

MANIACOMAC'S



PPC CRACKING BIBLE



ALL THE BEST MAC CRACKING TEXTS IN ONE
ALL THE KNOWLEDGE YOU NEED TO FASTLY AND EFFECTIVELY CRACK RECENT MAC PPC SOFTWARES

VERSION

1.0.2

ASSEMBLED BY MANIACOMAC



FOREWORDS

Welcome Everyone !

So you want to learn how to crack Mac softwares ?

Well, to be true, this book is your best bet and the place to start looking at. This book is and will be the Mac Cracking Heaven, and also the start point of learning.

I guess this book will be the ultimate Mac cracking guide for the newbies , beginners, next generation Mac crackers and even a reference guide to more advanced Mac Crackers. I really hope this work on the subject will help some, this is my own contribution to the Community.

It covers about everything from Mac PPC Assembly language learning to the most recent softwares cracking methods use by the most talented crackers. I think I gathered all the best Mac cracking texts in one, to created what I hope will be a must have for the Mac Cracking Community members. It is organized in a logical learning progression, in order to help those nembies who really don't know where to start.

This is version 1.0.2 of the book and a lot of improvements are to come soon. A lot of new texts will be added in the futur releases ... and maybe a french version of this book will be released If I'm not too lazy ! All new releases of this book will be uploaded only at some specific places, and the version number and date of release will be mentioned each time.

You can also send me some texts you want or think should be added to this book, to make it more complete, the most complete ever. Texts about OS X cracking would be really appreciate, this is a missing part of the Mac Cracking Community and I hope we will be in a position to fix this issue soon.

Well I don't know if you can freely distribute this book but I think you should not cause it contains texts from other authors. So let's wait and keep it for your personal use.

Have fun with this book and good learning ! I am really glad to share this book with all of you. So STK and let's Share The Knowledge ! ... and Krack Different !

Oh I almost forgot ...

Many many thanks to ProZaq, The Vassal, Shepherd, ORC, Fluffy, Pablo, iÇ@açk, Orygun, Halo Ghost, Halo Driver, TheShark, Anarchie, Absym, Buck, Xena, Charasi, Cendryom, Akuma, Coldhit, fintler, Mazzie, Postscript, Snapcase, Jaffa, Jakko, Phreak, Mopar, Ahton, Cyborg, Dot, Byte, Observer, Mouto, Svede, Stardust, FireSt0rm, Thoughts, Corsec, Pyrus Malus, KrackerJAG, etc ... just to mention a few, and also many other nicknames for their help, support and more ! Also many thanks to many servers admins, sub-admins and sysop I cannot mention here, or rather I do not dare to mention. ;-) Let me know if you helped my in some ways and then I forgot to mention you here, and you would like to be mentioned in the futur releases.

Maniacomac ® 2004

For any comments or suggestions about this book

maniacomac@hotmail.com

Maniacomac is also known as MoM

PowerPC Assembly Language Beginners Guide

Chapter 1

What is Machine Code ? And what is Assembly Language?

Machine code is the code that your computer actually runs. Processors don't understand English, Swedish, Basic, C or anything else recognizable to humans (if C is?). Only ones and noughts (binary, meaning "two possible states") - machine code is ones and noughts. Luckily, nobody has to program in machine code any more. We use assemblers and compilers, but we do need to know what machine code is.

When your computer is running, it continuously gets instructions from memory. These instructions tell the processor what to do now. On a PowerPC based machine, the instructions are made up of thirty two binary digits or "bits". A bit can be in one of two states; either a 1 or a 0. 32 bits together are called a word.

The pattern of the word will tell the processor how to execute the instruction. A certain word may add two numbers together, whilst another word could make the processor store a number back in memory, and so on. A program consists of lots of instructions strung together, and of course, some data for the instructions to operate on.

