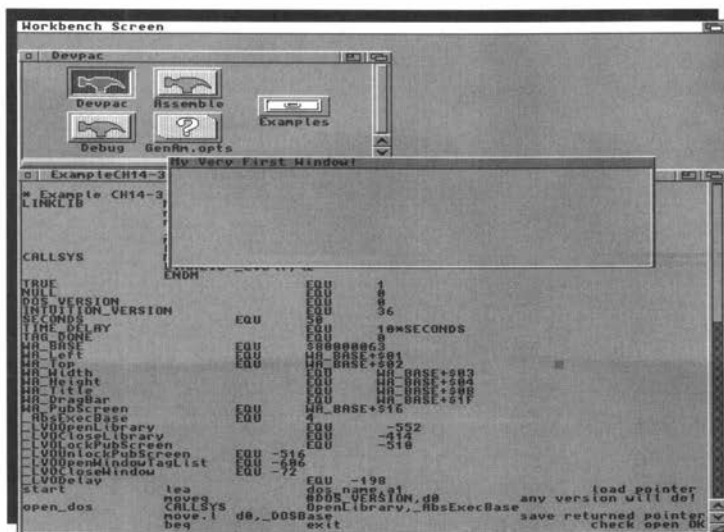
However, don't forget that you only really need up-to-date files if you have an up-to-date, or upgraded, Amiga. Many programmers, having originally purchased the (now dated) Workbench 1.3 header/include file set have quite sensibly continued to use them simply because they are still using 1.3 based machines.

```

* This file contains:
*
* 1. The traditional identifiers for gadget Flags, Activation, and Type,
*    and for window Flags and IDCMP classes. They are defined in terms
*    of their new versions, which serve to prevent confusion between
*    similar-sounding but different identifiers (like IDCMP_WINDOWACTIVE
*    and MFLE_ACTIVATE).
*
* 2. Some tag names and constants whose labels were adjusted after V36.
*
* By default, 1 and 2 are enabled.
*
* Set INTUI_V36_NAMES_ONLY to exclude the traditional identifiers and
* the original V36 names of some identifiers.
*
*-----
*
*      IFND INTUITION_INTUITION_I
*      INCLUDE "intuition/intuition.i"
*      ENDC

```


Intuition programming is a massive subject and it's impossible to do it justice in an introductory book of this nature. This chapter should, however, place you in good stead for tackling some of the more advanced Amiga programming books later on...



In Chapter 13 I mentioned that in recent releases of the Amiga's operating system quite a few things have changed. Let's make one thing clear at the outset – the new system facilities make life easier for the Amiga coder, rather than harder. There are however those changes to contend with and unfortunately some of the most visible ones affect even the most elementary operations that an enthusiastic new Amiga coder might experiment with. New methods for opening screens and windows are a typical case in point because these operations, like many others, are now done using things called *tag lists*.

Ringling the Changes

Before we get stuck into the main tag list discussions it is worthwhile looking at the types of problems that the

Commodore system programmers have had to contend with as they upgraded the Amiga environment. This will help put a number of otherwise confusing *alternative Amiga library function* issues into context.

First and foremost comes the need for backward compatibility. Software companies who must maintain products to run on all O/S versions in current use can be badly hit by poorly thought out operating system *enhancements*. To their credit Commodore have gone to great lengths to minimise these types of difficulties. Even so, modifying an existing Amiga product so that it runs say under both Release 2 and 1.3 is hard work and still little short of a nightmare for most programmers. The reason I'm mentioning all this is simple – you will find that with Release 2 onwards some operations can be performed in a variety of seemingly different ways and it is important to understand why this is so. Much of the flexibility has been provided primarily for those developers who, in terms of compatibility, were in the unfortunate position of being stuck between a rock (the 1.3 O/S) and a hard place (Release 2 and later).

Given that the 1.3 user base is likely to diminish as users upgrade existing machines, and new models like the brilliant A1200 and A4000/030 make their mark, many developers chose to provide (and maintain) separate versions of their products. This latter approach is also the one that most Amiga users will want to adopt with their own programs because experience shows that once they're working with the new environment their interest in 1.3 coding will dwindle rapidly! In order to appreciate some of the new system function options (available from Release 2 onwards) it is necessary to understand how tag lists fit into the compatibility scenario.

When New Windows Are Not!

I've already mentioned that the official include files contain, amongst other things, templates (definitions) for system structures used to define various entities used by the system. With operating system releases up to and including 1.3, one of these templates, called a NewWindow structure, provided standard names and internal structure position data for the various attribute fields: size, position and so on. To open a window you would create a NewWindow structure, fill in the appropriate details, and then call an Intuition library function named `OpenWindow()`.

In order to provide the Release 2 system enhancements, however, some established operations, like window opening, required additional parameters to be specified and Commodore's problem was to find a way to do this that would minimise any code compatibility upsets. What they wanted was a solution that would eliminate altogether any need to extend existing system structures in future O/S releases. The approach they adopted is based on the use of arrays, or lists of arrays, that contain self-identifying parameter values each consisting of an identifying label and a corresponding *real* parameter value. Since these lists provide a way of tagging additional parameters onto existing O/S structures, they were called tag lists. Where appropriate, new library functions look for such items and use them either in addition to, or as a replacement for, any existing structure data they might have used in the past.

Tag lists solve the problem of providing additional parameters but they do not, on their own, provide any help as far as backward compatibility goes. One of the things that Commodore did from Release 2 was to create an extended NewWindow definition which included an additional extension field at the end of the new structure. A special include file flag value was defined which, when set, told the OpenWindow() system routine that tag item values were present. When running under Release 1.3 or earlier this extension field was obviously ignored but, by using these types of transparent extensions coupled with conditional code that looks for Release 2 libraries or later, developers were (and, in theory, still are) able to write code that worked under all O/S versions.

Window Opening

So far I've tried to paint a general picture about how and why tag lists came into existence and why you are going to find a variety of seemingly similar window opening functions in newer versions of the intuition library. Before tackling some 68000 code issues there are a few more points about the opening of Intuition windows that need to be made.

Since Release 2 there have actually been five different ways to write window opening code. For a start, a programmer can set up an ExtNewScreen structure containing a pointer to a tag list holding any additional parameters required.

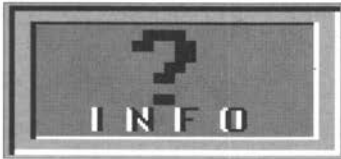
Alternatively, an OpenWindowTagList() function can be used in one of two different ways. The originally required parameters can be specified, *a la* 1.3, in a NewWindow structure with additional (Release 2

onward) arguments being provided in a separate tag list. Or, a NULL NewWindow pointer can be used coupled with a tag list that contains all of the required window parameters. This latter approach often turns out to be the easiest because only the non-default window attribute values need be supplied.

That covers three of the approaches available for making a window opening call. Unfortunately (or fortunately depending on your viewpoint) two more variations exist based on the use of the amiga.lib linker library's OpenWindowTags() function. Rather than passing a single tag list pointer, this function expects to get its tag parameters from the stack along with a NewWindow pointer. Again the NewWindow pointer can be NULL so tag based parameters can be used exclusively if required.

Insider Guide #30 – Window Opening – The Bottom Line

Whilst the 1.3 operating system offered just one basic routine for opening a window, from Release 2 onward there are five! Despite how it may seem, this hasn't been done to cause budding Amiga programmers grief – it is just that it has been necessary to provide both a backward compatibility pathway for those who need it, and to ensure increased flexibility as far as future enhancements are concerned.



Forget about the stack-oriented OpenWindowTags() approaches because they've been provided primarily for C programmers.

Forget also about the use of the NewWindow structures (extended or otherwise) because

these structures now only remain in the include files for developers who needed compatibility with Workbench 1.3 and earlier. Concentrate instead on understanding how the OpenWindowTagList() function is used when window attributes are specified completely by tag list data!

Tag Lists

Tag lists are based on an Amiga system structure known as a TagItem and if you look in the Utility/tagitem.i header you'll find that TagItems are defined in this sort of fashion:

STRUCTURE	TagItem,0
ULONG	ti_Tag

ULONG	ti_Data
LABEL	ti_SIZEOF

The STRUCTURE, ULONG and LABEL items are some rather ingenious macros designed to allow the assembly language include files to be written using C style structure units. Don't worry about the way they work just accept the fact that each tag item consists of a pair of long word (four byte) values. The first long word provides a 32 bit TagItem identity, the second a corresponding 32 bit data value.

Most tag identity values are context specific. The intuition.i header file contains definitions of all manner of Intuition-related tag identities. WA_Left, WA_Top, WA_Width and WA_Height, for instance, are used to specify window position and size information.

A number of general tag item values have also been defined and can be found, along with the TagItem structure, in the utility/tagitem.i file. Here are a few examples:

- TAG_IGNORE** Indicates that the associated data item should be ignored.
- TAG_SKIP** Skip this and the next ti_Data TagItems.
- TAG_MORE** Marks the end of one array and indicates that at least one other TagItem array exists. The ti_Data field points to the next TagItem array to be used.
- TAG_END** Signals the end of an array (in this case ti_Data would be unused).

It's important to understand that tag lists have been adopted to solve the problem of adding additional parameters to function calls once and for all. In short, from Release 2 onwards, they have become an integral part of the Amiga's programming environment. If you are interested in getting into up-to-date Amiga programming you *must* understand how they work. The rest of this chapter is concerned with making sure you do!

The program we're going to develop opens a window on the Workbench screen. This, rather simple, coding task is actually very useful because it draws a lot of earlier ideas together thus providing another essential step forward. Having said that, there is still much to explain and this is best done in stages. Our starting point, which should by now be familiar, are those all important library opening operations.

Open Sesame

For reasons that should soon become apparent we have got to open both the DOS and the Intuition libraries. These operations, as indicated in Figure 14.1, form the outer framework for our program:

```

OPEN DOS LIBRARY
      OPEN INTUITION LIBRARY
            Open a Window?
      CLOSE INTUITION LIBRARY
CLOSE DOS LIBRARY

```

Figure 14.1 A starting framework for the first Intuition example

The following code, which just opens two libraries instead of one, does nothing you've not seen before but take note that, since the Intuition library function which we shall be using has only been available from library version 36, we've got to specify this version as the earliest acceptable library. An EQUate statement has been used to define the library version – this is preferable to placing the number 36 into the code itself. It provides better program documentation and the definition, since it is near the start of the program, is easier to find and update should the need arise.

* Example CH14-1.s

```

LINKLIB      MACRO
              move.l    a6, -(a7)
              move.l    \2, a6
              jsr       \1(a6)
              move.l    (a7)+, a6
              ENDM

CALLSYS      MACRO
              LINKLIB  _LVO\1, \2
              ENDM

NULL                EQU    0
DOS_VERSION         EQU    0
INTUITION_VERSION   EQU    36

```



```
_AbsExecBase      EQU    4
_LV00openLibrary  EQU    -552
_LV0CloseLibrary  EQU    -414

start   lea    dos_name,a1      load pointer to
library                               name

        moveq  #DOS_VERSION,d0  any version will do!

open_dos CALLSYS OpenLibrary,_AbsExecBase

        move.l d0,_DOSBase      save returned pointer
        beq    exit             check open OK?

open_int lea    intuition_name,a1 load pointer to
                                       library name

        moveq  #INTUITION_VERSION,d0 specify
                                       minimum lib
                                       version

        CALLSYS OpenLibrary,_AbsExecBase

        move.l d0,_IntuitionBase save returned
                                       pointer
        beq    close_dos        check open OK?

*      LIBRARIES OPEN OK SO WE COULD DO SOMETHING!

close_int move.l _IntuitionBase,a1 library to close
        CALLSYS CloseLibrary,_AbsExecBase

close_dos move.l _DOSBase,a1      library to close
        CALLSYS CloseLibrary,_AbsExecBase

exit    clr.l  d0

        rts

_DOSBase      ds.l  1
_IntuitionBase ds.l  1
dos_name      dc.b  'dos.library',NULL
intuition_name dc.b  'intuition.library',NULL
```

Completing the Plan of Action

You might think that, once the

Intuition library has been opened, we can simply use some Intuition library function to open a window on the Workbench screen. Under some circumstances, however, a user (or an applications program) can close the Workbench and so a method of preventing this happening whilst a program is in the middle of setting up a window is needed.

Intuition provides something called a *locking function* which allows an application to force the Workbench – or other public screen – to stay open. An unlocking function is also available and this can be used at any time after the new window has been successfully created. For reasons of symmetry I prefer to pair lock/unlock calls in much the same way as open/close library operations are paired. The overall program plan can be seen in Figure 14.2.

```
OPEN DOS LIBRARY
OPEN INTUITION LIBRARY
  LOCK INTUITION SCREEN
    OPEN WINDOW
      WAIT FOR SPECIFIED TIME
        CLOSE WINDOW
          UNLOCK INTUITION SCREEN
            CLOSE INTUITION LIBRARY
              CLOSE DOS LIBRARY
```

Figure 14.2 The final framework for the first Intuition example

Why the time delay? Just to give you a chance to see the window appear, and this is basically a cop out aimed at keeping things as simple as possible. Fully fledged Intuition programs use gadgets and menus coupled to an Exec style inter-task message communications scheme. This type of coding, at the 68000 level, is sufficiently complex as to be outside the scope of this book. To produce runnable Intuition programs whilst avoiding these message-based techniques it has been necessary, having opened a window, to use a DOS Delay() function to enable the window to be seen on the display otherwise it would disappear almost immediately. If you want more details of the Exec messaging system see the Mastering Amiga Assembler and Mastering Amiga System references in the bibliography!

Adding the Screen Locking/Unlocking Code

These routines are used just like any other library functions. The required parameters are set up, the library base is placed in register a6, and the appropriate indirect subroutine call is made. On return the results – if any – are checked to see that they are valid. One way of locking the Workbench screen is to set up a static name definition using a dc.b directive:

```
workbench_name    dc.b 'Workbench',NULL
```

along with a variable to store the returned Workbench pointer

```
workbench_p      ds.l 1
```

and then use this now familiar type of library call code arrangement:

```
lock_screen  lea      workbench_name,a0  pointer to screen  
                                                    name  
  
              CALLSYS LockPubScreen,_IntuitionBase  
  
              move.l  d0,workbench_p      save returned  
                                                    pointer  
  
              beq    close_int           check return value?
```

Insider Guide #31 – Lock Em Up!

Function Name: LockPubScreen()

Description: Prevents a public screen from closing

Call Format: screen = LockPubScreen(name)

Registers: d0 a0

Arguments: name - pointer to text string giving name of screen

Return Value: screen - pointer to screen (or NULL if routine fails)



Insider Guide #32 – Free At Last!

Function Name: UnlockPubScreen()

Description: Release a public screen lock

Call Format: UnlockPubScreen(name,[screen])

Registers: a0 a1

Arguments: name - can provide pointer to name of screen.
[Normally supplied as NULL]

screen - pointer to screen

Return Value: None



Insider Guide #33 – Open Up!

Function Name: OpenWindowTagList()

Description: Opens window using NewWindow and/or tag list data

Call Format: window=OpenWindowTagList(new_window,tag_items)

Registers: d0 a0 a1

Arguments: new_window - pointer to a NewWindow structure
tag_items - pointer to a tag list

Return Value: window - address of window (NULL if routine fails)



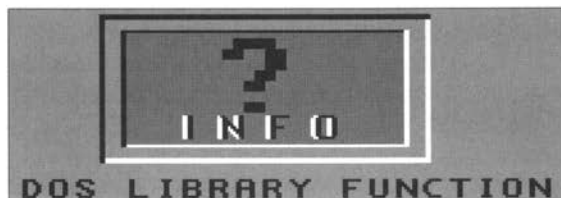
***Insider Guide #34 –
Closing Windows When You're Finished***

Function Name: CloseWindow()
Description: Does the obvious!
Call Format: CloseWindow(window)
Registers: a0
Arguments: window - pointer to the window to close
Return Value: None



Insider Guide #35 – A DOS Time Waster

Function Name: Delay()
Description: Causes program to be suspended for a specified time
Call Format: Delay(time)
Registers: d1
Arguments: time - delay specified in ticks (50 ticks = 1 sec)
Return Value: None



If by chance the Workbench screen is not open when this call is made Intuition will open it for you. The WorkBench screen unlocking is even simpler:

```
unlk_screen  move.l  #NULL,a0      screen name not needed
              move.l  workbench_p,a1  screen to unlock
              CALLSYS  UnlockPubScreen,_IntuitionBase
```

There's more good news because the OpenWindowTagList() function that we're going to use actually works even if you execute it *without* supplying either a structure block or a tag list. It sets up some reasonable defaults and even seems to identify the Workbench as the destination screen automatically. This means we can open a trial window very easily using:

```
open_window  move.l  #NULL,a0
              move.l  #NULL,a1      no tag list
              CALLSYS  OpenWindowTagList,_IntuitionBase
              move.l  d0>window_p    our window
              beq      unlk_screen
```

and close it using the CloseWindow() routine

```
move.l  window_p,a0      window to close
CALLSYS  CloseWindow,_IntuitionBase
```

If the DOS Delay() function is used to provide an arbitrary 5 second (250 tick) delay like this

```
move.l  #TIME_DELAY,d1
CALLSYS  Delay,_DOSBase
```

then – bingo! – we've got all we need to create a runnable, observable Intuition window. Put this new code into our existing framework and things really start to cook. If you assemble the resulting code (Example CH14-2.s), in the same way as explained in Chapters 9 and 10, you find that when the program is run, an Intuition window appears on the Workbench screen for about 5 seconds! (A68k users must remembering to put a terminal END statement into their source.)

* Example CH14-2.s

```
LINKLIB    MACRO
            move.l    a6, -(a7)
            move.l    \2, a6
            jsr      \1(a6)
            move.l    (a7)+, a6
            ENDM

CALLSYS    MACRO
            LINKLIB  _LVO\1, \2
            ENDM

NULL                      EQU    0
DOS_VERSION                EQU    0
INTUITION_VERSION         EQU    36
SECONDS                    EQU    50
TIME_DELAY                 EQU    5*SECONDS
_AbsExecBase              EQU    4
_LV0OpenLibrary           EQU    -552
_LV0CloseLibrary          EQU    -414
_LV0LockPubScreen         EQU    -510
_LV0UnlockPubScreen       EQU    -516
_LV0OpenWindowTagList     EQU    -606
_LV0CloseWindow           EQU    -72
_LV0Delay                  EQU    -198

start    lea    dos_name, a1        load pointer to
                                             library name
        moveq  #DOS_VERSION, d0    any version will do!
open_dos CALLSYS OpenLibrary, _AbsExecBase
        move.l d0, _DOSBase        save returned pointer
        beq    exit                check open OK?
open_int lea    intuition_name, a1  load pointer to
                                             library name
```

```
moveq    #INTUITION_VERSION,d0    specify
                                     minimum lib
                                     version

CALLSYS  OpenLibrary,_AbsExecBase

lock_screen
move.l   d0,_IntuitionBase        save returned
                                     pointer
beq      close_dos                check open OK?

lock_screen lea    workbench_name,a0  pointer to
                                     screen name

CALLSYS  LockPubScreen,_IntuitionBase

move.l   d0,workbench_p          save returned
                                     pointer
beq      close_int                check return value?

open_window
move.l   #NULL,a0
move.l   #NULL,a1                no tag list
CALLSYS  OpenWindowTagList,_IntuitionBase
move.l   d0>window_p            pointer to our
                                     window
beq      unlk_screen
move.l   #TIME_DELAY,d1
CALLSYS  Delay,_DOSBase
move.l   window_p,a0            window to close
CALLSYS  CloseWindow,_IntuitionBase

unlk_screen
move.l   #NULL,a0                screen name not
                                     needed
move.l   workbench_p,a1          screen to unlock
CALLSYS  UnlockPubScreen,_IntuitionBase

close_int
move.l   _IntuitionBase,a1       library to close
CALLSYS  CloseLibrary,_AbsExecBase

close_dos
move.l   _DOSBase,a1            library to close
CALLSYS  CloseLibrary,_AbsExecBase

exit
clr.l   d0
rts                                           logical end of program
```


<code>_DOSBase</code>	<code>ds.l</code>	<code>1</code>
<code>_IntuitionBase</code>	<code>ds.l</code>	<code>1</code>
<code>workbench_p</code>	<code>ds.l</code>	<code>1</code>
<code>window_p</code>	<code>ds.l</code>	<code>1</code>
<code>dos_name</code>	<code>dc.b</code>	<code>'dos.library',NULL</code>
<code>intuition_name</code>	<code>dc.b</code>	<code>'intuition.library',NULL</code>
<code>workbench_name</code>	<code>dc.b</code>	<code>'Workbench',NULL</code>

Adding Some Tag Data

It's quite surprising that the previous code works because, as you may have

noticed, we didn't tell Intuition which screen we wanted our window to open on. Along with many other possible parameters, this is something that can be specified in a tag list. The numerical values of the tag identities need to be the same as those defined by Commodore and, for clarity, the standard tag identity names should also be used. Without access to the official headers these items need to be set up within the source code itself. The following EQUates do just that and produce results identical to those found in the official `intuition/intuition.i` and `utility/tagitem.i` include files. The official documentation provides many other tag definitions:

<code>TAG_DONE</code>	<code>EQU</code>	<code>0</code>
<code>WA_BASE</code>	<code>EQU</code>	<code>\$80000063</code>
<code>WA_Left</code>	<code>EQU</code>	<code>WA_BASE+\$01</code>
<code>WA_Top</code>	<code>EQU</code>	<code>WA_BASE+\$02</code>
<code>WA_Width</code>	<code>EQU</code>	<code>WA_BASE+\$03</code>
<code>WA_Height</code>	<code>EQU</code>	<code>WA_BASE+\$04</code>
<code>WA_Title</code>	<code>EQU</code>	<code>WA_BASE+\$0B</code>
<code>WA_DragBar</code>	<code>EQU</code>	<code>WA_BASE+\$1F</code>
<code>WA_PubScreen</code>	<code>EQU</code>	<code>WA_BASE+\$16</code>

`WA_Left`, `WA_Top`, `WA_Width`, `WA_Height` and `WA_Title` are used to provide details of the size and title of the window provided in the corresponding data fields. `WA_DragBar`, as the name suggests, asks Intuition to place a drag bar on the window if the data item field is set to `TRUE` and `WA_PubScreen` is used to supply the address of the screen being used.