Programs can be written in many kinds of different languages, some common ones are BASIC (Beginners All purpose Symbolic Instruction Code), C, and Pascal. These are termed high level languages. High level languages have to translate the terms used by the programmers into machine code. This is fine, except that the high level languages only know a certain way of doing things, and sometimes have to string together significantly long sets of ones and noughts to get the desired end result. What this means in practice is that any compiled language, will only be as good as the compiler.

Assembly language on the other hand is a low level language. Each assembly language mnemonic translates directly into a machine code instruction. The difference between a high level language such as C and assembly language is that an intelligent being is doing the compiling. Because a human is writing the code and not a machine, the code can be written in the best way to achieve maximum speed, and use the minimum of memory. For this reason assembly language in the hands of a competent programmer will always be a lot faster than any compiled language.

Just a little note here, and this has nothing to do with this guide, but...
The acronym TWAIN stands for Technology Without An Interesting Name - I just thought you might not know that and find it "Interesting". Oh well, onwards...

What is a Mnemonic?

The dictionary defines it as "something to help the memory" - in this case its a word that represents a machine code instruction.

The PowerPC family of processors understands about 60 basic instructions, which means the PowerPC assembly language programmer has 60 different mnemonics to remember. Compared to a

high level language, which may have as many as 800 different instructions (BASIC for example), this is a small number to memorize. In practice not all these instructions will be used, and you may find yourself using as few as 10 instructions regularly, so its not difficult to learn.

What does a Mnemonic look like?

The PowerPC has an instruction to add two numbers together and store the result away. In ones and noughts it may look like 0011110010000100001101000000000000 (don't quote me on that!), which is a complete and utter mouthful, so we use the mnemonic "add". We get an assembler to translate the "add" to the required machine code. Its as simple as that.

Some other useful mnemonics include:

LI - Load immediate - loads a number into a processor register.

STW - Store a word from a processor register into memory

LWZ - Load a word from memory into a processor register.

Thus, assembly language uses mnemonics to represent machine instructions, or code.

Basic Processor Operation

The goal of the processor is to take data (the input), perform some form of processing on the data, and then store the data in a useful way (output). To be able to do this, the processor needs temporary storage within itself called registers. The PowerPC family all have two types of registers - integer registers identified as r0 to r31 and floating point registers named f0 to f31. We won't be looking at the floating point registers for a while yet, but they are one of the most powerful aspects of the PowerPC family.

The processor can put data into these registers and then use the registers as inputs to calculations or other operations. The result of the operation can then be stored in another register, or one of the input registers. For example, the instruction add r3,r4,r5 adds register 5 to register 4 and stores the result in register 3, whereas add r5,r4,r5 adds the contents of register 5 to register 4 and stores the result back in register 5.

To show you how easy it is, examine the following PowerPC assembly language "snippet".

```
do_add: li      r3,10          *load register 3 with the number 10
        li      r4,20          *load register 4 with the number 20
        add     r5,r4,r3       *add r3 to r4 and store the result in r5
        stw     r5,sum(rtoc)   *store the contents of r5 (i.e. 30)into the memory
location
        blr                    *called "sum"
        blr                    *end of this piece of code

sum:    ds.w    1              *define the storage we need for the result.
```

The first thing to notice is the layout. All assembler languages tend to be like this. The doing bits of the instructions (store, add, etc) - the actual instruction, is in the "second column". Technically it is preceded by white space - either tabs or space characters. This way the assembler knows they are instructions.

So what is the word "do_add:" up against the left hand edge of the page? A Label. Labels mark a position in the code and are also used when we need to define things, such as data. Notice the label has a colon : after it. When you are referencing these labels in the code, you don't actually use the colon - for example in the line "stw r5,sum". In this case, the label "do_add:" is used as an identifier, or name, for this piece of code. Labels are used instead of real memory addresses because we don't care what address is assigned to the label, that's the assembler's worry. If the assembler knows the address of the label, then we can use the labels in our program and let Fantasm worry about tying up all the addresses. If we want to run a routine called fred, then we can just branch to fred. The assembler will work out what fred's address is and do the right thing.

The colon after the label is not essential when you are defining a label in the first column, but it helps in two ways. Firstly it clearly identifies this thing as a label to the assembler, the editor AND us humans. Secondly, it can be helpful when you are editing the program, and need to find a specific label. If you search for just "sum" you will find all the places that access sum - but if you search for "sum:" you will go to the place where the label "sum" is defined. (note, that Anvil does not need colons at the end of labels to be able to hyperjump to them).

Labels are always placed right up against the left hand edge of the window; there must be no white space in front of them. The label is followed by some "white space", normally a tab character - the tab key on your keyboard, although a normal space, or run of spaces is fine.

Not all lines need a label, just those lines of your program that you want to branch to, or lines that define data that you want to reference by name.

Following the optional label comes the instruction. This tells the processor what it's going to be doing. The instruction is followed by some more white space before the "operands". These are the things the instruction manipulate. PowerPC instructions sometimes have three operands or more, and sometimes none at all - it depends on the instruction. If there are operands, then the first is either the source or destination of the result of the operation. The other two provide the data to be worked on - for example add r3,r4,r5; in this case, r5 is added to r4 and the result is placed in r3.

Any ideas about what this program does? It adds 20 and 10 and puts the result in sum. Here's the breakdown.

Line 1 - do_add: li r3,10

The 'Add' is a 'label'. Labels are used to reference lines of a program so we can change the program flow, by 'branching' to labels. Think of it as a name for this part of the program.

li means load immediate. In this case it means move the word '10' (remember that a word is 32 bits) into the processor register 3.

Line 2 - li r4,20

This line has no label, and therefore can't be branched to. The instruction moves another number, in this case 20, into the processor register 4.

Line 3 - add r5,r4,r3

This line instructs the processor to add the two numbers together and place the result in register number 5.

Line 4 - stw r5,sum(rtoc)

This instruction moves the result of the addition into memory. Where in memory? At "sum".

Line 5 - blr

This instruction returns from this part of the program. if you like it means that's the end of this piece of code so go back to whatever piece of code called this piece of code. "blr" is a mnemonic for branch to link register.

Generally, whenever we jump to a new piece of code, the address of the calling code is stored in the link register so the code can return when finished.

Line 6 - sum: ds.w 1

This line is used to define the location of 'sum' in memory. Its not an instruction to the processor, as the processor has finished with this code in line 5. This line is used by Fantasm to set up 'sum' in memory ready for the program when it is run. The 'sum' tells Fantasm that this is the name we want to use. Again it is a label.

The ds.w means define space as words. Fantasm reserves the number of words needed for this label in memory. This is called a 'directive', meaning that it is a directive to the assembler (Fantasm), and not a mnemonic to be translated into machine code.

If you found that complicated, don't worry as we'll come back to directives and program structure later.

If you had trouble understanding that, read through it again. There's nothing devious or particularly clever about assembly language programming - just common sense most of the time. Staying awake is important too.

If after rereading the above you still have difficulty understanding it, don't worry about it, just carry on with this text, as it was throwing you in at the deep end, however, if you followed it just fine, example project #1 will build the code and throw you into the low level debugger (if you have one installed).

This is what the above code, as a complete program, will look like in Anvil (your colors may be different) :

```
do_add: entry
        start_up
        illegal
        li r3,10
        li r4,20
        add r5,r4,r3
        stw r5,sum(rtoc)
        lwz r6,sum(rtoc)
        tidy_up

sum:    ds.w    1
        global do_add
```

Bits, Bytes, HalFs, Words and Longs.

As you may already know, all computers have 'memory'. Memory is a very wide term, that can be broken down as follows.

1. Long term memory. This type of memory is for long term storage of information.

Theoretically speaking it means any form of memory that doesn't forget when you switch the power off. This boils down to disks, both floppy and hard.