Insider Guide #36 –**If You Have the Official Amiga Include Files...**

Then you do not need to set up the tag identity values because they are provided in the intuition/intuition.i and utility/tagitem.i include files. Nor is it necessary to define the LINKLIB macro because that is present in the exec/libraries.i file. Best of all you only need to specify one include file, namely intuition/intuition.i, in your source and this automatically brings in the various files just mentioned – and many more. In practice then, you just need this statement at the start of your source code:

```
INCLUDE      intuition/intuition.i
```



Most tag identities and values can be set up as static definitions consisting of identity+value pairs. After all we can decide what values are needed *before* we assemble the program!

The data part of the WA_PubScreen however, which is a pointer to the public screen, cannot be set up in this way since this info comes back from the LockPubScreen() routine. Because of this it is convenient to give the WA_PubScreen data field a separate label. The Workbench address has to be stored anyway in order to release the public screen lock so I've arranged to store it directly in the tag list data rather than store it separately and move a copy of the address to the tag list. I've arbitrarily chosen to create a 440x100 pixel window (called 'My Very First Window!') with a drag bar which opens at the x/y screen location of (100,50). Here are the tag list storage definitions being used:

```
tags          dc.1    WA_PubScreen
workbench_p  ds.1     1
              dc.1    WA_Left,100
              dc.1    WA_Top,50
              dc.1    WA_Width,440
              dc.1    WA_Height,100
              dc.1    WA_DragBar,TRUE
              dc.1    WA_Title>window_name
              dc.1    TAG_DONE,NULL
```

By adding the above mentioned fragments to the existing program, and changing the `OpenWindowTagList()` call so that register `a1` gets loaded with the start of the tag list like this:

```
open_window  move.l    #NULL,a0
              lea      tags,a1                our tag list
              CALLSYS OpenWindowTagList,_IntuitionBase
```

We are now able to pass Intuition the tag list information about the window. Here, to end this chapter, is the finished source code which shows you how the above ideas fit together:

*** Example CH14-3.s**

```
LINKLIB      MACRO
              move.l    a6,-(a7)
              move.l    \2,a6
              jsr      \1(a6)
              move.l    (a7)+,a6
              ENDM

CALLSYS      MACRO
              LINKLIB  _LVO\1,\2
              ENDM

TRUE          EQU      1
NULL         EQU      0
DOS_VERSION  EQU      0
INTUITION_VERSION EQU    36
SECONDS      EQU      50
TIME_DELAY   EQU      10*SECONDS
TAG_DONE     EQU      0
WA_BASE      EQU      $80000063
WA_Left      EQU      WA_BASE+$01
WA_Top       EQU      WA_BASE+$02
WA_Width     EQU      WA_BASE+$03
WA_Height    EQU      WA_BASE+$04
WA_Title     EQU      WA_BASE+$0B
```

WA_DragBar	EQU	WA_BASE+\$1F
WA_PubScreen	EQU	WA_BASE+\$16
_AbsExecBase	EQU	4
_LV0OpenLibrary	EQU	-552
_LV0CloseLibrary	EQU	-414
_LV0LockPubScreen	EQU	-510
_LV0UnlockPubScreen	EQU	-516
_LV0OpenWindowTagList	EQU	-606
_LV0CloseWindow	EQU	-72
_LV0Delay	EQU	-198
start	lea	dos_name,a1 load pointer to library name
	moveq	#DOS_VERSION,d0 any version will do!
open_dos	CALLSYS	OpenLibrary,_AbsExecBase
	move.l	d0,_DOSBase save returned pointer
	beq	exit check open OK?
open_int	lea	intuition_name,a1 load pointer to library name
	moveq	#INTUITION_VERSION,d0 specify minimum lib version
	CALLSYS	OpenLibrary,_AbsExecBase
	move.l	d0,_IntuitionBase save returned pointer
	beq	close_dos check open OK?
lock_screen	lea	workbench_name,a0 pointer to screen name
	CALLSYS	LockPubScreen,_IntuitionBase
	move.l	d0,workbench_p save returned pointer
	beq	close_int check return value?
open_window	move.l	#NULL,a0
	lea	tags,a1 our tag list
	CALLSYS	OpenWindowTagList,_IntuitionBase

Amiga Insider Guide

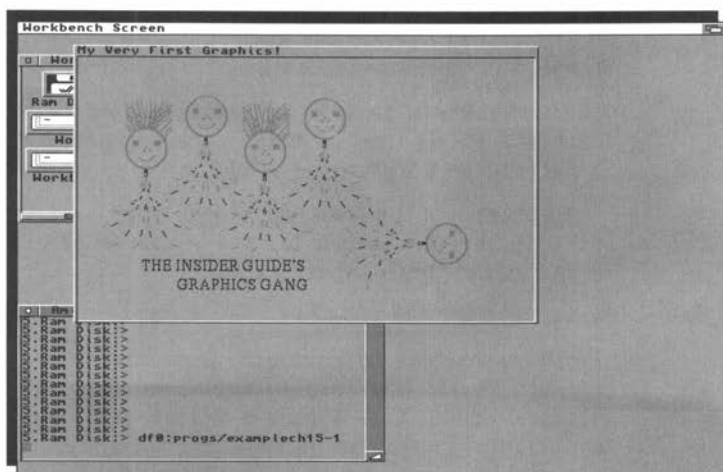
```

                                move.l  d0,window_p          pointer to our
                                                                window
                                beq      unlk_screen
                                move.l  #TIME_DELAY,d1
                                CALLSYS  Delay,_DOSBase
                                move.l  window_p,a0          window to close
                                CALLSYS  CloseWindow,_IntuitionBase
unlk_screen  move.l  #NULL,a0          screen name not
                                                                needed
                                move.l  workbench_p,a1        screen to unlock
                                CALLSYS  UnlockPubScreen,_IntuitionBase
close_int   move.l  _IntuitionBase,a1  library to close
                                CALLSYS  CloseLibrary,_AbsExecBase
close_dos   move.l  _DOSBase,a1        library to close
                                CALLSYS  CloseLibrary,_AbsExecBase
exit        clr.l  d0
                                rts          logical end of program

_DOSBase    ds.l  1
_IntuitionBase ds.l  1
window_p    ds.l  1
tags        dc.l  WA_PubScreen
workbench_p ds.l  1
                                dc.l  WA_Left,100
                                dc.l  WA_Top,50
                                dc.l  WA_Width,440
                                dc.l  WA_Height,100
                                dc.l  WA_DragBar,TRUE
                                dc.l  WA_Title,window_name
                                dc.l  TAG_DONE,NULL
dos_name    dc.b  'dos.library',NULL
intuition_name dc.b  'intuition.library',NULL
```

<code>workbench_name</code>	<code>dc.b 'Workbench',NULL</code>
<code>window_name</code>	<code>dc.b 'My Very First Window!',NULL</code>

Creating Amiga graphics is an exciting and rewarding job but, surprisingly, it is not that difficult even when programming in assembler language. This chapter unveils some of the tricks that are needed to get you started...



The easiest way to make a start with Amiga graphics is to use some of the high-level drawing functions available in the Intuition library. Intuition's arrangements for drawing graphics into multiple-bitplane screens and windows are, in terms of the underlying ideas, quite complex but the existence of pre-written routines means that all the complexity can be nicely hidden away. Intuition provides routines for displaying text, polygon shapes which come under the general name of Borders, and bitplane images. In this chapter it is the image-oriented operations that come under the magnifying glass.

Intuition's image drawing is based on a structure – a standardised block of data – known, believe it or not, as an *Image structure*. The Intuition Image structure tells Intuition things about the size and location of the image and is used primarily in conjunction with a library routine called `DrawImage()`.

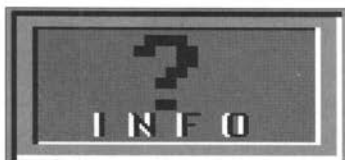
Insider Guide #37 – Graphics The Easy Way**Function Name:** DrawImage()**Description:** This is Intuition's high-level Image drawing routine**Call Format:**`DrawImage(rastport, image, left_offset, top_offset);`**Registers:** a0 a1 d0 d1**Arguments:** rastport – pointer to a RastPort
 image – pointer to an Image structure
 left_offset – a general left offset used with all of the linked Image structures of a particular DrawImage() call.
 top_offset – a general top offset used with all of the linked Image structures of a particular DrawImage() call.**Return Value:** None**Notes:** It is convenient to have displacement offsets in the DrawImage() call itself because this allows a global offset to be applied to a chain of Image structures. You may have a group of a couple of dozen separate images on display but, if you so desire, can reposition the whole group (keeping their relative positions the same) by altering the global offsets.

On the face of it this function call arrangement makes the display of graphics images very easy. In practice there is a big problem looming because, although using the Image structures and the DrawImage() function is easy enough, creating the associated Image data is not. Working out from first principles exactly how to create the Image data for a particular object (whether it be a cloud, a tree or some fancy backdrop display) turns out to be nigh-on impossible.

The good news is that, as a programmer, you *never* have to do this because nowadays tools are available which make the task of creating even the most complex graphics a piece of cake. Firstly, the existence of clear inter-program graphics definition guidelines (part of the now famous IFF standard) encouraged the creation of programs that can read and write graphics data using a common data-file format.

Secondly, programs such as Electronic Art's *DPaint* and Cloanto's *Personal Paint* have provided an easy means of creating IFF picture files without the programmer having to be involved with the underlying complexities of bitplane data generation. More help has appeared and tools, such as Inovatronic's *Power Windows*, are able to convert IFF brushes into the equivalent assembly source compatible Image data.

Insider Guide #38 – BitPlane Graphics Theory



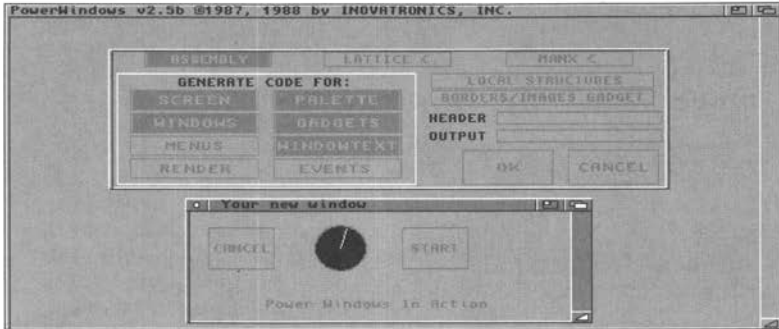
An example bit-by-bit plan for a small graphics object is provided in the Intuition sections of the Addison Wesley Libraries RKM manual but you are unlikely to have to use this approach. The relationships between displays, bitplanes, Images etc, are dealt with very thoroughly in the RKM manuals and, when you get to the point where you start to need in-depth information, they are without doubt the best place to look.

Getting Graphics into Code

As already mentioned, the task of creating and using graphics in your Amiga programs has been considerably eased by some sophisticated graphics-support tools. Deluxe Paint for instance can be used to create any required graphics objects which can then be stored as picture files. This can be done using either complete screen pictures or small, user definable, graphic sections called brushes. By switching on Deluxe Paint's X/Y co-ordinate display a user can easily create objects of a given size. If some graphic images 40 pixels by 60 pixels are needed then a suitable background area can be marked out, the images drawn, and the brush facility then used to save that particular area of the display.

So, how do you get say a Deluxe Paint drawing into your program? It is possible for a program to read in an IFF file and convert it into a suitable (Amiga displayable) form directly. The advantages of this particular approach are that you only need to read the picture into memory just prior to displaying it, so it becomes very easy to change the graphics without re-assembling the program – you just swap one IFF file for another. The disadvantage is that some rather complicated programming is needed to handle straight IFF graphics file loading.

The other approach is to take the IFF file and convert it to an Intuition Image structure using a utility such as Power Windows. Having done that the Image structure and the associated Image data can be read into the source code of the program and displayed using the Intuition DrawImage() function.



Tools like PowerWindows are a great help to the serious Amiga coder.

A Runnable Example

The next program is based on the last example of Chapter 14 but in addition to opening a window on the Workbench screen, it also draws some bitplane graphics. In this case DPaint was used to draw the picture, which was subsequently saved as an IFF Brush. Power Windows was then used to convert the brush to assembler-style data statements.

The DrawImage() function requires a pointer to the window's rastport (drawing area) and this address is stored inside the window structure, ie the block of data set up by Intuition when a window is opened. I've not discussed structure creation and structure access in this book because, although important, it leans heavily towards C language style concepts and complicated macros that are best left for more advanced books. To be honest it isn't necessary to understand how C-style structures are created in order to use them and, since tools like Power Windows can create the structures automatically anyway, all that's really needed initially is a little help in using them.

By loading an address register with the base of a window structure it is possible to *reach inside* and extract, ie copy, the window's rastport address. It's done using the 68000's indirect addressing with displacement addressing mode – the same as that used to execute system library routines. Where do we get the base from? This is the window pointer, the window address, that was collected when the window was

Insider Guide #39 – Devpac to A68k Assembler Section Conventions

We've already seen that A68k requires source files to contain an explicit END statement at the end of the source code. This is easily done by reading the source file into any available ASCII text editor – ED or Memacs will do – moving to the end of the text file and inserting a terminal END statement as the last line of the source code.

There is also a minor difference between the A68k and Devpac assemblers in the syntax used for program section identification. Most assembler programs contain real code, initialised data areas, and uninitialised areas. The section directive is used to tell the assembler about the purpose of particular areas of source code. The assembler subsequently embeds this information in the object code so that the linker can use it. For most small programs you don't need to use, or worry about, section directives unless you are including graphics data – this must end up in chip memory in order for the Amiga's custom chips (namely the blitter) to access it.

With Devpac this is done using this sort of statement

SECTION IMAGE,DATA_C

Where IMAGE is just an arbitrary section name and DATA_C is a keyword indicating chip memory.

Charlie Gibb's A68k assembler requires slightly different section syntax and so a small change is necessary for coders using this program. The source code line shown above, which has to appear just before the graphics data itself, must with A68k, be changed to

SECTION IMAGE,DATA,CHIP

```

window_name      dc.b  'My Very First Graphics!',NULL

Image1:
dc.w  0,0      ;XY origin relative to container TopLeft
dc.w  255,215  ;image width and height in pixels
dc.w  7        ;number of bitplanes in image
dc.l  2        ;imageData1 : pointer to imageData
dc.b  255,255 ;PlanePick and PlaneOff
dc.l  1        ;next Image structure
ImageData1:
dc.w  255,255,255,255,255,255,255,255,255,255,255,255
dc.w  255,255,255,255,255,255,255,255,255,255,255,255
dc.w  255,255,255,255,255,255,255,255,255,255,255,255
dc.w  255,255,255,255,255,255,255,255,255,255,255,255
dc.w  255,255,255,255,255,255,255,255,255,255,255,255
dc.w  255,255,255,255,255,255,255,255,255,255,255,255
dc.w  255,255,255,255,255,255,255,255,255,255,255,255
dc.w  255,255,255,255,255,255,255,255,255,255,255,255

```

opened. In the example code this pointer comes back in register d0, so I just copy it to a convenient address register, a1:

```
move.l d0,a1      window address in a1
```

The offset, ie the displacement which is used to reach into the window structure, is another one of those many items defined in the official Amiga include files. It's official name is wd_RPort, so named because

it refers to a **WinDow RastPORT**. When this value is available the Amiga programmer can copy the field:

```
move.l wd_RPort(a1),a0      copy rastport pointer into a0
```

In the first version of the program I've assumed that the header files are *not* available and, since the RKM manuals tell me that the numerical value of the wd_RPort offset is 50 (decimal), I've just added an additional EQUate to the source code to create my own definition of this item.

Once the rastport address is available the rest of the DrawImage() parameters can be set up and the function executed in the normal fashion:

```
lea      Image1,a1          pointer to image
moveq    #20,d0             example left offset
moveq    #15,d1            example top offset
CALLSYS  DrawImage,_IntuitionBase
```

Now all that's needed are some examples which show these drawing routines in action.

If You Haven't Got the Official Includes

Everyone using Devpac has the official include files so this example is based on using the A68k assembler. All of the macros, tag identities, LVO values and field definitions have therefore been, like many previous examples, placed directly into the source and no official include files are needed to assemble it correctly:

```
* Example CH15-1.s
LINKLIB  MACRO
          move.l    a6,-(a7)
          move.l    \2,a6
          jsr      \1(a6)
          move.l    (a7)+,a6
          ENDM
CALLSYS  MACRO
          LINKLIB  _LVO\1,\2
          ENDM
```

Amiga Insider Guide

TRUE	EQU	1	
NULL	EQU	0	
DOS_VERSION	EQU	0	
INTUITION_VERSION	EQU	36	
SECONDS	EQU	50	
TIME_DELAY	EQU	10*SECONDS	
TAG_DONE	EQU	0	
WA_BASE	EQU	\$80000063	
WA_Left	EQU	WA_BASE+\$01	
WA_Top	EQU	WA_BASE+\$02	
WA_Width	EQU	WA_BASE+\$03	
WA_Height	EQU	WA_BASE+\$04	
WA_Title	EQU	WA_BASE+\$0B	
WA_DragBar	EQU	WA_BASE+\$1F	
WA_PubScreen	EQU	WA_BASE+\$16	
wd_RPort	EQU	50	
_AbsExecBase	EQU	4	
_LV0OpenLibrary	EQU	-552	
_LV0CloseLibrary	EQU	-414	
_LV0LockPubScreen	EQU	-510	
_LV0UnlockPubScreen	EQU	-516	
_LV0OpenWindowTagList	EQU	-606	
_LV0CloseWindow	EQU	-72	
_LV0Delay	EQU	-198	
_LV0DrawImage	EQU	-114	
start	lea	dos_name,a1	load pointer to library name
	moveq	#DOS_VERSION,d0	any version will do!
open_dos	CALLSYS	OpenLibrary,_AbsExecBase	
	move.l	d0,_DOSBase	save returned pointer
	beq	exit	check open OK?

```
open_int    lea        intuition_name,a1  load pointer to
                                                    library name
            moveq    #INTUITION_VERSION,d0  specify
                                                    minimum lib
                                                    version
            CALLSYS  OpenLibrary,_AbsExecBase
            move.l   d0,_IntuitionBase  save returned
                                                    pointer
            beq     close_dos          check open OK?
lock_screen lea        workbench_name,a0  pointer to
                                                    screen name
            CALLSYS  LockPubScreen,_IntuitionBase
            move.l   d0,workbench_p    save returned
                                                    pointer
            beq     close_int          check return
                                                    value?
open_window move.l   #NULL,a0
            lea     tags,a1            our tag list
            CALLSYS  OpenWindowTagList,_IntuitionBase
            move.l   d0>window_p      pointer to our
                                                    window
            beq     unlk_screen
draw_image  move.l   d0,a1            window address in
                                                    a1
            move.l   wd_RPort(a1),a0  copy rastport
                                                    pointer into a0
            lea     Image1,a1        pointer to image
            moveq    #20,d0          example left
                                                    offset
            moveq    #15,d1          example top
                                                    offset
            CALLSYS  DrawImage,_IntuitionBase
wait        move.l   #TIME_DELAY,d1
            CALLSYS  Delay,_DOSBase
            move.l   window_p,a0      window to close
            CALLSYS  CloseWindow,_IntuitionBase
```

Amiga Insider Guide

```
unlk_screen  move.l  #NULL,a0          screen name not
                                         needed
                                         move.l  workbench_p,a1    screen to unlock
CALLSYS      UnlockPubScreen,_IntuitionBase
close_int    move.l  _IntuitionBase,a1  library to close
CALLSYS      CloseLibrary,_AbsExecBase
close_dos    move.l  _DOSBase,a1        library to close
CALLSYS      CloseLibrary,_AbsExecBase
exit         clr.l  d0
                                         rts          logical end of program

_DOSBase     ds.l  1
_IntuitionBase ds.l  1
window_p     ds.l  1
tags         dc.l  WA_PubScreen
workbench_p  ds.l  1
                                         dc.l  WA_Left,50
                                         dc.l  WA_Top,20
                                         dc.l  WA_Width,420
                                         dc.l  WA_Height,250
                                         dc.l  WA_DragBar,TRUE
                                         dc.l  WA_Title,window_name
                                         dc.l  TAG_DONE,NULL
dos_name     dc.b  'dos.library',NULL
intuition_name dc.b  'intuition.library',NULL
workbench_name dc.b  'Workbench',NULL
window_name  dc.b  'My Very First Graphics!',NULL
Image1:      Generated from IFF brush using Power
                                         Windows
dc.w  0,0    ;XY origin relative to container
                                         TopLeft
dc.w  395,215 ;Image width and height in pixels
dc.w  2      ;number of bitplanes in Image
dc.l  ImageData1 ;pointer to ImageData
```

dc.b \$0003,\$0000 ;PlanePick and PlaneOnOff
dc.l NULL ;next Image structure

SECTION Image,DATA,CHIP

ImageData1:

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$001F,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$001F,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$001F,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

 /\/\/\/\/\/\/\/\/\//
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$001F,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$001F,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$001F,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

END

The Official Approach

When the official Amiga include files are available many of the definitions present in the previous version are unnecessary because they are available in the includes files themselves. LVO values are available from two main sources. Firstly, they are defined as part of the amiga.lib linker library so any program that needs to link with amiga.lib has access to the necessary values automatically. Secondly both Commodore and commercial assemblers like Devpac provide utilities for generating LVO values from the Commodore function description files; these form part of the official include file disk material. The following source assumes that a file, called function_offsets.i, is available with suitable LVO offset information:

*** Example CH15-2.s**

```

                INCLUDE intuition/intuition.i
                INCLUDE function_offsets.i

CALLSYS      MACRO
                LINKLIB _LVO\1,\2
                ENDM

TRUE          EQU      1
NULL          EQU      0
DOS_VERSION   EQU      0
INTUITION_VERSION EQU    36
SECONDS       EQU      50
TIME_DELAY    EQU      10*SECONDS

start        lea      dos_name,a1          load pointer to
                                                library name
                moveq   #DOS_VERSION,d0    any version will
                                                do!

open_dos      CALLSYS  OpenLibrary,_AbsExecBase
                move.l  d0,_DOSBase        save returned
                                                pointer
                beq     exit                check open OK?

open_int      lea      intuition_name,a1    load pointer to
                                                library name

```

```
moveq    #INTUITION_VERSION,d0    specify
                                     minimum lib
                                     version

CALLSYS  OpenLibrary,_AbsExecBase

lock_screen  move.l    d0,_IntuitionBase    save
                                               returner
                                               pointer

          beq        close_dos            check open OK?

          lea        workbench_name,a0    pointer to
                                               screen name

CALLSYS  LockPubScreen,_IntuitionBase

          move.l    d0,workbench_p    save returned
                                               pointer

          beq        close_int            check return
                                               value?

open_window  move.l    #NULL,a0

          lea        tags,a1            our tag list

CALLSYS  OpenWindowTagList,_IntuitionBase

          move.l    d0>window_p    pointer to our
                                               window

draw_image  beq        unlk_screen

          move.l    d0,a1            window address
                                               in a1

          move.l    wd_RPort(a1),a0    copy rastport
pointer into a0

          lea        Image1,a1        pointer to image

          moveq     #20,d0            example left
                                               offset

          moveq     #15,d1            example top
                                               offset

CALLSYS  DrawImage,_IntuitionBase

wait       move.l    #TIME_DELAY,d1

CALLSYS  Delay,_DOSBase

          move.l    window_p,a0        window to close

CALLSYS  CloseWindow,_IntuitionBase
```

Amiga Insider Guide

```
unlk_screen  move.l  #NULL,a0          screen name not
                                         needed
                                         move.l  workbench_p,a1    screen to unlock
CALLSYS      UnlockPubScreen,_IntuitionBase
close_int    move.l  _IntuitionBase,a1  library to close
CALLSYS      CloseLibrary,_AbsExecBase
close_dos    move.l  _DOSBase,a1        library to close
CALLSYS      CloseLibrary,_AbsExecBase
exit         clr.l  d0
                                         rts          logical end of program
_DOSBase     ds.l  1
_IntuitionBase ds.l  1
window_p     ds.l  1
tags         dc.l  WA_PubScreen
workbench_p  ds.l  1
                                         dc.l  WA_Left,50
                                         dc.l  WA_Top,20
                                         dc.l  WA_Width,420
                                         dc.l  WA_Height,250
                                         dc.l  WA_DragBar,TRUE
                                         dc.l  WA_Title,window_name
                                         dc.l  TAG_DONE,NULL
dos_name     dc.b  'dos.library',NULL
intuition_name dc.b  'intuition.library',NULL
workbench_name dc.b  'Workbench',NULL
window_name  dc.b  'My Very First Graphics!',NULL
Image1:      Generated from IFF brush using Power
                                         Windows
dc.w  0,0    ;XY origin relative to container
                                         TopLeft
dc.w  395,215 ;Image width and height in pixels
dc.w  2      ;number of bitplanes in Image
dc.l  ImageData1 ;pointer to ImageData
```

dc.b \$0003,\$0000 ;PlanePick and PlaneOnOff

dc.l NULL ;next Image structure

SECTION Image,DATA_C

ImageData1:

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$001F,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$001F,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$001F,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

//\//\//\//\//\//\//

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$001F,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$001F,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

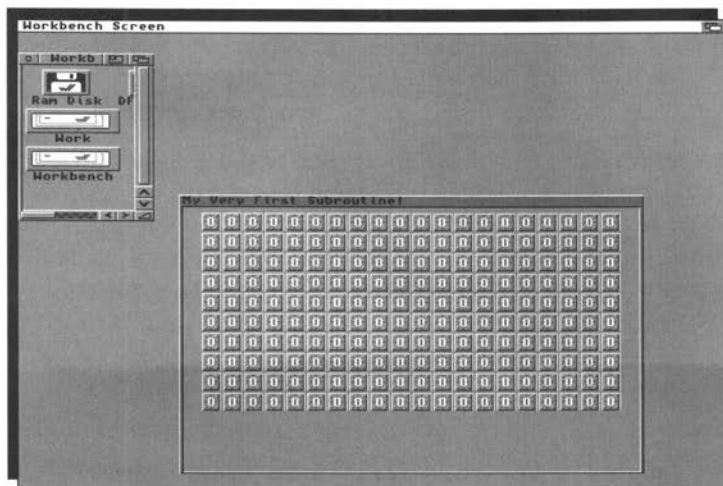
dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$001F

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

dc.w \$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000,\$0000

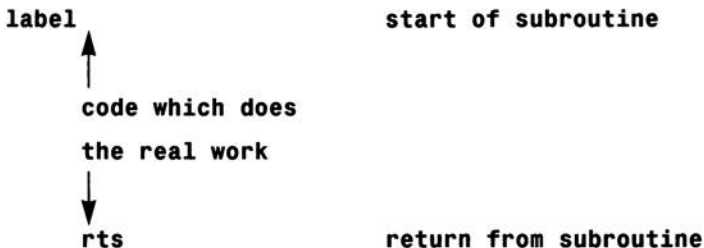
This chapter deals with a number of topics, all carefully chosen to lead you on to greater things...



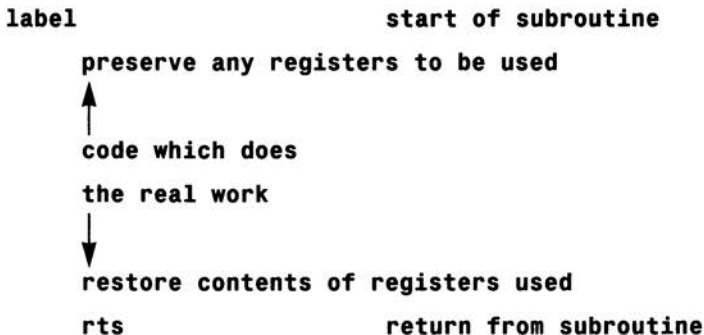
The examples we've dealt with so far have been kept deliberately straightforward and you'll have doubtless noticed that the examples have adopted a "do this, then that, then something else..." style coupled with some conditional test branches to ensure that the right things happen at the right time. This is fine for small pieces of code but as programs get larger these simple linear arrangements become less and less attractive. In short, methods need to be found for identifying and separating the various logical tasks a program must perform. Equally important, coding has to progress in a way which makes program development (and code re-use) less traumatic. One of the ways this is done is to identify those jobs which are suitable for writing as subroutines.

Subroutines

When a particular piece of code is recognised as being generally useful – or perhaps is repeated many times within just one program – it is worth writing as a subroutine and on the 68000 the basic arrangement takes this form:



The subroutine, as likely as not, has to use one or more of the 68000's registers and this creates a potential problem because the contents of some of those registers may already contain information important to the program. The usual way of preventing a subroutine from inadvertently destroying register data is to preserve the contents of any registers being used (by placing them on the 68000's stack) and, when the subroutine has finished its work, re-instating them. In general then the framework adopted by most subroutines actually looks more like this:



These multiple save/restore operations are so common that the 68000 has provided special instructions for the job. They are called *multiple move*, or *movem*, instructions and they allow ranges of registers to be specified. If, for instance, we want to store data registers d2-d7, and address registers a2-a5, on the 68000 stack using the stack pointer register sp (ie a7) it could be done like this:

```
movem.l d2-d7/a2-a5, -(sp) preserve registers
```

and to reinstate the contents of those registers:

```
movem.l (sp)+,d2-d7/a2-a5 restore registers
```

so 68000 subroutines end up using these type of schemes:

```
label start of subroutine  
movem.l d2-d7/a2-a5, -(sp) preserve registers  
↑  
code which does  
the real work  
↓  
movem.l (sp)+,d2-d7/a2-a5 restore registers  
rts return from subroutine
```

Subroutine Parameters

Subroutines usually need some sort of data to act on and these items are known

as the subroutine's parameters. There are a variety of parameter passing techniques available to the 68000 programmer but the one used in our examples is that used by the Amiga's run-time library functions – ie any values the subroutine needs are passed in microprocessor registers.

Very often a supplied parameter needs to be re-used by the subroutine and one commonly seen Amiga-specific coding slip is arranging to supply one of more values in registers d0, d1, a0, or a1, and at some point making a run-time library call. The registers just mentioned, the so called *scratch* registers, are often destroyed by the library call and the official Amiga documentation specifically states that the values of those registers should always be regarded as lost!

An Example Subroutine

We're going to develop a nice and easy, but still quite effective routine,

called DrawGrid() that takes a specified graphics object (defined as an Intuition image) and creates a tile/wallpaper effect within a window by drawing multiple copies of the image using the sort of caller-defined MxN grid, as seen in Figure16.1.

Quite a few parameters need to be passed to this DrawGrid() subroutine. Knowing from the function description that the Intuition DrawImage() library routine needs a rastport pointer in a0, an image pointer in a1, plus left and top offsets in d0 and d1 the following register arrangements were chosen:

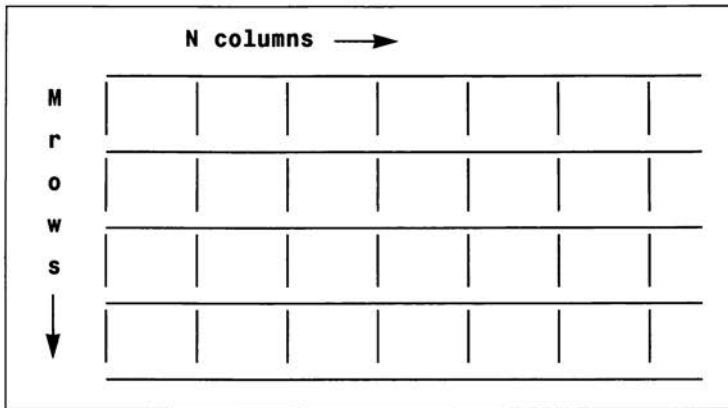


Figure 16.1. A MxN caller defined grid.

a0 is to hold the window rastport pointer

a1 is to hold the image pointer

d0 is to hold the starting left offset value

d1 is to hold the starting top offset value

d2 holds the required horizontal block count, ie column count

d3 is to hold the required vertical block count, ie row count

The subroutine draws each row of the grid by making DrawImage() library calls incrementing the function's left offset drawing position by the width of the image block each time. Once a row is complete the top offset value is increased by the height of the image and the row drawing operations repeated.

Rather unfortunately we must assume that the parameters present in a0, a1, d0 and d1 (the scratch registers) are destroyed by each and every DrawImage() Intuition call. So these values need to be copied to other registers at the start of the routine so that they can be reloaded as required. Also, a copy has to be kept of the original left offset because one working value needs to be increased as the images in any given row are drawn but the original value is still needed to reset the offset at the start of the second and subsequent row drawing operations.

Now all of these values could be placed on the 68000's stack, or stored in ds.x defined memory locations but, for maximum execution speed it is actually faster to keep as much data as possible within the 68000 registers themselves. I have therefore, somewhat arbitrarily, chosen to preserve the rastport pointer in register a2, the image pointer in regis-

ter a3, the left offset in d7, the top offset in a4, the column block count in a5 and opted to collect and store the image width and height in d4 and d5 respectively. Additionally register d6 is used to store the current left offset value at any given time. Why were address registers chosen for some data items? It's simply that almost all (data and address) registers were needed to store all of the various items!

Parameters supplied:

- a0 holds the window rastport pointer**
- a1 holds the image pointer**
- d0 holds the starting left offset value**
- d1 holds starting top offset value**
- d2 holds required horizontal block count, ie column count**
- d3 holds required vertical block count, ie row count**

Additional registers used within the routine:

- d4 stores image width**
- d5 stores image height**
- d6 holds an updated (current) left offset**
- d7 copy of original left offset**
- a2 copy of rastport pointer**
- a3 copy of image pointer**
- a4 copy of original top offset**
- a5 copy of original column count**

Although I've not made a point of emphasising this in earlier chapters, it's worth remembering that instructions which use smaller size operands do execute more quickly and, of course, memory space is saved. Graphics routines, such as the one we are developing, should make all reasonable efforts to take advantage of such things and in this case a lot of the data, block counts, the DrawImage() offsets etc, can in fact be specified as word sized (two byte) data items.

The image width and height values, which are also word sized fields do not need to be explicitly provided as they're stored in the image structure itself and can be obtained using the word-based form of indirect addressing with displacement. In the case of the Intuition Image structure the displacements required to obtain the image width and height are given the standard ig_Width and ig_Height respectively.

Predefined values are available in the intuition.i include file but since they have absolute values of 4 and 6 it is easy enough to define identical EQUate values if necessary and so still write this type of conventional Amiga code:

```

move.w   ig_Width(a1),d4       get image width in d4
move.w   ig_Height(a1),d5     get image height in d5

```

Coupled to the previously mentioned initial parameter copy operations, the subroutine entry code therefore ends up looking like this:

```

move.l   a0,a2                 preserve rastport pointer
move.l   a1,a3                 preserve image pointer
move.w   d0,d6                 d6 = current left offset
move.w   d0,d7                 preserve left offset for re-use
move.w   d1,a4                 preserve top offset
move.w   d2,a5                 preserve column count for re-use
move.w   ig_Width(a1),d4      get image width in d4
move.w   ig_Height(a1),d5    get image height in d5

```

Drawing a Row of Images

Basically we draw the image structure (using the same function call

arrangement as in the last chapter), decrease the horizontal block count, and test to see whether the count is zero thus checking all horizontal images in a row have been drawn. If the current row is complete we move onto the next row; otherwise we reset the top offset value in register d1 (which the library call might have destroyed), update the left offset value (by adding the image width to it), reset the d0, a0 and a1 parameters which may also have been destroyed by the library call, and continue looping back to the DrawImage() function:

```

draw_row   CALLSYS   DrawImage,_IntuitionBase
           subq     #1,d2       decrease count
           beq     next_row
           move.w   a4,d1       set top offset
           add.w   d4,d6       form new left offset
draw_row2  move.w   d6,d0       needed for library
                               function call

```

```
move.l    a2,a0    restore rastport pointer
move.l    a3,a1    restore image pointer
bra       draw_row keep going
```

At the start of each new row we decrease the count value and check whether another row needs to be drawn. If it does the left offset value is reset to the start of the row, the column count (which represents the number of horizontal blocks to be drawn) is similarly reset, and the top offset value is increased by the height of the image. In the following code notice how branch on equal (beq) and branch always (bra) instructions are used to create program loops which finally exit when the count values (which are decreased by one each time a loop is executed) become zero:

```
next_row  subq    #1,d3    decrease count
          beq     draw_end
          move.w  d7,d6    reset start left
                          offset for row
          move.w  a5,d2    reset column count
          move.w  a4,d1
          add.w   d5,d1
          move.w  d1,a4    top offset for next
                          row
          bra     draw_row2
```

draw_end

These types of program loops with counter exits become increasingly important in more advanced 68000 programming and in fact the 68000 even provides some rather more specialised automated instructions for creating this type of code.

By collecting all the things discussed so far we can piece together this, reasonably efficient, DrawGrid() routine. I have incidentally chosen to preserve and restore all registers used (including scratch registers d0, d1, a0 and a1) because this allows the routine to be re-used quickly without having to reload any unchanged scratch register based parameter values:

DrawGrid:

```
; Requires following parameters on entry...
; a0 holds window rastport pointer
; a1 holds image pointer
; d0 holds starting left offset value
; d1 holds starting top offset value
; d2 holds required horizontal block count, ie column
count
; d3 holds required vertical block count, ie row count
    movem.l  d0-d7/a0-a5,-(sp)  preserve
                                registers
    move.l   a0,a2              preserve rastport pointer
    move.l   a1,a3              preserve image pointer
    move.w   d0,d6              d6 = current left offset
    move.w   d0,d7              preserve left offset for
                                re-use
    move.w   d1,a4              preserve top offset
    move.w   d2,a5              preserve column count for
                                re-use
    move.w   ig_Width(a1),d4    get image width in
                                d4
    move.w   ig_Height(a1),d5  get image height in
                                d5
draw_row    CALLSYS DrawImage,_IntuitionBase
    subq    #1,d2              decrease count
    beq     next_row
    move.w   a4,d1              set top offset
    add.w   d4,d6              form new left offset
draw_row2   move.w   d6,d0      needed for library
                                function call
    move.l   a2,a0              restore rastport pointer
    move.l   a3,a1              restore image pointer
    bra     draw_row           keep going
```

```
next_row  subq    #1,d3      decrease count
          beq     draw_end
          move.w  d7,d6      reset start left
                          offset for row
          move.w  a5,d2      reset column count
          move.w  a4,d1
          add.w   d5,d1
          move.w  d1,a4      top offset for next row
          bra     draw_row2
draw_end  movem.l (sp)+,d0-d7/a0-a5 restore registers
          rts
```

Are we finished? Not yet, because hidden in this routine is a glaring inefficiency that is easily eliminated. The CALLSYS macro is pushing a6 onto the stack and retrieving it after the DrawImage() routine returns. In this particular routine these actions are quite pointless because I've been a little crafty – register a6 has deliberately not been used except within CALLSYS. There is therefore no reason why, by just setting up a6 initially, we can't replace the CALLSYS generated code with the single equivalent indirect subroutine call, thus eliminating all of those a6 push/pull operations. Trust me – the time savings are significant.