Once the data is written to disk, it stays there until its either overwritten, or your four year old decides to open the case on one of your floppies and play frisbee with what's inside.

There are big differences between floppy disks, and hard disks. Floppy disks are removable, cheap and slow. If you slide the metal cover of a floppy you'll see a brown plastic disk inside the jacket. This is the actual floppy disk. Data is recorded onto the disk, in the same way that music is recorded onto a cassette, one bit at a time using a magnetic recording head.

Because the disk is arranged in tracks (80 on a HD disk), and your Mac can select any one of these tracks at random, you don't have to fast forward over the whole disk to get at the data you want.

The big disadvantage with floppy disks is that they are very slow compared to hard disks.

Hard disks are not (generally) removable, not cheap and not so slow.

A hard disk works in a similar way to floppy disks. They have tracks, and head(s), but because the disk spins a lot faster (5400 R.P.M. and upwards) and the heads are a lot closer to the disk, you can store a lot more information on them.

Hard disks have a reputation for being fragile. This is not just paranoia about them, but a fact. In a hard disk, to get the heads as close as possible to the recording surface, the heads fly on a cushion of air created by the disk spinning. If the hard disk drive is knocked whilst it is in use, its quite possible for the heads to crash into the disk surface! This is a good reason why you should use the shutdown menu item in the Finder, so the drive can move the heads away from the disk before the power goes off and the disk slows down. Most hard drives automatically move the heads away from the disk when the power goes off (this is known as auto parking), but just in case...

All disks, whether hard or floppy segment the tracks up into sectors. This makes the drives more usable. For example if you have a high density floppy disk, capable of holding 1.4 megabytes (1 megabyte is 1 million bytes), and the floppy disk itself has 80 tracks, this means that each track can hold 0.0175 megabytes, or 17.5 kilobytes.

This means that the smallest amount of information that could be written to a floppy disk is 17.5K. If a 1k file was saved to disk, it would still take up 17.5K on the floppy! If however, each track is further split up into sectors, and the drive knew where each of these sectors were, then small files would take up less space. For example if the track was split up into 20 sectors, then the smallest addressable unit on the floppy would be 17.5 K divided by 20, which is 875 bytes. Now a 1k file only takes up 2 sectors on disk amounting to 1.75K.

Getting data off a disk, whether it's a floppy, a hard disk, a CD or any other kind of storage medium is a slow process - far too slow for the processor. The data in long term

memory, such as a program, is transferred to short term memory before it can be run, or accessed by the processor.

The other type of long term memory, associated with Macintoshes, is pram, or parameter random access memory. This memory doesn't forget its data when a Mac is switched off because it has a battery that keeps the memory running.

In here the Mac stores vital information; its configuration, so that then next time its switched on, it can read the set up information from pram, and configure itself exactly as it was. Examples of the data stored in pram are the sound volume, how many flashes a menu bar makes, keyboard repeat speed etc.

(Other computers may have this type of memory. PC-compatibles use a type of memory that is usually referred to as CMOS (complementary Metal Oxide Semiconductor) memory. Using this name is a bit naughty, since it should be called "battery backed C.M.O.S. memory" - C.M.O.S. memory itself loses its contents without power. In PC's this stores things like hard drive type, floppy types, time and date - general set up information., which drive is called drive C: (yeah, real hi-tech stuff) etc.)

2. Short term memory - this is the memory the processor has direct access to. It is split into random access memory (RAM) and read only memory (ROM).

Ram can be written to and read from, whereas ROM can only be read from. RAM is generally faster than ROM.

The more RAM you have, the more data can be stored inside the computer at any one time. If the data is a program, then the more ram you have, the bigger the program you can run.

As was noted previously, the PowerPC range of processors use instructions made up from 32 bits. The basic unit of memory is a byte, which is 8 bits, so the PowerPC needs to read four bytes for every basic instruction.