DrawGrid:

```
;   Requires following parameters on entry...
;   a0 holds window rastport pointer
;   a1 holds image pointer
;   d0 holds starting left offset value
;   d1 holds starting top offset value
;   d2 holds required horizontal block count, ie column
    count
;   d3 holds required vertical block count, ie row count
draw_row2 movem.l  numbers      bogus row
          movem.l  d0-d7/a0-a6,-(sp) preserve registers
          move.l   _IntuitionBase,a6 set up library base
          move.l   a0,a2        preserve rastport pointer
```

```
move.l  a1,a3      preserve image pointer
move.w  d0,d6      d6 = current left offset
move.w  d0,d7      preserve left offset for
                   reuse
move.w  d1,a4      preserve top offset
move.w  d2,a5      preserve column count for
                   re-use
move.w  ig_Width(a1),d4  get image width in
                       d4
move.w  ig_Height(a1),d5  get image height in
                       d5
draw_row  jsr      _LV0DrawImage(a6)  a faster
                                       alternative
subq    #1,d2      decrease count
beq     next_row
move.w  a4,d1      set top offset
add.w  d4,d6      form new left offset
draw_row2 move.w  d6,d0      needed for library
                           function call
move.l  a2,a0      restore rastport pointer
move.l  a3,a1      restore image pointer
bra     draw_row   keep going
next_row  subq    #1,d3      decrease count
beq     draw_end
move.w  d7,d6      reset start left offse for
                   row
move.w  a5,d2      reset column count
move.w  a4,d1
add.w  d5,d1
move.w  d1,a4      top offset for next row
bra     draw_row2
draw_end movem.l  (sp)+,d0-d7/a0-a6  restore registers
rts
```

Building a Test Framework

All we have to do now is put this subroutine into a run-

able example to check that it actually works. The next program, suitable for A68k coders and others without the official includes, is based on the ideas discussed in Chapter 15. Instead of the previous direct DrawImage() function call some example parameters are set up for our new DrawGrid() routine:

```
draw_images  move.l   d0,a1           window address in a1
              move.l   wd_RPort(a1),a0  copy rastport
              lea     Image1,a1         pointer to image
              moveq   #20,d0           example left offset
              moveq   #15,d1           example top offset
              moveq   #20,d2           example columns count
              moveq   #10,d3           example rows count
```

and the subroutine simply executed like this:

```
jsr    DrawGrid    our subroutine
```

* Example CH16-1.s

```
LINKLIB    MACRO
            move.l    a6,-(a7)
            move.l    \2,a6
            jsr      \1(a6)
            move.l    (a7)+,a6
            ENDM

CALLSYS    MACRO
            LINKLIB  _LVO\1,\2
            ENDM

TRUE      EQU    1
NULL      EQU    0
DOS_VERSION    EQU    0
INTUITION_VERSION    EQU    36
```

```
SECONDS                EQU    50
TIME_DELAY             EQU    10*SECONDS
TAG_DONE              EQU    0
WA_BASE               EQU    $80000063
WA_Left              EQU    WA_BASE+$01
WA_Top               EQU    WA_BASE+$02
WA_Width            EQU    WA_BASE+$03
WA_Height           EQU    WA_BASE+$04
WA_Title            EQU    WA_BASE+$0B
WA_DragBar          EQU    WA_BASE+$1F
WA_PubScreen        EQU    WA_BASE+$16
wd_RPort            EQU    50
ig_Width            EQU    4
ig_Height           EQU    6
_AbsExecBase        EQU    4
_LV0OpenLibrary     EQU    -552
_LV0CloseLibrary    EQU    -414
_LV0LockPubScreen   EQU    -510
_LV0UnlockPubScreen EQU    -516
_LV0OpenWindowTagList EQU    -606
_LV0CloseWindow     EQU    -72
_LV0Delay           EQU    -198
_LV0DrawImage       EQU    -114

start    lea    dos_name,a1    load pointer to
                                library name
                                moveq    #DOS_VERSION,d0    any version will
                                do?
open_dos CALLSYS OpenLibrary,_AbsExecBase
                                move.l  d0,_DOSBase    save returned
                                pointer
                                beq     exit        check open OK?
```


Amiga Insider Guide

```
open_int    lea        intuition_name,a1    load pointer to
                                                    library name
                                                    moveq       #INTUITION_VERSION,d0    specify
                                                    minimum lib
                                                    version
CALLSYS     OpenLibrary,_AbsExecBase
move.l      d0,_IntuitionBase    save returned
                                                    pointer
                                                    beq        close_dos            check open OK?
lock_screen lea        workbench_name,a0  pointer to screen
                                                    name
CALLSYS     LockPubScreen,_IntuitionBase
move.l      d0,workbench_p      save returned
                                                    pointer
                                                    beq        close_int          check return value?
open_window move.l      #NULL,a0
                                                    lea        tags,a1            our tag list
CALLSYS     OpenWindowTagList,_IntuitionBase
move.l      d0>window_p        pointer to our
                                                    window
                                                    beq        unlk_screen
draw_images move.l      d0,a1            window address in
                                                    a1
                                                    move.l     wd_RPort(a1),a0    copy rastport
                                                    pointer into a0
                                                    lea        Image1,a1        pointer to image
                                                    moveq     #20,d0            example left offset
                                                    moveq     #15,d1            example top offset
                                                    moveq     #20,d2            example columns
                                                    count
                                                    moveq     #10,d3            example rows count
                                                    jsr       DrawGrid          our subroutine
wait        move.l      #TIME_DELAY,d1
CALLSYS     Delay,_DOSBase
```

```
        move.l   window_p,a0           window to close
        CALLSYS CloseWindow,_IntuitionBase
unlk_screen  move.l   #NULL,a0          screen name not
                                                needed
        move.l   workbench_p,a1       screen to unlock
        CALLSYS UnlockPubScreen,_IntuitionBase
close_int   move.l   _IntuitionBase,a1 library to close
        CALLSYS CloseLibrary,_AbsExecBase
close_dos   move.l   _DOSBase,a1       library to close
        CALLSYS CloseLibrary,_AbsExecBase
exit        clr.l   d0
        rts                               logical end of program
```

DrawGrid:

```
; Requires following parameters on entry...
; a0 holds window rastport pointer
; a1 holds image pointer
; d0 holds starting left offset value
; d1 holds starting top offset value
; d2 holds required horizontal block count, ie column
count
; d3 holds required vertical block count, ie row count
        movem.l  d0-d7/a0-a6,-(sp)     preserve registers
        move.l   _IntuitionBase,a6     set up library
                                                base
        move.l   a0,a2                 preserve rastport
                                                pointer
        move.l   a1,a3                 preserve image
                                                pointer
        move.w   d0,d6                 d6 = current left
                                                offset
        move.w   d0,d7                 preserve left
                                                offset for re-use
        move.w   d1,a4                 preserve top offset
```

Amiga Insider Guide

	move.w	d2,a5	preserve column count for re-use
	move.w	ig_Width(a1),d4	get image width in d4
	move.w	ig_Height(a1),d5	get image height in d5
draw_row	jsr	_LV0DrawImage(a6)	a faster alternative
	subq	#1,d2	decrease count
	beq	next_row	
	move.w	a4,d1	set top offset
	add.w	d4,d6	form new left offset
draw_row2	move.w	d6,d0	needed for library function call
	move.l	a2,a0	restore rastport pointer
	move.l	a3,a1	restore image pointer
	bra	draw_row	keep going
next_row	subq	#1,d3	decrease count
	beq	draw_end	
	move.w	d7,d6	reset start left offset for row
	move.w	a5,d2	reset column count
	move.w	a4,d1	
	add.w	d5,d1	
	move.w	d1,a4	top offset for next row
	bra	draw_row2	
draw_end	movem.l	(sp)+,d0-d7/a0-a6	restore registers
	rts		
_DOSBase	ds.l	1	

```
_IntuitionBase    ds.1    1
window_p          ds.1    1
tags              dc.1    WA_PubScreen
workbench_p       ds.1    1
                  dc.1    WA_Left,50
                  dc.1    WA_Top,20
                  dc.1    WA_Width,420
                  dc.1    WA_Height,250
                  dc.1    WA_DragBar,TRUE
                  dc.1    WA_Title,window_name
                  dc.1    TAG_DONE,NULL
dos_name          dc.b    'dos.library',NULL
intuition_name    dc.b    'intuition.library',NULL
workbench_name    dc.b    'Workbench',NULL
window_name       dc.b    'My Very First Subroutine!',NULL
```

Image1:

```
dc.w    0,0                ;XY origin relative to container
                          TopLeft
dc.w    19,18              ;Image width and height in pixels
dc.w    2                  ;number of bitplanes in Image
dc.l    ImageData1        ;pointer to ImageData
dc.b    $0003,$0000        ;PlanePick and PlaneOnOff
dc.l    NULL              ;next Image structure
```

SECTION Image,DATA,CHIP**ImageData1:**

```
dc.w    $0000,$0000,$0000,$0000,$00C0,$8000,$00C0,$8000
dc.w    $08C2,$8000,$0000,$8000,$00C0,$8000,$00C0,$8000
dc.w    $1CCE,$8000,$1CCE,$8000,$00C0,$8000,$00C0,$8000
dc.w    $0000,$8000,$00C0,$8000,$08C2,$8000,$00C0,$8000
dc.w    $3FFF,$8000,$0000,$0000,$0000,$0000,$7FFF,$C000
dc.w    $40C0,$4000,$50C4,$4000,$40C0,$4000,$41E0,$4000
```

```
dc.w    $43F0,$4000,$43F0,$4000,$5FFE,$4000,$5FFE,$4000
dc.w    $43F0,$4000,$43F0,$4000,$41E0,$4000,$50C4,$4000
dc.w    $40C0,$4000,$40C0,$4000,$4000,$4000,$0000,$0000
END
```

The Official Alternative

If the official Amiga includes are available many explicit definitions

can be avoided in the program source. Here's the Devpac version that adopts the same include file arrangements as the Chapter 15 example:

*** Example CH16-2.s**

```
                INCLUDE intuition/intuition.i
                INCLUDE function_offsets.i
CALLSYS        MACRO
                LINKLIB _LVO\1,\2
                ENDM

TRUE           EQU    1
NULL          EQU    0
DOS_VERSION   EQU    0
INTUITION_VERSION EQU 36
SECONDS       EQU    50
TIME_DELAY    EQU    10*SECONDS

start         lea     dos_name,a1      load pointer to
                                                library name
              moveq  #DOS_VERSION,d0  any version will
                                                do!

open_dos      CALLSYS OpenLibrary,_AbsExecBase
              move.l d0,_DOSBase     save returned
                                                pointer

              beq    exit            check open OK?

open_int      lea     intuition_name,a1 load pointer to
                                                library name
              moveq  #INTUITION_VERSION,d0 specify
                                                minimum lib
                                                version
```

```
CALLSYS  OpenLibrary,_AbsExecBase
lock_screen  move.l  d0,_IntuitionBase      save returned
                                     pointer
           beq    close_dos        check open OK?
           lea   workbench_name,a0  pointer to
                                     screen name
CALLSYS  LockPubScreen,_IntuitionBase
open_window  move.l  d0,workbench_p    save returned
                                     pointer
           beq    close_int        check return value?
           move.l #NULL,a0
           lea   tags,a1           our tag list
CALLSYS  OpenWindowTagList,_IntuitionBase
draw_images  move.l  d0>window_p      pointer to our
                                     window
           beq    unlk_screen
           move.l d0,a1            window address in
                                     a1
           move.l wd_RPort(a1),a0  copy rastport
                                     pointer into a0
           lea   Image1,a1        pointer to image
           moveq #20,d0           example left offset
           moveq #15,d1           example top offset
           moveq #20,d2           example columns
count       moveq  #10,d3         example rows count
           jsr   DrawGrid         our subroutine
wait        move.l  #TIME_DELAY,d1
CALLSYS  Delay,_DOSBase
           move.l window_p,a0      window to close
CALLSYS  CloseWindow,_IntuitionBase
unlk_screen  move.l  #NULL,a0        screen name not
                                     needed
```

```
        move.l   workbench_p,a1      screen to unlock
        CALLSYS UnlockPubScreen,_IntuitionBase
close_int  move.l   _IntuitionBase,a1  library to close
        CALLSYS  CloseLibrary,_AbsExecBase
close_dos  move.l   _DOSBase,a1       library to close
        CALLSYS  CloseLibrary,_AbsExecBase
exit       clr.l   d0
          rts                logical end of program
```

DrawGrid:

```
; Requires following parameters on entry...
; a0 holds window rastport pointer
; a1 holds image pointer
; d0 holds starting left offset value
; d1 holds starting top offset value
; d2 holds required horizontal block count, ie column
  count
; d3 holds required vertical block count, ie row count
        movem.l  d0-d7/a0-a6, -(sp)  preserve registers
        move.l   _IntuitionBase,a6  set up library base
        move.l   a0,a2               preserve rastport
          pointer
        move.l   a1,a3               preserve image
          pointer
        move.w   d0,d6               d6 = current left
          offset
        move.w   d0,d7               preserve left
          offset for re-use
        move.w   d1,a4               preserve top offset
        move.w   d2,a5               preserve column
          count for re-use
        move.w   ig_Width(a1),d4     get image width in
          d4
```

	move.w	ig_Height(a1),d5	get image height in d5
draw_row	jsr	_LV0DrawImage(a6)	a faster alternative
	subq	#1,d2	decrease count
	beq	next_row	
	move.w	a4,d1	set top offset
	add.w	d4,d6	form new left offset
draw_row2	move.w	d6,d0	needed for library function call
	move.l	a2,a0	restore rastport pointer
	move.l	a3,a1	restore image pointer
	bra	draw_row	keep going
next_row	subq	#1,d3	decrease count
	beq	draw_end	
	move.w	d7,d6	reset start left offset for row
	move.w	a5,d2	reset column count
	move.w	a4,d1	
	add.w	d5,d1	
	move.w	d1,a4	top offset for next row
	bra	draw_row2	
draw_end	movem.l	(sp)+,d0-d7/a0-a6	restore registers
	rts		
_DOSBase	ds.l	1	
_IntuitionBase	ds.l	1	
window_p	ds.l	1	
tags	dc.l	WA_PubScreen	
workbench_p	ds.l	1	


```
dc.l    WA_Left,50
dc.l    WA_Top,20
dc.l    WA_Width,420
dc.l    WA_Height,250
dc.l    WA_DragBar,TRUE
dc.l    WA_Title>window_name
dc.l    TAG_DONE,NULL
dos_name dc.b    'dos.library',NULL
intuition_name dc.b  'intuition.library',NULL
workbench_name dc.b  'Workbench',NULL
window_name dc.b  'My Very First Subroutine!',NULL
Image1:
dc.w    0,0          ;XY origin relative to container
                    TopLeft
dc.w    19,18       ;Image width and height in pixels
dc.w    2           ;number of bitplanes in Image
dc.l    ImageData1 ;pointer to ImageData
dc.b    $0003,$0000 ;PlanePick and PlaneOnOff
dc.l    NULL        ;next Image structure
SECTION Image,DATA_C
ImageData1:
dc.w    $0000,$0000,$0000,$0000,$00C0,$8000,$00C0,$8000
dc.w    $08C2,$8000,$0000,$8000,$00C0,$8000,$00C0,$8000
dc.w    $1CCE,$8000,$1CCE,$8000,$00C0,$8000,$00C0,$8000
dc.w    $0000,$8000,$00C0,$8000,$08C2,$8000,$00C0,$8000
dc.w    $3FFF,$8000,$0000,$0000,$0000,$0000,$7FFF,$C000
dc.w    $40C0,$4000,$50C4,$4000,$40C0,$4000,$41E0,$4000
dc.w    $43F0,$4000,$43F0,$4000,$5FFE,$4000,$5FFE,$4000
dc.w    $43F0,$4000,$43F0,$4000,$41E0,$4000,$50C4,$4000
dc.w    $40C0,$4000,$40C0,$4000,$4000,$4000,$0000,$0000
```

Reaping the Benefits

Now that the tried and tested DrawGrid() subroutine is available it can of course be used to draw tile effects ad infinitum. Just set up the required parameters, block counts etc, identify the image to be used... and then make the subroutine call. It's also quick and easy to swap images and/or alter starting positions and block counts because you just reset those parameters that have changed and then repeat the subroutine call. Here for example is a code fragment which generates one tile effect in the top part of a display, and uses a different effect in the lower part:

```
draw_images  move.l  d0,a1          window address in a1
              move.l  wd_RPort(a1),a0  copy rastport
                                      pointer into a0

              lea    Image1,a1        pointer to image
              moveq  #20,d0          example left offset
              moveq  #15,d1          example top offset
              moveq  #20,d2          example columns
                                      count
              moveq  #5,d3           example rows count
              jsr    DrawGrid         use subroutine
              lea    Image2,a1        pointer to second
                                      image
              moveq  #120,d1         second top offset
              jsr    DrawGrid         re-use subroutine
```

There's plenty of scope for experiment, and experiment you should. Here, to finish this chapter and get you started, is the above type of split tile, twin image, modification of the first program of this chapter. If you are a Devpac or other official include file user just remove the appropriate preliminary definitions and add the include file references mentioned earlier:

* Example CH16-3.s

```
LINKLIB      MACRO
              move.l  a6, -(a7)
              move.l  \2,a6
              jsr    \1(a6)
```

```
                move.l  (a7)+,a6
                ENDM
CALLSYS        MACRO
                LINKLIB _LV0\1,\2
                ENDM

TRUE          EQU      1
NULL          EQU      0
DOS_VERSION   EQU      0
INTUITION_VERSION EQU    36
SECONDS       EQU      50
TIME_DELAY    EQU      10*SECONDS
TAG_DONE      EQU      0
WA_BASE       EQU      $80000063
WA_Left       EQU      WA_BASE+$01
WA_Top        EQU      WA_BASE+$02
WA_Width      EQU      WA_BASE+$03
WA_Height     EQU      WA_BASE+$04
WA_Title      EQU      WA_BASE+$0B
WA_DragBar    EQU      WA_BASE+$1F
WA_PubScreen  EQU      WA_BASE+$16
wd_RPort      EQU      50
ig_Width      EQU      4
ig_Height     EQU      6
_AbsExecBase  EQU      4
_LV0OpenLibrary EQU    -552
_LV0CloseLibrary EQU   -414
_LV0LockPubScreen EQU  -510
_LV0UnlockPubScreen EQU -516
_LV0OpenWindowTagList EQU -606
_LV0CloseWindow EQU    -72
_LV0Delay     EQU     -198
```

```
_LVODrawImage      EQU      -114

start      lea      dos_name,a1      load pointer to
                                library name

                                moveq   #DOS_VERSION,d0    any version will
                                do!

open_dos    CALLSYS  OpenLibrary,_AbsExecBase

                                move.l  d0,_DOSBase      save returned
                                pointer

                                beq     exit              check open OK?

open_int    lea      intuition_name,a1  load pointer to
                                library name

                                moveq   #INTUITION_VERSION,d0  specify
                                minimum lib
                                version

                                CALLSYS  OpenLibrary,_AbsExecBase

                                move.l  d0,_IntuitionBase  save returned
                                pointer

                                beq     close_dos          check open OK?

lock_screen lea      workbench_name,a0  pointer to
                                screen name

                                CALLSYS  LockPubScreen,_IntuitionBase

                                move.l  d0,workbench_p    save returned
                                pointer

                                beq     close_int          check return value?

open_window move.l  #NULL,a0

                                lea     tags,a1            our tag list

                                CALLSYS  OpenWindowTagList,_IntuitionBase

                                move.l  d0>window_p        pointer to our
                                window

                                beq     unlk_screen

draw_images move.l  d0,a1              window address in
                                a1

                                move.l  wd_RPort(a1),a0    copy rastport
                                pointer into a0

                                lea     Image1,a1          pointer to image
```

```
        moveq    #20,d0          example left offset
        moveq    #15,d1          example top offset
        moveq    #20,d2          example columns
                                count
        moveq    #5,d3           example rows count
        jsr      DrawGrid        use subroutine
        lea     Image2,a1        pointer to second
                                image
        moveq    #120,d1         second top offset
        jsr      DrawGrid        reuse subroutine
wait    move.l    #TIME_DELAY,d1
        CALLSYS  Delay,_DOSBase
        move.l   window_p,a0     window to close
        CALLSYS  CloseWindow,_IntuitionBase
unlk_screen  move.l    #NULL,a0   screen name not
                                needed
        move.l   workbench_p,a1  screen to unlock
        CALLSYS  UnlockPubScreen,_IntuitionBase
close_int   move.l    _IntuitionBase,a1  library to close
        CALLSYS  CloseLibrary,_AbsExecBase
close_dos   move.l    _DOSBase,a1      library to close
        CALLSYS  CloseLibrary,_AbsExecBase
exit        clr.l    d0
        rts                logical end of program
```

DrawGrid:

```
; Requires following parameters on entry...
; a0 holds window rastport pointer
; a1 holds image pointer
; d0 holds starting left offset value
; d1 holds starting top offset value
; d2 holds required horizontal block count, ie column
    count
; d3 holds required vertical block count, ie row count
```

	movem.l	d0-d7/a0-a6, -(sp)	preserve registers
	move.l	_IntuitionBase, a6	set up library base
	move.l	a0, a2	preserve rastport pointer
	move.l	a1, a3	preserve image pointer
	move.w	d0, d6	d6 = current left offset
	move.w	d0, d7	preserve left offset for re-use
	move.w	d1, a4	preserve top offset
	move.w	d2, a5	preserve column count for re-use
	move.w	ig_Width(a1), d4	get image width in d4
	move.w	ig_Height(a1), d5	get image height in d5
draw_row	jsr	_LV0DrawImage(a6)	a faster alternative
	subq	#1, d2	decrease count
	beq	next_row	
	move.w	a4, d1	set top offset
	add.w	d4, d6	form new left offset
draw_row2	move.w	d6, d0	needed for library function call
	move.l	a2, a0	restore rastport pointer
	move.l	a3, a1	restore image pointer
	bra	draw_row	keep going
next_row	subq	#1, d3	decrease count
	beq	draw_end	
	move.w	d7, d6	reset start left offset for row

Amiga Insider Guide

```
        move.w    a5,d2                reset column count
        move.w    a4,d1
        add.w     d5,d1
        move.w    d1,a4                top offset for next
                                        row
        bra      draw_row2
draw_end  movem.l  (sp)+,d0-d7/a0-a6    restore registers
        rts

_DOSBase      ds.l    1
_IntuitionBase ds.l    1
window_p      ds.l    1
tags          dc.l    WA_PubScreen
workbench_p   ds.l    1
              dc.l    WA_Left,50
              dc.l    WA_Top,20
              dc.l    WA_Width,420
              dc.l    WA_Height,250
              dc.l    WA_DragBar,TRUE
              dc.l    WA_Title>window_name
              dc.l    TAG_DONE,NULL
dos_name      dc.b    'dos.library',NULL
intuition_name dc.b    'intuition.library',NULL
workbench_name dc.b    'Workbench',NULL
window_name   dc.b    'Getting Clever Eh!',NULL
```

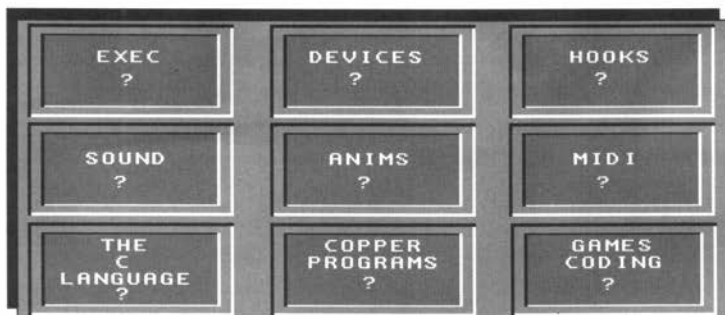
SECTION Image,DATA,CHIP

Image1:

```
dc.w    0,0                ;XY origin relative to container
                          TopLeft
dc.w    19,18              ;Image width and height in pixels
dc.w    2                  ;number of bitplanes in Image
dc.l    ImageData1        ;pointer to ImageData
```

```
dc.b  $0003,$0000  ;PlanePick and PlaneOnOff
dc.l  NULL          ;next Image structure
ImageData1:
dc.w  $0000,$0000,$0000,$0000,$00C0,$8000,$00C0,$8000
dc.w  $08C2,$8000,$0000,$8000,$00C0,$8000,$00C0,$8000
dc.w  $1CCE,$8000,$1CCE,$8000,$00C0,$8000,$00C0,$8000
dc.w  $0000,$8000,$00C0,$8000,$08C2,$8000,$00C0,$8000
dc.w  $3FFF,$8000,$0000,$0000,$0000,$0000,$7FFF,$C000
dc.w  $40C0,$4000,$50C4,$4000,$40C0,$4000,$41E0,$4000
dc.w  $43F0,$4000,$43F0,$4000,$5FFE,$4000,$5FFE,$4000
dc.w  $43F0,$4000,$43F0,$4000,$41E0,$4000,$50C4,$4000
dc.w  $40C0,$4000,$40C0,$4000,$4000,$4000,$0000,$0000
Image2:
dc.w  0,0           ;XY origin relative to container
                        TopLeft
dc.w  19,18         ;Image width and height in pixels
dc.w  2             ;number of bitplanes in Image
dc.l  ImageData2   ;pointer to ImageData
dc.b  $0003,$0000  ;PlanePick and PlaneOnOff
dc.l  NULL          ;next Image structure
ImageData2:
dc.w  $0000,$0000,$FFFF,$C000,$FFFF,$C000,$F007,$C000
dc.w  $F007,$C000,$F007,$C000,$F007,$C000,$F007,$C000
dc.w  $F007,$C000,$F007,$C000,$F007,$C000,$F007,$C000
dc.w  $F007,$C000,$F007,$C000,$F007,$C000,$FFFF,$C000
dc.w  $FFFF,$C000,$0000,$0000,$0000,$0000,$0000,$0000
dc.w  $0FF8,$0000,$0FF8,$0000,$0FF8,$0000,$0FF8,$0000
dc.w  $0FF8,$0000,$0FF8,$0000,$0FF8,$0000,$0FF8,$0000
dc.w  $0FF8,$0000,$0FF8,$0000,$0FF8,$0000,$0FF8,$0000
dc.w  $0000,$0000,$0000,$0000
END
```


er...?



In this book I've attempted to introduce 68000 assembly language specifically from an Amiga oriented viewpoint and my main aim was to provide you with the necessary easy footholds to get into low-level Amiga programming as quickly as possible. In the past many Amiga programmers have felt that this simply couldn't be done but I've always been convinced that it could and hopefully this book proves it. Needless to say, many of the more complex Amiga programming areas have had to be avoided.

By now you should have a good idea of what assembly language is all about, and know enough about the Amiga's operating system for the words "Amiga system library call" *not* to produce a cold sweat. In fact, given the details of a function in a particular library, you should by now be able to sketch out (and understand) code which opens the library, uses the function, and checks any returned values and so forth. This is an important achievement because a good 90% of all the code written by most 68000 coder is based on the use of pre-written Amiga library routines!

There are of course plenty of things I've not mentioned including those topics that are readily picked up from general, as opposed to Amiga specific, 680x0 books. The convoluted tricks that many 68000 programmers use to ensure that their code is compact was another topic placed on the back burner although perhaps Chapter 16 provided a little food for thought. Such things can and will be picked up from both the more advanced 68000 books and your own experience!

One of the other areas deliberately avoided, because newcomers find it a difficult topic to come to terms with is the use of something known as *Amiga start-up code*. Perhaps, however, a few words about this subject are appropriate at this stage. Shell based programs are easy to run. You just type the name of the program at the Shell prompt along with any required arguments. With Workbench programs you double-click on an associated program icon and herein lies yet another Amiga story. Programs which are to run from the Workbench have to execute some rather complicated message-oriented code and both Commodore, and indeed other developers, offer pre-assembled modules which take care of the awkward code details. These modules, known as start-up code modules, just need to be linked, as the first module, with the actual assembled program code.

Unfortunately start-up modules do tend to vary somewhat and so you need to get details of any modules supplied from your assembler documentation. Depending on their source, start-up modules are likely to do any number of things as well as handling the initial Workbench start-up message operations.

Start-up modules supplied with most C compilers, for instance, open the DOS library, set up standard I/O handles and so on. This means that if you are linking with such a module you do not have to write code explicitly to do these things. You do, however, have to conform to any conventions expected – start-up modules designed for use in C programs, for instance, expect the first label in a program, ie the start location of the real program code, to be labelled as `_main`. It is wisest at least to make a rough preliminary check of the contents of any start-up modules you are tempted to use in order to avoid linker errors due to missing connecting labels and so on.

Once the start-up code has been linked to a suitable program it is only necessary to create an icon file for the program – using the same filename as the program but with a `.info` filename extension – to allow the program to be run from Workbench by double-clicking on its icon. What do I mean by “suitable program”? It is best if I explain what

unsuitable programs are – they are programs which use DOS-oriented I/O operations, such as the examples discussed in Chapter 12 that wrote messages back to a Shell window. Programs started from the Workbench have no default I/O handles available and if you just tag an icon on to such programs and try to run them from the workbench, you'll cause the O/S to crash! Depending on the start-up module the same thing may well happen even when the start-up code is utilised. In other cases the start-up code thoughtfully opens a default console window which provides a sink for any DOS oriented program output. You'll find that most assemblers and compilers provide detailed notes about the start-up facilities they provide and the best idea is to read them. Like many things Amiga-wise all this takes time but after a while, and with a little experimentation, things will eventually make sense. You'll find additional Workbench-oriented examples on the associated Insider Guide disk.

And talking of complexity. This Insider Guide in dealing with low-level 68000 coding has tackled a subject which is rather more difficult to get to grips with than most other Amiga areas. Hopefully you've not found the path so far too difficult but do remember that we have only travelled a few steps down what may sometimes become a difficult road. Be in no doubt that the Amiga's operating system is not something you learn about in just a few days, weeks or even months. There is, however, plenty of good news as well because programming the Amiga can be both addictive and enjoyable and all serious programmers will tell you the same thing – the more you learn, the more you'll want to learn! As you progress you will doubtless follow your own path in terms of what you choose, Amiga-wise, to take an interest in but nowadays, with plenty of more advanced books to choose from, you'll never be far from help.

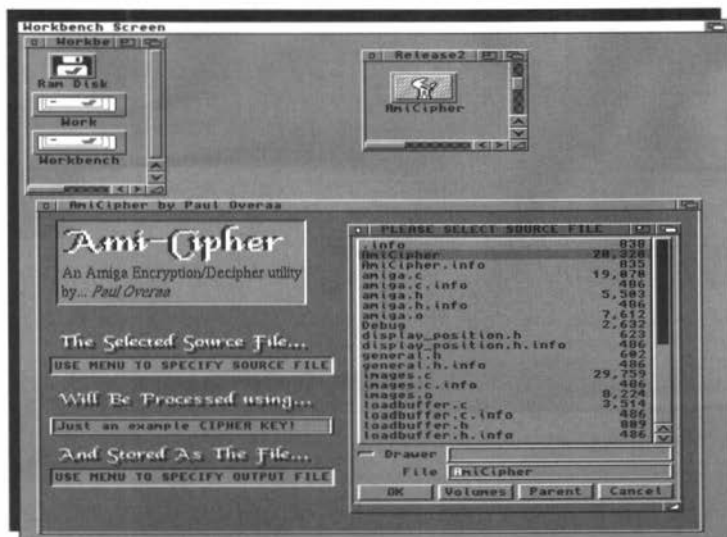
Regardless of the directions in which you travel you will almost certainly get to a point where more and more reliance has to be placed on the Amiga's official system documentation. The Addison Wesley Amiga ROM Kernel Reference Manuals are the ultimate source of Amiga system information and are worth their weight in gold but I would be less than honest if I told you that some experience with the C language would not be an advantage to you at this stage. Why? It's because, in the main, the official RKM reference manuals use C for their programming examples! Because of this my experience is that all programmers, including those whose sole interest was programming at the 680x0 microprocessor level, eventually need to come to terms with C just in order to cope with the official Amiga documentation.

This, from a long term viewpoint, is something which you should clearly keep in mind.

From the point of view of this current book however you have reached the end of the road and hopefully you feel ready to move on to more advanced books. Have fun, enjoy your coding, and make the most of one of the most accessibly priced and brilliant computer systems the world has ever seen!

The details provided here cover just some of the main instructions used in this book plus a few related ones.

Additional notes about their uses, the addressing mode restrictions, flag effects and so on have also been given.



This appendix covers only the main instructions used in this book and a few related ones. Additional notes about their uses, the addressing mode restrictions, flag effects and so on have also been given but for full details you should consult the references given in the bibliography.

The range of 68000 instructions can be roughly divided into the following classes:

- Data Movement instructions
- Flow Control (Jump, branch type) instructions
- Logical, shift and rotate type instructions
- Bit manipulation instructions
- Arithmetic instructions.

Many 68000 instructions can work with byte, word and long word operands. However, byte size values are not allowed if the destination or source operand is an address register.

Effective Address

Motorola 68000 literature uses the term *effective address* to refer to the address that the processor ultimately uses. For instructions which identify an operand, do something, and then store a result there is an effective source address and an effective destination address. Usually the context of the instruction makes it easy to identify these separate entities. When a general effective address needs to be stated, as opposed to a specific addressing mode description, it is common practice to use the term <ea>.

Op-Codes

The part of the binary machine code instructions which holds the real *68000-understandable* information about which operation the processor should perform is known as the operation code or op-code part of the instruction.

Sign Extension

Some 68000 instructions sign-extend byte or word data, ie they propagate the sign bit (bit 7 in the case of byte data or bit 15 for word sized operands) to produce a 32 bit value.

Notes on An/Dn Name Conventions

When talking generally about address registers and data registers it is common practice to use the terms An and Dn to indicate *any* address register or any data register.

60000 Addressing Modes

One of the most powerful features of the Motorola 68000 device is the rich variety of addressing modes that are available. Most processor instructions work on a piece of data called the operand and this data has to be stored somewhere. Many instructions use some real or implied source address (the effective source address), do something, and then transfer the result to some destination address (the effective destination address). In short the processor's addressing modes enable these source and destination addresses to be specified. Here's the run-down on the basic 68000 addressing schemes:

Inherent Addressing

This is one of the addressing modes which do not involve the specifying of memory locations because the processor knows which addresses it should use from the instruction op-code. The 68000's return-from-subroutine, rts, instruction for instance *inherently knows* that the stack pointer register is to be used to move data to and from memory – the details are built into the instruction itself. This is why the programmer does not need to specify an addressing mode for rts, and why none are listed.

Register Addressing

This is perfectly straightforward: Register addressing simply means that the operands reside in a processor's register and so no memory address information is needed. The official documentation splits register addressing into data and address register addressing but, for most practical purposes, the distinction is neither here nor there.

Immediate Addressing

Another straightforward mode where the data in question, ie the operand itself, is placed immediately after the instruction op-code in memory. In other words the effective address is the value of the program counter after the op-code part of the instruction has been fetched. The Motorola 68000 has long word, word and byte oriented immediate instructions but, in the latter case, the immediate data still gets stored as a word. The byte data is placed in the low-order part of the word and the upper byte is set to all zeros.

Absolute Addressing

This mode is also called direct addressing and actually consists of two schemes. With absolute long addressing the effective address used by the processor is the address contained in the four bytes (ie the long word) which follows the op-code and so this scheme can be used to address any memory location within a 32 bit addressing range.

A word (two-byte) addressing scheme known as absolute short addressing is also available and here only the lower 16 bits of an address need be specified – the upper half of the address is obtained by sign-extending bit 15 of the specified short address. This mode is quicker and more memory efficient than absolute long addressing but only addresses in the lower and upper 32k of address space (0000000 hex to 00007fff hex and ffff8000 hex to ffffffff hex) can be specified in this way.

Address Register Indirect Addressing

Here the address of the operand is held in an address register and so this scheme is not the same as conventional *indirect addressing* where the address of the operand is held in a memory location. Register indirect addressing is nevertheless a very powerful addressing mode and is indicated by placing parentheses around the register name. For example the instruction `move.b (a2), d0` will copy the contents of the byte whose address is in register `a2` into register `d0`.

Address Register Indirect with Displacement

This mode allows a fixed, but programmer defined, constant value to be added to the indirectly specified address. The displacement itself gets stored immediately after the op-code in memory and the effective address used by the processor is the sum of the contents of the address register and the specified displacement. For example the instruction

`move.b 20(a2), d0`

copies the contents of the byte whose address is formed by 'adding 20 to the address in register `a2`' into register `d0`.

You can find some examples of this addressing mode within this book for storing and retrieving items from Amiga system defined structures.

Address Register Indirect with Postincrement

This mode provides for the automatic incrementing of a specified address *after* it has been used. Byte, word and long word sizes may be specified and the processor increments the address by 1, 2 or 4 accordingly. The mode is specified by placing a plus sign after the normal

indirect addressing scheme. The instruction

`move.b (a2)+, d0`

copies the contents of the byte whose address is in register `a2` into register `d0` and, having done, that the contents of address register `a2` are automatically incremented by 1. This mode is convenient for handling lists of byte, word and long word values.

Address Register Indirect with Predecrement

This mode is similar to the above but it provides for the automatic decrementing of a specified address *before* it has been used. Again byte, word and long word sizes may be specified and the processor decrements the address by 1, 2 or 4 accordingly. The mode is specified

by placing a minus sign before the normal indirect addressing scheme. For example, the instruction

```
move.b -(a2), d0
```

copies the contents of the byte whose address is in register a2 into register d0 and having done that the contents of address register a2 are automatically decreased by 1. This mode is convenient for handling lists of byte, word and long word values. Chapter Eleven outlines the reasons why the addresses are decremented before use and, in the case of the previous mode, incremented after use.

Address Register Indirect with Index and Displacement

This is another useful, but initially confusing, 68000 addressing mode. The effective address is the sum of three separate addresses: an address register specified indirect address, an *index* value held in an address or data register (long or word values may be specified), and a programmer defined constant displacement.

The Motorola assembly language syntax for this addressing mode requires that the displacement is specified as with the basic register indirect addressing scheme but that the address register itself, and the index register, be enclosed within parentheses. The address register should be specified first, and the two enclosed items must be comma delimited. This is best illustrated by example and the instruction:

```
move.l 20(a0,d0.1), d2
```

forms an effective source address by taking the contents of register a0, adding the full 32 bit contents of register d0, and then adding 20 to the resulting address. In the case of the example statement the operand is retrieved from that address and placed in register d2.

Program Counter Relative with Displacement

Addressing modes that use offsets from the program counter, as opposed to absolute addresses are known as relative addressing modes. It's the microprocessor equivalent of you giving someone a friend's address by saying 'they live six doors further up' rather than saying 'they live at number 230'. The 68000 branch instructions automatically use relative addressing but many instructions allow explicit use of relative addressing with the option to include a displacement value. This mode, which we've not used in this book, is equivalent to the 'address register indirect with displacement' mode except for the fact that the program counter is used as the base register. It becomes useful when it is necessary to write position-independent 68000 code.

Program Counter Relative with Index and Displacement

Another addressing mode that has not concerned us in this book. In this case the basic relative addressing scheme is supplemented by both an address register or data register index value and a programmer-specified constant displacement. This mode is equivalent to the 'address register indirect with index and displacement' mode except for the fact that the program counter is used as the base register. Again it becomes useful when it is necessary to write truly position-independent 68000 code.

Data Movement Instructions

Mnemonic: LEA – Load Effective Address		
Purpose:	Loads an address register with a processor determined effective address.	
Addressing Modes:	Source	Destination
Data register direct		
Address register direct		X
Address register indirect	X	
Postincrement register indirect		
Predecrement register indirect		
Register indirect with displacement	X	
Register indirect with index	X	
Absolute Short	X	
Absolute Long	X	
PC relative with displacement	X	
PC relative with index	X	
Immediate		
Flags affected:	X	N Z V C
	-	- - - -

Notes: This instruction allows you to load an address register with an effective source address, ie the source address specified by virtue of a chosen addressing mode.

Example: The effective source address for the instruction:

```
lea new_window, a0
```

is the address of the location which has been labelled new_window (this is an example of absolute addressing).

Mnemonic: MOVE - Move Data from Source to Destination		
Purpose: Copies a source operand to specified destination		
Addressing Modes:	Source	Destination
Data register direct	X	X
Address register direct	X*	
Address register indirect	X	X
Postincrement register indirect	X	X
Predecrement register indirect	X	X
Register indirect with displacement	X	X
Register indirect with index	X	X
Absolute Short	X	X
Absolute Long	X	X
PC relative with displacement		
PC relative with index		
Immediate	X	
Flags affected:	X	N Z V C - Y Y 0 0

Notes: You can find plenty of examples of move instructions within this book. See the notes about the movea instruction and also be aware that address register direct addressing is *not* allowed if specified data size is byte!

There are a number of specialised move instructions which allow reading from and writing data to the whole status register or just the lower byte that holds the condition codes allowing you to forcibly clear/set the N, Z, V, C and X flags. Some of these instructions are privileged on one or more members of the 680x0 family and you should consult the official 680x0 documentation for details.

Mnemonic: MOVEA – Move Address		
Purpose: Loads an address register with a value		
Addressing Modes:	Source	Destination
Data register direct	X	
Address register direct	X	X
Address register indirect	X	
Postincrement register indirect	X	
Predecrement register indirect	X	
Register indirect with displacement	X	
Register indirect with index	X	
Absolute Short	X	
Absolute Long	X	
PC relative with displacement	X	
PC relative with index	X	
Immediate	X	
Flags affected: X N Z V C - - - - -		

Notes: Only word or long word operands can be specified and if the operation is word-sized then the address is sign-extended. Most 68000 assemblers accept

move <ea>,An

as well and the latter convention has been adopted in this book. You do however need to remember that when move is used to load an

address register it is really a movea instruction and the flags are *not* affected.

Mnemonic: MOVEM – Move Multiple Registers to Memory					
Purpose: Copies multiple registers to memory					
Addressing Modes:	Source	Destination			
Data register direct					
Address register direct					
Address register indirect		X			
Postincrement register indirect					
Predecrement register indirect		X			
Register indirect with displacement		X			
Register indirect with index		X			
Absolute Short		X			
Absolute Long		X			
PC relative with displacement					
PC relative with index					
Immediate					
Flags affected:	X	N	Z	V	C
	-	-	-	-	-

Notes: The main use of this instruction is for storing groups of registers on the stack. For example:

movem.l d0-d7/a0-a6, -(a7) push all registers onto the stack

Mnemonic: MOVEM – Move Multiple Registers From Memory		
Purpose: Copies multiple registers from memory		
Addressing Modes:	Source	Destination
Data register direct		
Address register direct		
Address register indirect	X	
Postincrement register indirect	X	
Predecrement register indirect		
Register indirect with displacement	X	
Register indirect with index	X	
Absolute Short	X	
Absolute Long	X	
PC relative with displacement		
PC relative with index		
Immediate		
Flags affected:	X N Z V C	
	- - - - -	

Notes: The main use of the instruction is for retrieving registers from stack. For example:

movem.l (a7)+,d0-d7/a0-a6 pull all registers from the stack.