RAM and ROM. are fast enough for the processor to run programs from. We'll come back to memory later when we talk about caches, but for now that's enough.

Numbers

As you know, computers these days are referred to as being digital, but a long time ago there were analog computers that did their calculations by whizzing around servos linked to potentiometers which would give a result as a voltage. Anyway, "digital" means numerical, so it stands to reason that computers run by using numbers? Quite true, as a number is just a number, but the clever bit is that numbers can also be used to represent codes for such things as what operation to perform., or a letter, or a sound volume etc.

So if a computer needs numbers to strut its stuff, what form are these numbers held in, do we just say "65", and it puts letter A up on the screen in glorious Technicolor? Not quite. The numbers the computer needs to carry out processing are held in the binary format, just the same as the instructions the computer executes. Binary is a number system in which a number can be one of two values, either zero or one, whereas denary means a number can have one of ten values, 0 through to 9.

How can we represent numbers larger than 1 in binary?

Well, lets take a look at our human denary system first.

If we start counting from zero upwards, we eventually get to 11, which means 10 plus 1, then when we get to 101, this means one hundred, no tens and one unit. If we term tens, hundred's and thousands as "multipliers", then any number can be expressed in terms of its multipliers; 1234 can be expressed as:

```
1 times 10^3
+
2 times 10^2
+
3 times 10^1
+
4
```

(note that in programming, * generally means multiply and ^ means to the power of , so $10^3=10*10*10$).

As the computer uses binary and not denary, numbers can be expressed as powers of 2 instead of ten. Where in denary the multipliers go 1,10,100,1000,10000 etc, in binary the multipliers go 1,2,4,8,16,32,64,128,256 etc.

It is helpful to be able to remember the multipliers up to a certain level, here's a list you should try to learn:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536

To express the number 13 in binary, break it down into powers of two.

```
13 is one 8, one 4, and 1 -
8 4 2 1
1 1 0 1
```

Thus, 13 is 1101 in binary.

We can take a very short break here, just to clarify some terms that are about to appear. Binary digits, 1's and 0's are termed "bits", which is just a short form of "binary digit". It's not very hip to keep shouting "Binary digit 3" when "Bit 3" sounds far cooler and saves time. When referring to bit locations within a group of bits (for example, within a word), bits are numbered from zero upwards, with bit zero being the rightmost bit. If you confuse left with right as I sometimes do, now is a good time to finally get it sorted out. Get two of those little "Post it(tm)" notes, and write "Left" on one and stick it on the left side of your monitor, and do the same for the right side, only this time the note should have "Right" written on it.

Ok, so if we take a byte (which is 8 bits) then the bits are numbered as follows:

76543210 - bit 7 on the left and bit 0 on the right.

Confusingly, IBM's books for PowerPC sometimes refer to bits the other way round, but hey, that's IBM.

Ok, back to the lesson....

Now try converting decimal 255 to binary.

Start with the multiplier above 255, 256. This is too great, so try 128.
 255 divided by 128=1 remainder 127. So we have a 128.
 127 divided by 64 =1 remainder 63.
 63 divided by 32 = 1 remainder 31
 31 divided by 16 = 1 remainder 15
 15 divided by 8 = 1 remainder 7
 7 divided by 4 = 1 remainder 3
 3 divided by 2 = 1 remainder 1
 1 divided by 1 = 1 remainder 0

Therefore 255 in binary is 11111111.

The last example is 471. 512 is too big, so:

471 divided by 256 = 1 remainder 215
 215 divided by 128 = 1 remainder 87
 83 divided by 64 = 1 remainder 23
 23 divided by 32 = 0 (because 32 doesn't divide into 23 even once. i.e. it won't "go")
 23 divided by 16 = 1 remainder 9
 7 divided by 8 = 0 (because it won't go)
 7 divided by 4 = 1 remainder 3
 3 divided by 2 = 1 remainder 1
 1 divided by 1 = 1

Thus 471 in binary is:
 111010111 - that's nine bits and it means that
 $1*1+1*2+1*4+1*16+1*64+1*128+1*256=471$ or more simply
 $1+2+4+16+64+128+256=471$

I admit, it isn't easy, but here are some more examples expanded to 8 bits.