Mnemonic: MOVEQ – Move Quick					
Purpose: Copies immediate data to a specified data register					
Flags affected:	X	N	Z	V	C
	-	Y	Y	0	0

Notes: This instruction provides a quick and efficient way to set a data register to a particular value which can be from -128 to +127 decimal. Most 68000 assemblers, given an immediate addressing move instruction, generate moveq instructions where possible. For example:

	Before	After
moveq#\$58, d0	d0=ffffff	d0=0000058

Flow Control Instructions

These instructions work by altering the contents of the

program counter.

Mnemonic: Bcc – Branch Conditionally					
Purpose:	Transfers program control using relative addressing				
Flags affected:	X	N	Z	V	C
	-	-	-	-	-

Notes: The branch data sizes may be byte or word so these instructions can branch in an area of 32K. When using a branch with a byte offset you can in fact put a .s (for short) suffix behind the instruction eg beq.s HERE. Similarly when using a branch with a word offset you can use a .w suffix – eg beq.w HERE. Most assemblers determine whether the short or word form is needed automatically and optimise word-branches to byte-branches whenever it is possible.

These instructions test a combination of the NZVC-flags in the status register and conditionally perform a branch to another address. If the testing of the condition codes is true, then the branch is taken, otherwise the instruction immediately following the bcc instruction is executed.

Fourteen variations of this instruction are available and a related bra (branch always) instruction adds another condition to the testable set:

bcc: where cc stands for carry clear. The branch is taken if the carry (C) bit is 0. This instruction is often used in combination with shift and rotate operations.

bcs: where cs stands for carry set. The branch is taken if the carry (C) bit is 1.

- beq: where eq stand for equal. The branch is taken if the zero (Z) bit is 1. This instruction, as we've seen many times within this book, is frequently used after tst and cmp type instructions.
- bne: where ne stands for not equal. The branch is taken if the zero (Z) bit is 0. This instruction is of course the opposite of beq.
- bpl: where pl stands for plus. The branch is taken if the negative (N) bit is 0.
- bmi: where mi stands for minus. The branch is taken if the negative (N) bit is 1.
- bvc: where vc stands for overflow clear. The branch is taken if the overflow (V) bit is 0 (this instruction is often used in conjunction with arithmetic instructions like add, mul and so on).
- bvs: where vs stands for overflow set. The branch is taken if the overflow (V) bit is 1.
- bge: where ge stands for greater or equal. The branch is taken when the negative (N) and overflow (V) bits contain the same value.
- bgt: where gt stands for greater than. The branch is taken in cases where either N= 1, V=1 and Z=0 or N=V=Z=0.
- ble: where le stands for lower or equal. This branch is taken in cases where Z=1 or the N and V bits contain different values.
- blt: where lt stands for less than. This branch is taken if the negative (N) and overflow (V) bits contain different values.
- bhi: where hi stands for higher. This branch is taken if the negative (N) and overflow (V) bits contain the same value.
- bls: where ls stands for lower or same. This branch is taken if the carry (C) and zero (Z) bits contain different values.
- bra: branch always. This instruction is commonly seen at the end of a loop to force control back to the top of the loop.

Mnemonic: Bcc – Branch Conditionally

Purpose:	Transfers program control using relative addressing				
----------	---	--	--	--	--

Flags affected:	X	N	Z	V	C
	-	-	-	-	-

Notes: See jsr notes below

Mnemonic: JSR – Jump to Subroutine					
Purpose: Transfers program control to a subroutine					
Addressing Modes:	Source	Destination			
Data register direct					
Address register direct					
Address register indirect		X			
Postincrement register indirect					
Predecrement register indirect					
Register indirect with displacement		X			
Register indirect with index		X			
Absolute Short		X			
Absolute Long		X			
PC relative with displacement		X			
PC relative with index		X			
Immediate					
Flags affected:	X	N	Z	V	C
	-	-	-	-	-

Notes: The bsr (branch to subroutine) and jsr (jump to subroutine) instructions are used for calling subroutines. The bsr form is a relative branch with a range of 32K. For subroutine calls beyond this range the jsr instruction should be used but having said that most assemblers would optimise jsr to bsr when possible (bsr is more efficient). When executing a bsr/jsr instruction, the 68000 pushes the program counter on the stack and then re-loads it with the target address.

Mnemonic: RTS – Return From Subroutine					
Purpose:	Transfers control to a stack-retrieved address				
Flags affected:	X	N	Z	V	C
	-	-	-	-	-

Notes: In a sense this is the counterpart of the bsr/jsr instructions because it reloads the program counter register with the value on top of the stack (this value will usually have been put there by a bsr or jsr instruction).

Mnemonic: JMP – Jump					
Purpose:	Transfers program control to a specified address				
Addressing Modes:	Source	Destination			
Data register direct					
Address register direct					
Address register indirect		X			
Postincrement register indirect					
Predecrement register indirect					
Register indirect with displacement		X			
Register indirect with index		X			
Absolute Short		X			
Absolute Long		X			
PC relative with displacement		X			
PC relative with index		X			
Immediate					
Flags affected:	X	N	Z	V	C
	-	-	-	-	-

Notes: This instruction is a variant of the move instruction but in this case the destination register, namely the program counter, is inherently defined. You could therefore just as easily use `move.l (ea),PC` instead of `jmp <ea>`.

Logical Operations

Mnemonic: ANDI – AND Immediate		
Purpose: Bitwise AND of immediate data source with destination		
Addressing Modes:	Source	Destination
Data register direct		X
Address register direct		
Address register indirect		X
Postincrement register indirect		X
Predecrement register indirect		X
Register indirect with displacement		X
Register indirect with index		X
Absolute Short		X
Absolute Long		X
PC relative with displacement		
PC relative with index		
Immediate	X	
Status register		X
Flags affected:	X N Z V C	
	- Y Y 0 0	

Notes: In addition to the more conventional register and memory usage the destination may be the condition codes or the whole of the

68000 status register. In the latter case the instruction is privileged. As an example:

Instruction	Before	After
andi.b #7,d0	d0=9999aaaa	d0=9999aaa0

Mnemonic: EORI – Exclusive OR Immediate					
Purpose:	Bitwise Exclusive-OR of immediate data source with destination				
Addressing Modes:	Source	Destination			
Data register direct		X			
Address register direct					
Address register indirect		X			
Postincrement register indirect		X			
Predecrement register indirect		X			
Register indirect with displacement		X			
Register indirect with index		X			
Absolute Short		X			
Absolute Long		X			
PC relative with displacement					
PC relative with index					
Immediate	X				
Status register		X			
Flags affected:	X	N	Z	V	C
	-	Y	Y	0	0

Notes: Destination may be condition codes or the whole of the 68000 status register. In the latter case the instruction is privileged. For example:

	Before	After
eori.b #\$ff,d6	d6=eeeeee30	d6=eeeeecf

Mnemonic: NOT – Logical Complement					
Purpose: Performs a bitwise complement of an operand					
Addressing Modes:	Source	Destination			
Data register direct		X			
Address register direct					
Address register indirect		X			
Postincrement register indirect		X			
Predecrement register indirect		X			
Register indirect with displacement		X			
Register indirect with index		X			
Absolute Short		X			
Absolute Long		X			
PC relative with displacement					
PC relative with index					
Immediate					
Flags affected:	X	N	Z	V	C
	-	Y	Y	0	0

Notes: The instruction `not.w An` has the same effect as `eor.w #$ffff,An`.

Mnemonic: ORI – Inclusive OR Immediate		
Purpose: Performs bitwise OR using immediate data source		
Addressing Modes:	Source	Destination
Data register direct		X
Address register direct		
Address register indirect		X
Postincrement register indirect		X
Predecrement register indirect		X
Register indirect with displacement		X
Register indirect with index		X
Absolute Short		X
Absolute Long		X
PC relative with displacement		
PC relative with index		
Immediate	X	
Status register		X
Flags affected:	X N Z V C	
	- Y Y 0 0	

Notes: Destination may be condition codes or the whole of the 68000 status register. In the latter case the instruction is privileged.

Example

	Before	After
ori.b #ff,d0	d0=efefefef	d0=efefefff

Shift and Rotate Operations

A whole range of left and right shifts and rotate instructions

are available on the 68000 processor. Here are two examples:

Mnemonic: ASL – Arithmetic Shift Left in Data Register					
Purpose:	Left shifts the contents of a data register				
Flags affected:	X	N	Z	V	C
	Y	Y	Y	Y	Y

Notes: This instruction arithmetically left shifts the contents of the data register and the carry (C) and extend (X) flags receive the last bit shifted out. The shift count may be specified either by another data register or by immediate data and, in the latter case, a shift count in the range 1-8 may be specified. When a data register is used counts in the range 0-63 are allowed.

ASL instructions can be used as a fast form of multiplying an operand by a factor of two. The lower bit of the destination is always set to zero. Example:

	Before	After
asl.l #4,d1	d1=0000000f	d1=000000f0

Mnemonic: ASL - Arithmetic Shift Left in Memory					
Purpose: Left shifts the contents of a memory location					
Addressing Modes:		Source	Destination		
Data register direct					
Address register direct					
Address register indirect			X		
Postincrement register indirect			X		
Predecrement register indirect			X		
Register indirect with displacement			X		
Register indirect with index			X		
Absolute Short			X		
Absolute Long			X		
PC relative with displacement					
PC relative with index					
Immediate					
Flags affected:	X	N	Z	V	C
	Y	Y	Y	Y	Y

Notes: This form of the instruction is restricted to a one-bit shift and can only be used for word sized operands.

Bit Manipulation Instructions

Again many instructions exist. One example is:

Mnemonic: BTST – Test a Bit		
Purpose: Tests an operand bit and sets zero flag accordingly		
Addressing Modes	Source	Destination
Data register direct	X	X
Address register direct		
Address register indirect		X
Postincrement register indirect		X
Predecrement register indirect		X
Register indirect with displacement		X
Register indirect with index		X
Absolute Short		X
Absolute Long		X
PC relative with displacement		X
PC relative with index		X
Immediate	X	
Flags affected:	X N Z V C	
	- - Y - -	

Arithmetic Instructions

Mnemonic: ADD – Add Binary					
Purpose: Add source operand to data register destination					
Addressing Modes:	Source	Destination			
Data register direct	X	X			
Address register direct	X*				
Address register indirect	X				
Postincrement register indirect	X				
Predecrement register indirect	X				
Register indirect with displacement	X				
Register indirect with index	X				
Absolute Short	X				
Absolute Long	X				
PC relative with displacement	X				
PC relative with index	X				
Immediate	X				
Flags affected:	X	N	Z	V	C
	Y	Y	Y	Y	Y

Notes: Address register direct addressing is not allowed for byte size operations. Two forms of the instruction are available.

Example:

	Before	After
add.w d0,d2	d0=00000011	d0=00000011
	d2=0000FFFA	d2=0000000B
	XNZVC=00000	XNZVC=11001

Mnemonic: ADDI – Add Immediate					
Purpose:	Add immediate data to specified operand				
Flags affected:	X	N	Z	V	C
	Y	Y	Y	Y	Y

Notes: This instruction has exactly the same characteristics as the 'ADD using a data register source' instruction, except that immediate addressing is used to specify the source – ie the source must be a constant.

Mnemonic: ADDQ – Add Quick					
Purpose:	Add data specified in instruction code to operand				
Flags affected:	X	N	Z	V	C
	Y	Y	Y	Y	Y

Notes: Similar in effect to ADDI but the value is built into the instruction code itself. The immediate values in the source field are restricted to the range 1 to 8. This instruction is the fastest way to add a number between 1 to 8 to a destination operand.

Additional notes on ADD, ADDI, ADDQ: Most assemblers optimise your code automatically and so if, for example, you write `add #1,Dn` then the assembler translates it automatically to `addq #1,Dn` thus reducing the size of the object code and saving a few clock cycles of execution time.

Mnemonic: CLR – Clear and Operand					
Purpose: Sets specified register or memory location to zero					
Addressing Modes:	Source	Destination			
Data register direct		X			
Address register direct					
Address register indirect		X			
Postincrement register indirect		X			
Predecrement register indirect		X			
Register indirect with displacement		X			
Register indirect with index		X			
Absolute Short		X			
Absolute Long		X			
PC relative with displacement					
PC relative with index					
Immediate					
Flags affected:	X	N	Z	V	C
	-	0	1	0	0

Notes: You cannot use `clr` to clear an address register but most assemblers allow instructions like `clr a0` to be written and then substitute a `sub.l a0,a0` instruction which has the same effect (`sub.l a0,a0` in the case of `clr a0`).

Example:

	Before	After
<code>clr.w d0</code>	<code>d0=bbbbbbbb</code>	<code>d0=00000000</code>
	<code>NZVC=1011</code>	<code>NZVC=0100</code>

Mnemonic: CMP - Compare		
Purpose: Compares operand with a data register and sets flags		
Addressing Modes:	Source	Destination
Data register direct	X	X
Address register direct	X	
Address register indirect	X	
Postincrement register indirect	X	
Predecrement register indirect	X	
Register indirect with displacement	X	
Register indirect with index	X	
Absolute Short	X	
Absolute Long	X	
PC relative with displacement	X	
PC relative with index	X	
Immediate	X	
Flags affected:	X N Z V C	
	- Y Y Y Y	

Notes: CMP is a subtraction instruction which affects only the condition codes.

Example:

	Before	After
cmp.l d2,d3	d2=00000001	d2=00000001
	d3=00000002	d3=00000002
	NZVC=1111	NZVC=0000

Mnemonic: CMPA – Compare Address					
Purpose:	As CMP but uses an address register as destination				
Flags affected:	X	N	Z	V	C
	-	Y	Y	Y	Y

Notes: This instruction differs only from CMP in that the second operand is an address register and that the data size cannot be byte.

Mnemonic: CMPI – Compare Immediate					
Purpose:	As CMP but compares against immediate data				
Flags affected:	X	N	Z	V	C
	-	Y	Y	Y	Y

Mnemonic: CMPM – Compare Memory					
Purpose:	Compares contents of two memory locations				
Flags affected:	X	N	Z	V	C
	-	Y	Y	Y	Y

Notes: Similar to CMP, but both the source and destination operands must use postincrement addressing. This instruction is used to compare areas of memory.

Additional note on all CMPx instructions: Most assemblers accept instructions like `cmp.w (a0)+,(a1)+` and `cmp.l #3,d0`

Mnemonic: DIVS – Signed Divide					
Purpose:	Divides a 32 bit destination by a 16 bit source				
Flags affected:	X	N	Z	V	C
	-	Y	Y	Y	0

Notes: This instruction performs a division between two signed numbers. The destination register is always a longword and the source operand is always a word. After the division the destination operand contains the result. The quotient is always in the lower word and the remainder is always in the high order word of the data register!

Mnemonic: MULS – Signed Multiply					
Purpose:	Multiplies two 16 bit operands				
Flags affected:	X	N	Z	V	C
	-	Y	Y	0	0

Notes: This instruction performs a multiplication of the source and destination operand, putting the result in the destination operand.

Mnemonic: SUBI – Subtract Immediate					
Purpose:	Subtract immediate data from specified operand				
Flags affected:	X	N	Z	V	C
	Y	Y	Y	Y	Y

Notes: This instruction, except for the fact that subtraction is involved, has exactly the same characteristics as the ADDI instruction.

Mnemonic: SUBQ – Subtract Quick					
Purpose:	Subtract data specified in instruction code				
Flags affected:	X	N	Z	V	C
	Y	Y	Y	Y	Y

Notes: Similar in effect to SUBI but the value is built into the instruction code itself. The immediate values in the source field are restricted to the range 1 to 8. This instruction is the fastest way to subtract a number between 1 to 8 from a destination operand.

Additional notes on SUB, SUBI, SUBQ: Most assemblers optimise your code automatically and if, for example, you write `sub #1,Dn` the assembler automatically translates it to `subq #1,Dn` thus reducing the size of the object code and saving a few clock cycles of execution time.

Mnemonic: TST – Test an Operand					
Purpose: Test an operand and set status flags accordingly					
Addressing Modes:	Source	Destination			
Data register direct		X			
Address register direct					
Address register indirect		X			
Postincrement register indirect		X			
Predecrement register indirect		X			
Register indirect with displacement		X			
Register indirect with index		X			
Absolute Short		X			
Absolute Long		X			
PC relative with displacement					
PC relative with index					
Immediate					
Flags affected:	X	N	Z	V	C
	-	Y	Y	0	0

To Get the Complete Story

to the instructions outlined in this appendix the 68000 includes many other specialist instructions including special commands called link and unlnk which can produce subroutines that use stack-based parameter passing, local variables etc. There are binary coded decimal (BCD) operations, a range of supervisor-mode-only commands (privileged instructions), specific 680x0 trap generating and handling instructions and so forth.

Consult the official 68000 literature for details. In addition

More notes on those important library functions...



The tables in this appendix provide library vector offset values and register usage details for some commonly used Amiga library functions (some of which have been used in the programs of this book). The term void in the following descriptions indicates that no return value is supplied. For full details of all libraries and their available functions you should consult the official Amiga documentation.

DOS functions LVO offset

<code>_LVOpen</code>	-30	<code>file(d0) = Open(name,mode) (d1,d2)</code>
<code>_LVOClose</code>	-36	<code>void() = Close(file) (d1)</code>
<code>_LVORead</code>	-42	<code>collected(d0) = Read(file,buffer,length) (d1,d2,d3)</code>
<code>_LVOWrite</code>	-48	<code>written(d0) = Write(file,buffer,length) (d1,d2,d3)</code>
<code>_LVOutput</code>	-60	<code>filehandle(d0) = Output()</code>
<code>_LVODelay</code>	-198	<code>void() = Delay(time) (d1)</code>

Exec functions

<code>_LVAllocMem</code>	-198	<code>block(d0) = AllocMem(size,type) (d0,d1)</code>
<code>_LVFreeMem</code>	-210	<code>void() = FreeMem(block,size) (a1,d0)</code>
<code>_LVCloseLibrary</code>	-414	<code>void() = CloseLibrary(library) (a1)</code>
<code>_LVOpenLibrary</code>	-552	<code>base(d0) = OpenLibrary(name, version) (a1,d0)</code>

Graphics functions

<code>_LVORectFill</code>	-306	<code>void() = RectFill(rastport,x1,y1,x2,y2) (a1,d0,d1,d2,d3)</code>
---------------------------	------	---

Intuition functions

<code>_LVOCloseScreen</code>	-66	<code>void() = CloseScreen(screen) (a0)</code>
<code>_LVOCloseWindow</code>	-72	<code>void() = CloseWindow(window) (a0)</code>
<code>_LVODisplayBeep</code>	-96	<code>void() = DisplayBeep(screen) (a0)</code>
<code>_LVODrawImage</code>	-114	<code>void() = DrawImage(rastport,image,x,y) (a0,a1,d0,d1)</code>
<code>_LVOpenScreen</code>	-198	<code>screen(d0) = OpenScreen(new_screen) (a0)</code>
<code>_LVOpenWindow</code>	-204	<code>window(d0) = OpenWindow(new_window) (a0)</code>

The following Intuition functions are available only from Release 2 (v36) onwards:

<code>_LVOLockPubScreen</code>	-510	<code>screen(d0) = LockPubScreen(name)(d0)</code>
<code>_LVUnlockPubScreen</code>	-516	<code>void() = UnlockPubScreen(name,[screen]) (a0,a1)</code>
<code>_LVOpenWindowTagList</code>	-606	<code>Window(d0) = OpenWindowTagList (nw,tags)(a0,a1)</code>

Usage notes

The system macro LINKLIB can be used to generate function call code in an easy-to-read, and conceptually tidy, fashion. An Intuition library OpenScreen() call for instance might take this form:

```
LINKLIB _LV0OpenScreen, _IntuitionBase
```

and the instructions generated would be:

```
move.l a6, -(sp)  
move.l _IntuitionBase, a6  
jsr _LV0OpenScreen(a6)  
move.l (sp)+, a6
```

To create an executable program the _LV0OpenScreen reference must at some stage be resolved, ie the real value for it must be found. This may be done either at link time (via the LVO values present in amiga.lib), by using an include file which contains the appropriate LVO value, or by the programmer inserting a suitable EQUate within their program. Since the numerical LVO values are available from the system documentation programmers are sometimes tempted to use the numerical equivalents directly, for instance, knowing that the _LV0OpenScreen reference is -198 a programmer could decide to code the above library opening fragment in one of these ways:

- 1) **LINKLIB _LV0OpenScreen, _IntuitionBase**
- 2) **move.l a6, -(sp)**
move.l _IntuitionBase, a6
jsr _LV0OpenScreen
move.l (sp)+, a6
- 3) **LINKLIB -198, _IntuitionBase**
- 4) **move.l a6, -(sp)**
move.l _IntuitionBase, a6
jsr -198(a6)
move.l (sp)+, a6

The preferred approach is to use the LINKLIB macro (or an equivalent macro) but if you do write the code manually you should always use the LVO name and NOT the numerical value. There are two reasons for this: Firstly, the LVO name approach provides more readable code. Secondly, if Commodore-Amiga do ever change the existing function arrangements in a library then providing you've used the LVO symbol-

ic names it would be possible to re-assemble/re-link your program with the new LVO data and it would work. This would not be possible if you had used numerical LVO equivalents in your code. In short you should avoid the style of the last two examples shown above!

Some extra help with the jargon...



active screen

On the Amiga this is the screen currently displaying the active window.

active window

The window currently receiving input from a user. On the Amiga only one window can be active at any one time.

address

A number which identifies a storage location in memory. addressing mode a term related to the way in which a microprocessor locates the operand that an instruction is to work on.

alert

A special red/black Amiga display used for emergency messages.

ALU

Arithmetic Logic Unit

angle brackets

These characters, < and >, are frequently used to identify command line parameters. For example... dir <filename> implies that 'filename' is a parameter which you, the user, should supply.

arithmetic logic unit

Part of a microprocessor which performs arithmetic and logical operations.

arguments

The values supplied when a function is used. These values are also often called parameters.

ASCII

American Standard Code for Information Interchange consists of a set of 96 displayable and 32 non displayed characters based on a seven bit code.

asynchronous

Some operation which is executed/performed without reference to an overall timing source. Asynchronous operations can therefore occur at irregular timing intervals.

background program

A program, task, or process, which is running somewhere in memory but not interacting directly via a terminal.

back-up

To make a duplicate of a program or data disk. Back-up copies are usually made for either safety or security purposes.

baud rate

A measurement of the rate of data transmission through a serial port. The baud rate divided by ten is a rough measure of the number of characters being transmitted per second.

BCD

Binary Coded Decimal.

binary

A number system using base 2 for its operations.

bit

An abbreviation of "binary digit".

bitmap

An array of bits which form a system's display memory. Modifying the data in the bitmap alters the picture on the display. The Amiga uses a bitmap display consisting of a number of two dimensional 'bitplanes'.

blanking interval

The period of time when a video beam is outside of the screen display area. It's a good time for a program to do things which might visually jar the display - the idea is to ensure that all of the necessary changes have been made before the video beam comes back into the visual area.

boot

To start up a computer system.

BPS

Bits per second.

branch

A type of processor instruction which causes control to pass from one section of a program to another. The branch is achieved by altering the contents of the processor's program control register which is the register which tells the processor from where it should get its next instruction. On the 68000 the term is reserved for instructions which use relative addressing.

buffer

An area of memory used to hold data temporarily whilst being collected or transmitted.

bug

A fault within a program that has not yet been found. Also see "Debug".

C

A high-level programming language: one of the best that has ever been developed.

call

To activate a program, function or procedure.

character string

A sequence of printable characters.

checksum

A number which is used to ensure that a block of data is correct and has not been inadvertently changed. Checksums are used to verify proper transmission and reception of data, to guard against deliberate alteration of sensitive file records etc.

clear

Change the value of a binary bit from 1 to zero.

CLI

Command line interface – precursor of the Shell.

clipping

Preventing the parts of an image which lie outside a specified drawing area from being displayed.

colour indirection

Powerful pixel colouring technique whereby the binary number formed by the appropriate image bits determines which colour register is used.

colour register

The Amiga has 32 hardware colour registers which means it has the ability to select from a palette of up to 4096 colours. (256 from 16 million - AGA?)

command file

An ordinary (usually ASCII) textfile containing executable system commands.

comment

A remark, social or otherwise, written within a program.

commenting out

In the assembly language world this term implies that part of the source code of a program has been eliminated not by removing it but by adding * or ; characters at the beginning of each line of a code section. This renders it inoperative because those lines are then treated as comments by the assembler. It is a trick frequently used by programmers during program development.

complement

“Binary complement”, the process of turning all 1s to 0s and all 0s to 1s.

concatenate

Join together. Strings, files etc, may be concatenated!

constant

Any value which does not change.

contiguous

Adjacent, lying next to each other etc. A contiguous block of memory is a block whose addresses are numerically adjacent and contain no gaps.

control character

A character that signifies the start or finish of some process.

Copper

An abbreviation for the Amiga's Co-processor chip.

Co-processor

The brilliant and powerful Amiga chip which handles much of the display work. This chip has its own instruction set which allows it to modify display characteristics without requiring 68000 processor intervention. Advanced Amiga programmers write their own Copper lists (Co-processor programs) for doing strange and wonderful graphics tricks.

CPU

Central Processing Unit.

crash

A term used when a computer program terminates unexpectedly or when the system hardware or software malfunctions. Usually reserved for serious problems that have no way of escape other than restarting the system.

CRT

Cathode Ray Tube

debug

To eliminate errors within a program.

debugger

A program designed to help programmers find errors – bugs – in their programs. Nowadays some highly sophisticated interactive debuggers are available which can link into the original source code as a program is executing.

decimal constant

A constant written as a base 10 number.

default value

A value which is supplied automatically if no other is given.

delimiting characters

Characters placed at the beginning or end of a character string.

destination file

A file being written to.

DMA

Direct Memory Access

direct memory access

A method of data transfer whereby intelligent hardware devices can read and write to memory without the main microprocessor being involved.

disable

To prevent something from being used.

display memory

The RAM area that contains data used to produce the screen image.

display mode

A particular type of screen display – low resolution, high resolution, non-interlaced etc.

editor

See text editor.

enable

To make something available for use.

EOF

End Of File

Exec

The Amiga's low level system software which controls tasks, task switching, interrupt scheduling, message passing, I/O and many other underlying system functions.

FIFO

First In First Out

file

A set of data items held on diskette, tape or other medium.

filename

A name given to a file for identification purposes.

fill

To colour or draw a pattern into an enclosed area.

flag

A single bit within a microprocessor register or memory location which has been chosen to represent some TRUE/FALSE, YES/NO, type situation.

floating point

A means of representing numbers in the binary equivalent of "scientific notation", ie by specifying an exponent and a mantissa.

glitch

A transient, normally unreproducible, problem usually associated with some hardware malfunction.

hard copy

The printed listing of some computer output as opposed to the output displayed on a VDU screen.

header file

Another term for a C include file.

hexadecimal

A base 16 numbering system using the digits 0-9 and the letters A-F.

hexadecimal constant

A base 16 constant which in assembler is written with the prefix \$ followed by the hexadecimal digits themselves.

IDCMP

Intuition Direct Communications Message Port.

Arguably the most important means of two-way, program-to-Intuition, communication.

I/O

input/output.

interrupt

An externally instigated request that, if accepted, causes the processor to save its current status and perform some required function. When the function has been completed the status of the processor is restored and control handed back to the interrupted program.

IntuiMessage

Messages created for applications programs by Intuition.

Intuition

Users regard Intuition as the Amiga's high level graphics interface, ie the overall Workbench orientated WIMP arrangement. Programmers take a much lower level view regarding Intuition as a mass of system routines and object definitions which can be used to simplify their programming tasks. The Intuition approach

allows programmers to easily create programs which use windows, gadgets, menus etc.

jump

A processor instruction which causes control to pass from one section of a program to another. The jump is achieved by altering the contents of the processor's program control register which is the register which tells the processor where it should get its next instruction from. The 68000 implements ordinary jumps, subroutine style jumps and branches – the last term is reserved for instructions which use relative addressing.

label

Rectangular shaped paper, often sticky, used for placing identification markings on objects.

label

An identification name used within the source code to refer to a particular section of coding.

long word

On the Amiga this implies a 32 bit binary number.

low-level language

A computer language whose primitive operations are closely related to the processor on which the language runs. Assembly languages are low-level.

memory map

A diagram showing the allocation of the various parts of memory chosen for a particular system or program.

message port

A fundamental software structure used by Exec's communication mechanism.

null-terminated string

A string of bytes which are terminated by a zero value.

octal

A base 8 numbering system.

operand

The value upon which an instruction or statement operates.

operating system

A collection of routines that perform the I/O and other hardware dependent chores that are needed for a computer to function.

parallel port

Hardware device which, on the Amiga, is used for transmitting data eight bits at a time. Mainly used for printer connection.

parameter

Any value which must be explicitly passed to a subroutine, function, procedure or program in order for it to be properly executed.

peripheral

Any external or remote device connected to a computer system, eg a printer.

pixel

The smallest addressable part of a screen display.

playfield

Another name for a screen background.

pointer

An address, record number or other indicator that specifies the next item of a data set taken in a specified logical order. With 68000 assembly language pointers are normally taken to mean addresses.

primitives

Another name for Amiga library functions and system routines.

refresh

To re-draw part (or all) of a graphics display.

render

Draw an image into a display area.

RAM

Random Access Memory.

ROM

Read Only Memory.

set

The act of turning a binary 0 into a binary 1 value.

Shell

An improved CLI interface which offers a number of useful new facilities including line editing and re-use of previously typed commands.

software

Any program or routine for a computer.

source code

The text version of a program, ie the program actually written in the first place.

source file

A file from which data is being read.

synchronous

Operations which are performed with reference to a controlling overall timing source.

syntax

The formal grammatical structure of a language.

text editor

A program that enables text to be written, manipulated, stored etc. Wordprocessor programs are sophisticated text editors.

title bar

An optional strip at the top of a window or screen which may contain either a name, some system gadgets or both.

tool

An Amiga Workbench name for an application program.

two's complement

A numerical representation in which positive numbers are represented as ordinary signed binary but negative numbers are represented by complementing the number and adding one.

VDU

Visual display unit

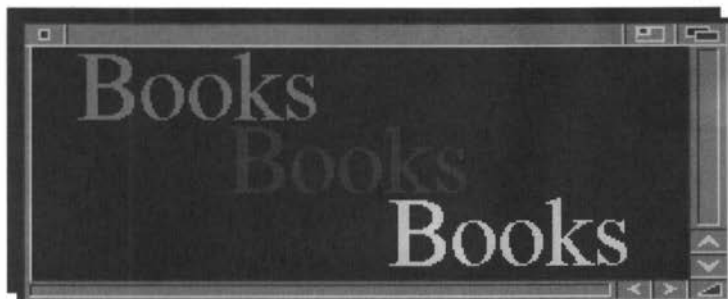
word

In the world of the 68000 programmer, and on the Amiga, a word is taken to mean a 16 bit binary number.

Workbench

The Amiga's inbuilt high-level interface applications program which allows users to interact with AmigaDOS, run applications programs etc, without getting involved with Shell commands.

Some useful sources of
further information...



The following books are a selection of those currently available on assembly language programming and the Amiga. They've been chosen because they have all, at some period in time, been found to be particularly useful. See also the details of the *Mastering Amiga System* and *Mastering Amiga Assembler* books provided immediately after this bibliography.

Title: Amiga ROM Kernel Reference Manual – Libraries
Author: Commodore-Amiga Inc.
Publisher: Addison-Wesley
ISBN: 0-201-56774-1
One of the major Amiga reference books.

Title: Amiga ROM Kernel Reference Manual – Devices
Author: Commodore-Amiga Inc.
Publisher: Addison-Wesley
ISBN: 0-201-56775-X
One of the major Amiga reference books.

Title: Amiga ROM Kernel Reference Manual –
Includes & Autodocs

Author: Commodore-Amiga Inc.

Publisher: Addison-Wesley

ISBN: 0-201-56773-3

One of the major Amiga reference books.

Title: Amiga Hardware Reference Manual

Author: Commodore-Amiga Inc.

Publisher: Addison-Wesley

ISBN: 0-201-56776-8

One of the major Amiga reference books..

Title: Amiga User Interface Style Guide

Author: Commodore-Amiga Inc.

Publisher: Addison-Wesley

ISBN: 0-201-57757-7

One of the major Amiga reference books.

Title: The AmigaDOS Manual

Author: Commodore-Amiga Inc.

Publisher: Bantam Books

ISBN: 0-553-35403-5

Now in its third edition this is the most comprehensive guide to the internal workings of AmigaDOS that exists but parts of it are technically heavy going.

Title: The Kickstart Guide to the Amiga

Author: Dave Parkinson and Mike Boley.

Publisher: Ariadne Software Ltd.

This book has been about for quite a few years now so it is a little out of date in places. Nevertheless it contains a lot of useful information and is still worth reading.

Title: Computers – From Logic to Architecture

Author: R. D. Dowsing and F. W. D Woodhams

Publisher: Van Nostrand Reinhold

ISBN: 0-278-00093-2

Contains good general introductions to hardware issues (processors, memory chips and so on) including some 68000 material.

Amiga Insider Guide

Title: Dr Dobb's Toolbook of 68000 Programming
Authors: Editors of the Dr Dobbs Journal
Publisher: Prentice Hall
ISBN: 0-13-216557-0

A goldmine for ideas once you are fairly 68000 proficient, but does not contain any Amiga specific material.

Title: 68000 Assembly Language Programming
Authors: Kane, Hawkins and Leventhal
Publisher: Osborne/McGraw-Hill
ISBN: 0-931988-62-4

A very good general Motorola 68000 book with very detailed accounts of the instruction set.

We've taken you through the door of 68000 programming and, as a result, you'll be getting an idea of where your interests lie.

Now comes the opportunity to take your computing a serious step further with the Bruce Smith Books range of titles.



Bruce Smith Books is dedicated to producing quality Amiga publications which are both comprehensive and easy to read. Our Amiga titles are written by some of the best known names in the marvellous world of Amiga computing. If you have found that your *Insider Guide* has proved informative and want to delve deeper into your Amiga then why not try one of our highly rated *Mastering Amiga* guides? In other words – if you enjoyed getting insider your Amiga, now is the time to master it!

Below you'll find details of all the *Mastering Amiga* and *Insider Guide* range books that are currently available or due for publication soon.

Compatibility

We endeavour to ensure that all *Mastering Amiga* books are fully compatible with all Amiga models and all releases of AmigaDOS and Workbench. The *Mastering AmigaDOS* books are constantly updated to reflect Commodore's evolving Amiga operating system so you can be

sure that these bibles of Amiga computing will keep up to date with you and your computer. Please check the list of titles currently and soon to be available below for full compatibility.

<i>Book</i>	<i>A500</i>	<i>A500+</i>	<i>A600</i>	<i>A1200</i>	<i>A2000</i>	<i>A3000</i>	<i>A4000</i>
Mastering AmigaDOS2 Vol. 1	Y	Y	Y‡	N	Y	Y*	N
Mastering AmigaDOS2 Vol. 2	Y	Y	Y‡	N	Y	Y*	N
Mastering AmigaDOS3 Vol. 1	N	N	N	Y	N	Y#	Y
Mastering AmigaDOS3 Vol. 2	N	N	N	Y	N	Y#	Y
Mastering Amiga Workbench2	N	Y	Y	N	Y	Y*	N
Mastering Amiga Beginners	Y†	Y	Y	Y	Y†	Y	Y
Amiga Gamer's Guide	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga AMOS	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga C	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga Printers	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga System	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga Assembler	Y	Y	Y	Y	Y	Y	Y
Mastering Amiga ARexx	N	Y	Y	Y	Y	Y	Y
Amiga A600 Insider Guide	N	N	Y	N	N	N	N
Amiga A1200 Insider Guide	N	N	N	Y	N	N	N
Amiga A1200 Insider Guide 2	N	N	N	Y	N	N	N
Workbench 3 A-Z Insider Guide	N	N	N	Y	N	Y#	Y
Y† State if you have AmigaDOS 1.3	Y* Earlier versions with AmigaDOS2						
Y‡ 80% compatible with 2.1 version	Y# Latest versions with AmigaDOS3						

Brief details of these guides along with review segments are given below. If you would like a free copy of our catalogue and to be placed on our mailing list then phone or write to the address below.

You can order a book simply by writing or using the simple tear our form to be found towards the end of this book.

Our mailing list is used exclusively to inform readers of forthcoming Bruce Smith Books publications along with special introductory offers which normally take the form of a free software disk when ordering the publication direct from us.

Bruce Smith Books, PO Box 382, St. Albans, Herts, AL2 3JD
Telephone: (0923) 894355 – Fax: (0923) 894366

Note that we offer a 24-hour telephone answer system so that you can place your order direct by 'phone at a time to suit yourself. When ordering by 'phone please:

- Speak clearly and slowly
- Leave your full name and full address
- Leave a day-time contact phone number
- Give your credit card number and expiry date
- Spell out any unusual names

Note that we do not charge for P&P in the UK and we endeavour to dispatch all books within 24-hours.

Buying at your Bookshop

All our books can be obtained via your local bookshops – this includes WH Smiths which will be keeping a stock of some of our titles – just enquire at their counter. If you wish to order via your local High Street bookshop you will need to supply the book name, author, publisher, price and ISBN number – these are all summarised at the very end of this appendix.

Overseas Orders

Please add £3 per book (Europe) or £6 per book (outside Europe) to cover postage and packing. Pay by sterling cheque or by Access, Visa or Mastercard. Post, Fax or Phone your order to us.

Dealer Enquiries

Our distributor is Computer Bookshops Ltd who keep a good stock of all our titles. Call their Customer Services Department for best terms on 021-706 1188.

Summary Book Details

A600 Insider Guide by Bruce Smith

ISBN: 1-873308-14-0, Price £14.95, 256 pages.

A perfect companion for all A600 and A600HD users. This book provides you with a unique insight into the use of Workbench and AmigaDOS on all versions of the Amiga A600.

Assuming no prior knowledge it shows you how to get the very best from your machine in a friendly manner and using its unique *Insider Guide* illustrations (see A1200 description below).

A1200 Insider Guide by Bruce Smith

ISBN: 1-873308-15-9, Price £14.95, 256 pages.

The World's best selling Amiga A1200 book from the world's number one selling Amiga author! Assuming no prior knowledge, it shows you how to get the very best from your A1200 in a friendly manner and using its unique *Insider Guide* illustrations. Configuring your system for printer, keyboard, Workbench colours, use of Commodities and much, much more has made this the best-selling book for the A1200.

As well as easy to read explanations of how to get to grips with the Amiga, the book features 55 of the unique Insider Guides, each of which displays graphically a set of step by step instructions. Each Insider Guide concentrates on a especially important or common task which the user has to carry out on the Amiga. By following an Insider Guide the user learns how to control the Amiga by example. Beginners to the A1200 will particularly appreciate this approach to a complex computer.

The disks which come with the A1200 contain a wealth of utilities and resources which allow you to configure the computer for your own way of working. The step by step tutorials take you through using these point by point, anticipating any problems as they go. There are also fully fledged programs such as MultiView and ED which can seem impenetrable for the new user but which become clear when observed in use over the shoulder of author Bruce Smith.

Great new features such as the colour wheel, Intellifonts, using MSDOS disks with CrossDos and configuring sound are dealt with in detail. A useful appendix acts as a file locator so that any of the many files on the Amiga disks can be quickly found.

Amiga A1200 Next Steps by Peter Fitzpatrick

ISBN: 1-873308-24-8, Price £14.95, 256 pages. Available Nov. 93.

For those who have mastered the very basics of the A1200 this book is the ideal companion to our *Amiga A1200 Insider Guide*. Leaving the basics of the Workbench and AmigaDOS behind this book takes you the next step and shows you how to get the very most out of your A1200, using both the software supplied and other material readily available.

For example, learn how to use MultiView to write your own adventure game and edit a picture! Create your own fully recoverable Ram disk, get better results when you print out, recover deleted files. We even show you how to add your own hard disk and copy software onto it! This is only the tip of the iceberg. *Amiga A1200 Next Steps* is worth its weight in gold!

Workbench 3 A to Z Insider Guide by Bruce Smith

ISBN: 1-873308-28-0, Price £14.95, pages TBA. Available Dec. 93.

From the world's number 1 selling Amiga book author comes this indispensable guide which covers every aspect of the Amiga Workbench version 3. Complete with illustrations, it provides comprehensive coverage of every Workbench menu option and icon across every disk – and more.

An indispensable guide and essential reference for every Workbench 3 owner!

Mastering Amiga Beginners by Bruce Smith and Mark Webb

ISBN: 1-873308-17-5, Price £19.95, 320 pages. FREE Games disk.

Mastering Amiga Beginners is the book for the growing number of novice computer users who turn to the Amiga as the natural computer for home entertainment and self-education.

The authors have built up a wide experience of beginners' requirements and the problems they encounter and now this vast knowledge of the subject has been distilled into 320 pages of sensible advice and exciting ideas for using the Amiga.

Mastering AmigaDOS 2 Volume One – Revised Edition by Bruce Smith and Mark Smiddy

ISBN: 1-873308-10-8, Price £21.95, 416 pages. FREE Utilities disk.

Volume One of the *Mastering AmigaDOS 2* dual volume set is a complete tutorial to AmigaDOS, designed to help the beginner become the expert. From formatting a disk to multi-user operation, over 400 pages spans every aspect of the Amiga's operation. The book is packed with DOS one-liners and scripts.