	128	64	32	16	8	4	2	1
25 =	0	0	0	1	1	0	0	1
129=	1	0	0	0	0	0	0	1
56 =	0	0	1	1	1	0	0	0
90 =	0	1	0	1	1	0	1	0

In Fantasm, to show that a number is binary, we precede it with a percent sign - %10101010

Eight bits are termed a byte. One byte can hold 256 different values, so every conceivable letter and punctuation mark can be defined in one byte, or alternatively, 256 different codes can be defined, or 256 different colors for a pixel.

As we know, the PowerPC demands its instructions in 32 bit chunks, these are called words - how many possible values are there with 32 bits?

$2^{32} = 4294967296$. (easy with a calculator).

With 32 bits 4294967296. different values can be defined.

With 32 bits making up a basic PowerPC instruction, there are possibly over 4294967296 different instructions the processor could execute - in practice there are about 60 or so

basic instructions. The other bits within the instruction are used to hold the data the instruction works on.

For data, 32 bits can hold big numbers, sometimes it's too wasteful; for example to store the character "A" we only need 8 bits, or a byte. And sometimes you may want an intermediate size, or 16 bits (65536 possible values), so we also have data sizes called "halfs" which is half a word. More on this later.

Hexadecimal Notation.

Hex is just a number system, the same as decimal and binary. In decimal we use base 10, in binary we use base 2, and in hex we use base 16. How can we use base 16 if we only have ten digits (0-9). Good question. The digit set is extended to 15 by using the letters A-F.

If you can understand binary, hex isn't a big deal. Each hex digit represents 4 bits or a nibble.

An example is probably easiest to understand:

Here's an easy one:

255 in binary is 11111111 (8 ones).

To get 255 in hex first convert to binary, then split it up into nibbles (i.e. half bytes),
1111 1111.

Each hex digit is a nibble, so 1111 is 15 in decimal, or F in hexadecimal.

Therefore 255 is 11111111 in binary or FF in hex. To show this is a hex number we precede it with a dollar sign - \$FF or the C language standard of "0x" - 0xFF. The choice of preceding hex number with either a "\$" or "0x" is up to you.

To convert from hex to decimal, convert the hex to binary, then decimal:

convert \$FACE to decimal -

F	A	C	E
1111	1010	1100	1110

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	1	1	1	1	0	1	0	1	1	0	0	1	1	1	0

then add together all the ones:

$$32768+16384+8192+4096+2048+512+128+64+8+4+2 = 64206$$

Now try \$9276

You should get 37494. Of course the quickest way is to use a the computer to do it for you and "drop" into Macsbug! (a debugger, covered later).

The PowerPC family can manipulate numbers as either bits, bytes, halfwords, words and doubles (64 bit via the floating point unit, or FPU).

Now that we know a little of how the computer uses numbers, how are they stored in memory? Because the numbers are represented in binary, which is a string of ones and noughts, its easy to go from theory to practice. In memory, a 1 is represented by a voltage, whilst a nought is represented by no voltage. The memory is laid out in bytes (8 bits), so for the processor to get one instruction it needs to read four bytes.

Four bytes make up one word, so it stands to reason that words must live in quad aligned locations in memory, if the memory starts at location 0 (which it does). Thus if our first instruction was at address 0x80000, then our next instruction will be at 0x80004, then next at 0x80008 and the next at 0x8000C etc

If you really want to know more about the intricacies of the floating point number formats, go check with Random Rob from the Programmers Dream.

A Quick Summary

The number system used in computer is binary. Binary means one of two values. Either a digit is a one or a nought. When counting in binary, the power of two is applied as a multiplier. A bit is one binary digit, either a 1 or 0. A byte is 8 binary digits that can hold 256 possible values. A half is 16 bits that can hold 65536 possible values, and a word is 32 bits that can hold very big values.

Binary numbers are preceded by a percent sign - % and may contain the digits 1 and 0 only.

Decimal numbers are written as per normal and may contain the digits 0,1,2,3,4,5,6,7,8,9.

Hexadecimal number are preceded with either a \$ or an 0x - your choice and may contain the characters 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Addressing

Addressing is the term given to this question. "How does the processor know where the data is and how does the data get from the memory to the processor and back again?"

All computers have busses; highways for information. Typically there is a data bus and an address bus. The address bus tells the memory where the processor wants to read or write to. To get a word from memory location 1000, the processor puts 1000 on the address bus, then tells the memory to "read". The memory system will put the data on the data bus, and the processor can read it in. From now on, instead of the word "location", we'll use the word "address" to mean a location that the processor can access.

Inside the processor the address can either be stored in a "register" or form part of an instruction - for example `lwz r3,fred(rtoc)` - the address we are reading from is "fred". The PowerPC has 32 integer registers, each being 32 bits wide. Programs can modify these registers, so that the processor can keep temporary pointers to memory locations. A register is like a small piece of memory that is internal to the processor, and hence very fast. A PowerPC processor also has 32 floating point registers but we wont concern ourselves with these just yet.

An important point to realize is that it is not only memory and the processor that can access the address and data busses. Most peripherals, such as disk controllers, keyboards, screen driver hardware can also access these busses.

These peripherals are normally given a memory address that's well out of the way of the main program and data memory, so if the computers main memory ends at address \$1000000, the peripheral hardware addresses may start at \$80000000. If the video driver hardware lives at \$80000000, the processor can send and read data by reading and writing to this address.

We'll come back to peripherals later.

Program Counter

There is a special register called the program counter. This one keeps track of where in a program the processor is. Normally it increments by the size of each instruction, as each instruction is read in - that is it increments by four bytes (32 bits) after reading the current instruction so it points to the next instruction. Thus, if the program starts at address 1000, after the processor has read the first instruction, the program counter will be pointing to location 1004.

Normally, the program counter (PC) is incremented by four to point at the next instruction. However programs need a way of making decisions, and going off to do something else if need be. This is called branching or jumping. As an example consider a program that accepts names from the keyboard until ten names have been entered.

The program could go something like this:

step 1: get name 1 from keyboard

step 2: print the name on the screen

step 3: get name 2 from keyboard

step 4: print the name on screen

step 5: get name 3 from the keyboard

step 6: print name....

step 7: get name 4....

step 8: and so on....

As you can see the program is a repetition of steps 1 and 2. What would be nice is if we had a way of using steps 1 and 2 ten times over. By using a conditional check and a counter we can:

step 1: set counter to 1

step 2: get name from keyboard

step 3: print name on screen

step 4: add 1 to the counter

step 5: is the counter equal to 10? This is the conditional check

step 6: if no, then go to step 2

step 7: end the program

Step 6 is a conditional branch - it is taken if the condition is met - if the counter doesn't equal 10 then branch to step 2.

The processor would have to scrap the contents of the program counter and replace it with the memory location for the instruction at step 2.

To be able to perform conditional branching, or jumping, the processor has to have a method of flagging the result of operations.

In the above program the processor needs to know if the counter had reached 10.

How does it do this? It compares the value of the counter to 10. If it equals 10 the processor sets a flag in the "condition code" register. At step 6 this flag is checked. if the flag isn't set, then the program can branch back to step 2.

A compare is simply a subtract operation, but the processor just makes a note of the result (was it positive, negative, equal to zero etc.) and throws the result away.