Mastering AmigaDOS 2 Volume Two – Revised Edition by Bruce Smith and Mark Smiddy

ISBN: 1-873308-09-4, Price £19.95, 368 pages.

Mastering Amiga DOS 2, Volume Two is a complete A to Z reference to DOS commands and the current version has full details up to version 2.04. The action of each command is explained and examples to try

are provided. Chapters on AmigaDOS error codes, viruses, the Interchange File Format (IFF) and the Mountlist complete this valuable guide.

Mastering AmigaDOS 3 Volume One – Tutorial

by Bruce Smith and Mark Smiddy – Available Dec. 93.

ISBN: 1-873308-20-5, Price £21.95, pages TBA. FREE Utilities disk.

Volume One of the *Mastering AmigaDOS 3* dual volume set is a complete tutorial to AmigaDOS 2.0, 2.04 and 3. Designed to help the beginner become the expert it follows the highly successful format of the *Mastering AmigaDOS 2* series. From formatting a disk to multi-user operation, over 400 pages spans every aspect of the Amiga's operation. The book is packed with DOS one-liners and scripts.

Mastering AmigaDOS 3 Volume Two – Reference

by Bruce Smith and Mark Smiddy

ISBN: 1-873308-18-3, Price £21.95, 416 pages.

Following on from the best selling *Mastering AmigaDOS 2* volumes, *Mastering Amiga DOS 3, Volume Two* is a complete A to Z reference to DOS commands covering versions 2.0, 2.04 and 3. The action of each command is explained and examples to try are provided. Chapters on AmigaDOS error codes, viruses, the Interchange File Format (IFF), the Mountlist and the new hypertext system, AmigaGuide, complete this valuable guide.

Mastering Amiga System by Paul Overaa

ISBN: 1-873308-06-X, Price £29.95, 398 pages. FREE disk.

Serious Amiga programmers need to use the Amiga's operating system to write legal, portable and efficient programs. But it's not easy! Paul Overaa shares his experience in this introduction to system programming in the C language. The author keeps it specific and presents skeleton programs which are fully documented so that they can be followed by the newcomer to Amiga programming. The larger programs are fully-fledged examples which can serve as templates for the reader's own ideas as confidence is gained.

Mastering Amiga Workbench 2 by Bruce Smith

ISBN: 1-873308-08-6, Price £19.95, 320 pages.

Author Bruce Smith explains everything you will want to know about the Workbench version 2.x using screen illustrations throughout for ease of reference. Geared towards all types of users, it starts from the first steps and explains the philosophy of the Workbench and how it

ties in with your Amiga. Moving on to describe the best way to perform basic tasks such as disk copying, file transfer and how to customise your own Workbench disks, it moves on to work its way through each of the menu options with full descriptions of their use, providing many hints, tips and tricks on the way.

Mastering Amiga AMOS by Phil South

ISBN: 1-873308-13-2, Price £19.95, 320 pages.

AMOS has very quickly developed into one of the most exciting and accessible programming languages on the Amiga. Its easy to use interface and familiar BASIC structure are augmented by powerful libraries for games and graphics programming. *Mastering Amiga AMOS* is ideal for anyone investing in AMOS, EasyAMOS or AMOS Professional. Full of hints, tips and shortcuts to effective and spectacular AMOS programming, this book also contains many useful routines and program design ideas.

Mastering Amiga Assembler by Paul Overaa

ISBN: 1-873308-11-6, Price £24.95, 416 pages. FREE disk.

The *big brother* to the *Amiga Assembler Insider Guide*, this book explains the use of assembly language to write efficient code within the unique environment of the Amiga, doing so without duplicating standard 68000 material in over 400 pages. Instruction is achieved by short code examples amidst discussion of the issues involved in using machine code for various purposes. Subjects covered include cooperation with the System software, custom chips and the C language. All the popular Amiga assemblers are supported by the many code examples in this book.

Mastering Amiga C by Paul Overaa

ISBN: 1-873308-04-6, Price £19.95, 320 pages.

FREE Programs Disk and NorthC Public Domain compiler.

C is one of the most powerful programming languages ever created with much of the Amiga's operating system written using C. The introductory text assumes no prior knowledge of C and covers all of the major compilers, including the charityware NorthC compiler supplied with this book when ordered direct from BSB. It is ideal for anyone using their Amiga to catch up on computer studies!

Mastering Amiga ARexx by Paul Overaa

ISBN: 1-873308-13-2, Price £21.95, 336 pages. FREE disk.

Now a standard part of Commodore's software strategy and readily available to Workbench 2 and 3 users, ARexx has been much admired by the programming community and is now available to all as a third party product. This book is an ideal companion to the ARexx documentation, explaining ARexx's main features, how it controls other programs, its built-in functions and support libraries, methods for creating well structured ARexx programs and much, much more.

Mastering Amiga Printers by Robin Burton

ISBN: 1-873308-05-1, Price £19.95, 336 pages. FREE Programs disk

After reading *Mastering Amiga Printers*, any Amiga owner will be able to choose effectively the ideal printer for his or her requirements. The Amiga's own printer control software is pulled apart and explained from all points of view, from the Workbench to the operating system routines. Individual printer drivers are assessed and screen-dumping techniques explained.

Amiga Gamer's Guide by Dan Slingsby

ISBN: 1-873308-16-7, Price £14.95, 368 pages.

Everyone loves games and Amiga games are growing in sophistication, always setting new playing challenges while introducing ever more gasp-producing graphics and sound effects. Even the techies at Bruce Smith Books are, it seems, not immune to the games phenomenon. This latest book for the discerning Amiga owner, is a highly illustrated guide to your favourite Amiga games, including classics like *Shadow of the Beast* and recent top ten hits like *Putty*, *Formula One Grand Prix*, *Streetfighter 2* and *Indiana Jones*.

From sports sims to arcade adventures, *Amiga Gamer's Guide* gives you the hints and tips, hidden screens and puzzle solutions which you are looking for. Completed by a massive A to Z of tips and tricks for over 300 games, *Amiga Gamer's Guide* is the most masterful of games guides yet published.

Written by *CU Amiga* editor Dan Slingsby, *Amiga Gamer's Guide* contains a wealth of background information to the most popular Amiga games. The graphically appealing layout with hundreds of pictures used to illustrate the games and their storylines, makes this one of the most attractive Amiga books to be found on the bookshelves.

Note: Disks where indicated are supplied free only when ordered direct from Bruce Smith Books. E&OE.

Disk Order Form

Please rush me a copy of the *Amiga Assembler Insider Guide* disk

I enclose a Cheque/Postal Order for £1.50p made payable to
Bruce Smith Books Ltd.

Name.....

Address.....

.....Post Code

Contact phone number.

Send your order to:

**Assembler Insider Guide Disk,
Bruce Smith Books Ltd,
PO Box 382, St. Albans, Herts, AL2 3JD.**

Please note that unless otherwise requested we will add you to our mailing list. This mailing list is currently only used to mail out to our readers details of new and forthcoming books. This includes our catalogue *Mastering Amiga News*.

Please take the time to answer the following questions:

How did you find out about *Amiga Assembler Insider Guide*?

Where did you purchase your copy?

What other titles would you like to see in the *Insider Guide* range of books?



Book Order Form

Please rush me the following:

Amiga A1200 Insider Guide @ £14.95	£
Amiga A600 Insider Guide @ £14.95	£
Amiga A1200 Next Steps Insider Guide @ £14.95	£
Amiga Workbench 3 A to Z Insider Guide @ £14.95	£
Mastering Amiga Beginners @ £19.95 with Games Disk	£
Mastering AmigaDOS 3 Vol. One Tutorial @ £21.95 with Disk	£
Mastering AmigaDOS 3 Vol. Two Reference @ £21.95	£
Mastering AmigaDOS2 Vol. One @ £21.95 with Disk	£
Mastering AmigaDOS2 Vol. Two Revised Edition @ £19.95	£
Mastering Amiga C @ £19.95 with Scripts & PD NorthC Disk	£
Mastering Amiga Printers @ £19.95 with PD Disk	£
Mastering Amiga System @ £29.95 with Programs Disk	£
Mastering Amiga Assembler @ £24.95 with Programs Disk	£
Mastering Amiga AMOS @ £19.95	£
Mastering Amiga ARexx @ £21.95 with Programs Disk	£
Mastering Amiga Workbench 2 @ £19.95	£
Amiga Gamer's Guide @ £14.95	£
NEW: A1200 Workbench Tutorial Video @ £14.99 (inc VAT)	£

Postage (International Orders Only): £

Total: £

I enclose a Cheque/Postal Order* for £ p.

I wish to pay by Access/Visa/Mastercard* Expiry Date:

Card number:

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Name

Address

.....

Post Code Contact Phone No.....

Signed

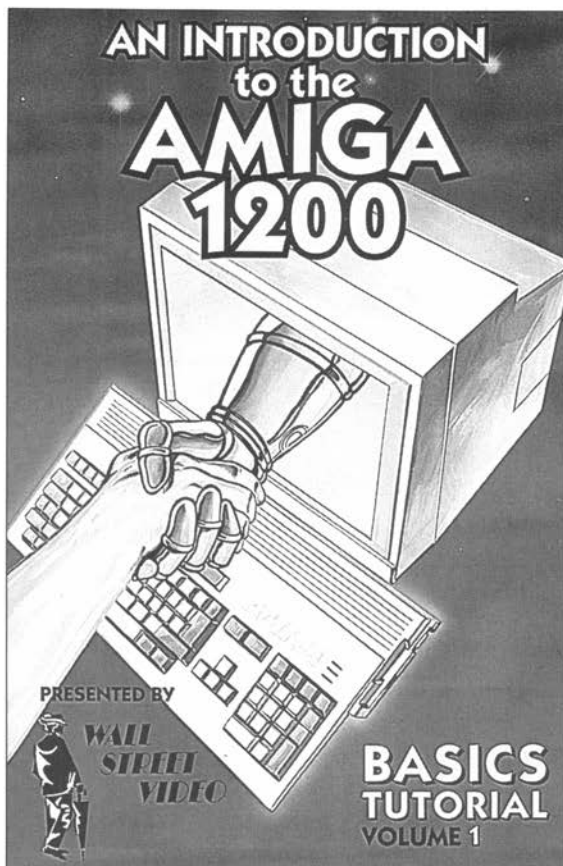
E&OE

Please send your cheques payable to Bruce Smith Books Ltd to:

Bruce Smith Books Ltd, FREEPOST 242, PO Box 382, St. Albans, Herts, AL2 3BR
Telephone: (0923) 894355 - Fax: (0923) 894366







**Attention all Amiga
A1200 Owners!**

NEW from Bruce Smith Books in association with Wall Street Video – Australia's leading Amiga training company – the perfect video introduction to using your Amiga A1200 and a perfect companion for the world's top selling A1200 book, Bruce Smith's classic *Amiga A1200 Insider Guide*. This one hour video provides a basic tutorial on how to set up and run your Amiga A1200 by using great animations and split screens to increase your understanding of the concepts being explained. Re-examine those tricky *grey areas* by instantly rewinding the video!

Applicable to both hard and floppy disk users the *Amiga A1200 Video* may also be used to understand the Amiga A4000 and at £14.99 represents outstanding value. Available from all good stores or direct from BSB. Simply phone (0923) 894355 to place your credit card order, today!

“The Mastering Amiga series provides top quality guidance for Amiga users.”

...but don't just take our word for it!

“If you're a beginner or a newcomer to Amigas, these two books provide an excellent way of finding your way around your new machine”

Richard Baguley, Amiga Format on the
A600 and A1200 Insider Guide series.
AF GOLD AWARD – 90%

“This book has been written with the absolute novice in mind. It doesn't patronise, yet neither does it baffle with jargon and slang”

Chris Lee, CU Amiga Review on Mastering Amiga C.

“I have to say that the best hands-on tutorial that I've seen is Mastering AmigaDOS 2 Volume One.”

Pat McDonald, Amiga Format on Mastering AmigaDOS Vol. 1.

“The definitive book on the subject, don't leave your Workbench without it!”

Neil Jackson, Amiga Format on Mastering AmigaDOS Vol. 2.

“...it's well worth buying a decent book on the subject – I personally recommend you get Mastering Amiga Printers.”

Jason Holborn, Amiga Format on Mastering Amiga Printers.

“The latest in the excellent range of specialist Amiga Books... covers every aspect of the complex Amiga system”

Damien Noonan, Amiga Format on Mastering Amiga System.

68000 assembly language	15, 17
68000 processor	26, 31

A

A68k pd assembler	23, 24, 79
AbsExecBase	71
ADD instruction	62, 210
ADDI add immediate instruction	64, 211
address registers	26, 27, 60
addressing – absolute	34, 62, 191
addressing – immediate	34, 35, 59, 64, 191
addressing – indirect	34, 192
addressing – inherent	34, 191
addressing – PC relative with displacement	193
addressing – PC relative with index and displacement	194
addressing – register	34, 191
addressing – register indirect	34, 192
addressing – register indirect with displacement	146, 192
addressing – register indirect with index and displacement	192, 193
addressing – register indirect with postincrement	99, 192
addressing – register indirect with predecrement	99, 192
addressing modes	34, 191
amiga.lib library	68
AmigaDOS	69, 105
AND	51
ANDI logical AND immediate instruction	203
arithmetic instructions	35, 62, 214, 210, 215
ASCII	19, 21, 43
ASL instruction	207, 208
assembler directives (<i>also see pseudo-ops</i>)	42
assemblers	17, 39

B

Bcc conditional branch instructions	35, 199
BEQ instruction	35, 75, 163
binary numbers	43
bit manipulation instructions	36, 209
Blink	20, 24, 80
BRA unconditional branch instruction	163, 200
branches and jumps	28
BSR branch to subroutine instruction	35, 37
BTST instruction	209
byte	27

C

CALLSYS macro	101, 109, 165
carry flag	28
chip memory	147
CloseLibrary()	69, 71, 74
CloseWindow()	133
CLR clear instruction	212
CMP compare instruction	213
CMPI compare immediate instruction	214
comments	39, 40
complementing	61
conditional assembly	44
conditional branching	35, 75, 163
CPU	15
crashes	22, 87

D

data movement instructions	34, 54
data registers	26, 27, 55
DC.x directives	43, 56
debuggers	20, 54, 90
decimal numbers	43
Delay()	130, 133
Deluxe Paint	145
Devpac	23, 80, 89
DisplayBeep()	74
DOS functions	106, 107, 110, 130, 133
DOS library	105, 130
DOSBase	107
DrawGrid() subroutine	159
DrawImage()	144
DS.x directives	43, 56
duplicate label errors	41

E

edit<->assemble cycle	20
editors	20, 79, 89
effective address	190
END pseudo-op statement	81, 134, 147
END statement is missing error	85
EORI exclusive OR immediate instruction	204
EQU directive	42, 108, 128
errors	41, 85, 95, 159

Exec68, 71
Exec functions.....68, 70, 71, 74, 219
Exec library68
execution cycle30

F

fetch cycle.....30
flags.....26, 28, 29, 56
flags – MOVEA effect on.....59
flow control35, 199
function – CloseLibrary()69, 71, 74
function – CloseWindow()133
function – Delay()131, 133
function – DisplayBeep()74
function – DrawImage()144
function – Input().....106
function – LockPubScreen().....131
function – OpenLibrary()68, 70
function – OpenWindow()124
function – OpenWindowTagList()125, 132
function – OpenWindowTags()126
function – Output()106
function – UnlockPubScreen()132
function – Write()107, 110
functions.....68, 217
functions – DOS106, 107, 110, 130, 133, 217
functions – Exec68, 70, 71, 74, 218
functions – Intuition74, 125, 131, 132, 144, 218
functions – LVO offsets217

G

guru.....22

H

header files103, 117
hexadecimal numbers.....43
high-level languages.....17

I

IFF pictures – using in programs145
I/O handles105, 187
images.....143
immediate addressing.....34, 35, 59, 64, 191
INCLUDE directive103

include files	103, 117, 119, 138
indirect addressing	34, 146, 192
inherent addressing	34, 191
Input().....	106
input/output handles	105, 187
instruction set	17, 30, 34
instructions – ADD	62, 210
instructions – ADDI	64, 211
instructions – ANDI	203
instructions – arithmetic	35, 62, 210
instructions – ASL	207, 208
instructions – Bcc	35, 199
instructions – BEQ.....	35, 75, 163
instructions – bit manipulation	36, 209
instructions – BRA	35, 163, 200
instructions – BRA/JMP	35, 202
instructions – BSR	35, 37
instructions – BTST	209
instructions – CLR	212
instructions – CMP	213
instructions – CMPI	214
instructions – data movement.....	54, 194
instructions – EORI	204
instructions – JMP	202
instructions – JSR	37, 201
instructions – LEA	65, 194
instructions – MOVE	34, 54, 195
instructions – MOVEA.....	59, 196
instructions – MOVEM.....	158, 197, 198
instructions – MOVEQ	198
instructions – NOT.....	61, 205
instructions – ORI.....	206
instructions – quick	65, 198, 211
instructions – RTS	37, 75, 202
instructions – TST.....	216
Intuition.....	23, 74, 123
Intuition functions.....	74, 131, 132, 144

J

JMP jump instruction.....	35, 202
JSR jump to subroutine instruction	35, 201

L

label conventions41
labels.....39, 40, 90
languages – 68000 assembler16
languages – low-level benefits17
LEA load effective address instruction.....65, 194
libraries.....67, 72, 185
libraries – amiga.lib68
libraries – DOS.....105
libraries – example use76, 128
libraries – Exec.....68, 71
libraries – linker20, 68
libraries – run-time68
library opening.....68
library vector offset (LVO).....see LVO function offsets
linker libraries.....20, 68
linking.....20, 79, 86
LINKLIB macro98, 109, 120, 138
local labels41
location counter40, 108
locking function130
LockPubScreen().....131
long word.....27
long words – storage in memory.....58
loops162
LVO function offsets.....72, 73, 91, 153, 217

M

macros44, 97, 113, 126
memory conservation63
messages.....130, 186
microprocessor15, 25, 30
mnemonics17
MOVE data movement instructions34, 195
MOVEA effect on flags59
MOVEA move to address register instruction.....196
MOVEM instruction158, 197, 198
MOVEQ move quick instruction.....198

N

negative flag28
NOT instruction61, 205
NULL71, 75, 126

number conversion	47, 49
number systems	43, 47
numbers – binary	43, 48
numbers – decimal	43, 48
numbers – hexadecimal	43, 48

O

object code	20, 68
op-code	39, 190
OpenLibrary()	68, 70
OpenWindow()	124
OpenWindowTagList()	125, 132
OpenWindowTags()	126
operand format errors	86
operands	39, 43
ORI logical OR immediate instruction	206
O/S Release 2 onwards	123
Output()	106
overflow flag	28

P

parameters	110, 159
PC (program counter)	26, 28
Power Windows	145
program comments	40
program counter (PC)	26, 28
pseudo-ops	39, 42

Q

quick instructions	65, 198, 211
--------------------------	--------------

R

RAM	16
register addressing	34
register indirect addressing	34, 192
registers – address	26, 27, 60
registers – data	55
ROM	16
ROM Kernel reference manuals	145
RTS return from subroutine instruction	37, 75, 202
run-time libraries	68

S

scratch registers	112, 159, 163
-------------------------	---------------

section conventions	147
Shell/CLI programs	105
shift/rotate instructions	36, 207
sign extension	59, 190
source code	20, 82
stack operations	27, 158
stack pointer	26, 27, 99
start-up code	20, 44, 186
status register	26, 28, 56
stdin handle	105
stdout handle	105
storage allocation directives	42
storage in memory – long words	58
storage in memory – words	57
strings – null terminated	74
subroutines	37, 158, 159
supervisor mode	27
SysBase	71
system byte	28

T

tag identities	127, 137
TagItem structure	126
tag lists	123
text messages	108
truth tables	51
TST instruction	216

U

undefined symbol errors	86
underscore conventions	73
UnlockPubScreen()	132
user byte	28, 56
user mode	27

W

word	27
word alignment	28
words – <i>storage in memory</i>	57
Write()	107, 110

Z

zero flag	28, 29, 107
-----------------	-------------

Amiga assembler

Amiga assembler is the definitive beginners guide to learning machine code programming on all Amigas, including the A1200, A4000, A3000 and A600.

Each program example in the book can be assembled and run in under one minute, so the novice programmer can achieve practical results quickly, learning essential techniques along the way.

The unique graphical Insider Guides take you step by step through all the important operations so that you will be able to confidently type in and edit source code, assemble it, debug it and run it.

Learn how to create Workbench windows and menus, how to sense mouse actions, to display graphics, work with coprocessors and much much more.

This book is compatible with all the main assemblers on the market and no extra libraries are needed to run the examples. A support disk is available which contains the A68k assembler and all the examples from the book. Learning assembler on the Amiga has never been easier.

ISBN 1-873308-27-2



9 781873 308271

£14.95

BSB

Bruce Smith Books

Publishers of the
World's Best Selling
Amiga Books



**This was brought to you
from the archives of**

<http://retro-commodore.eu>