Summary

Hopefully, we have now covered enough ground to be able to summarize how a computer works and the basics of a PowerPC processor as follows:

1. The computer reads and executes instructions.
2. The instructions act on data.
3. Instructions are read from memory via the data bus. The address in memory from where the instruction is coming from is set up by the processor on the address bus.
4. Data can be read and written to memory via the data bus. Again, the address in memory of where the data is coming from or going to is set up on the address bus.
5. Data coming in from memory to the processor is termed as being read
6. Data going out from the processor to memory is termed as being written.
7. By reading and writing data from certain areas of memory, the processor can control peripherals.
8. The processor knows where to get the next instruction from because the program counter register always points to (holds the address of) the next instruction to be executed.
9. The value in the program counter can be altered as a result of conditional checks during the running of a program.

10. The link register can be used to hold the return address when the processor decides to jump to another piece of code. This return address can be jumped to by executing a blr instruction.
11. The PowerPC processor has 32 general purpose registers. These registers are 32 bits wide.
12. The PowerPC processor has 32 floating point registers. These registers are 64 bits wide.
13. The PowerPC processor instructions generally take three operands.
14. Binary is a number system based around the multiplier 2.
15. Hexadecimal is a number system based around the multiplier 16. The numbers 10 through 15 are identified with the letter "A" to "F". A=10, B=11, C=12 etc.
16. When identifying bit positions, bit zero is the rightmost bit.

PowerPC Assembly Language Beginners Guide Chapter 2

A few Instructions, Variables, TOC and BBS

The first thing we need to be able to do is get data into the processor; if we can't do this, then we can't do any work. The basic instruction used for loading data from memory is, funnily enough, called "Load". There are a few variants on it, depending on the size of the data we wish to load. The most frequently used, loads a word from memory into one of the processors registers. The full name of this instruction is Load Word and Zero, or LWZ as Fantasm knows it. LWZ takes the form of:

```
lwz rx,EA
```

Where rx is an integer register, such as r3 and EA describes the effective address of where to get the data from. The data will be loaded into rx affecting all bits of the register, as on most PowerPC processors the integer registers are 32 bits wide (there are 64 bit PowerPC processors such as the 620. If an LWZ instruction is processed on one of these, the upper 32 bits are set to zero, hence Load Word and zero).

LWZ is complimentary to Store Word, which takes data (32 bits) from a register and stores it in memory.

Registers can be loaded with data quite easily with the Load Immediate instruction; LI. This takes a 16 bit value and puts it into the lower 16 bits of a register. Because the register is 32 bits wide, the 16 bit value is sign extended to affect the upper 16 bits. This means that if bit 15 is a 1, bits 16 to 31 are made 1's. If bit 15 is a zero, bits 16 to 31 are made zero's. This Load Immediate loads a 16 bit signed number into a register

Suppose we want to set a 32 bit variable called my_variable to 0x00001234, here's what the code would look like:

```
li    r3,0x1234
stw
r3,my_variable(`bss)
```

You're probably thinking "what's this bss thingy?" Ok. don't panic, it's not hard, we'll cover it in a second under "variables". Notice in the above example I cheated by loading in a 16 bit value that would not have bit 15 set, so 0x1234 comes out just as 0x00001234 when loaded into the register.

If I had loaded 0xf234 the register would have been sign extended to become 0xfffff234, which may not be what we wanted. If we wanted to load a 32 bit unsigned value into a register with Fantasm 5 we could use a "macro" called "movei". Macros, at the simplest level are a way of defining new instructions to do things you couldn't normally do. movei is used exactly the same as li:

```
movei r3,0xf234
```

After this macro, r3 would be equal to 0x0000f234. Hopefully you can see the difference. To use this macro in Fantasm 5 you need to either make LS_PPC_macros.def a globally included file (these are files that are inserted into every file in your project) or include the file into your file with an "includeh" directive. More about directives later.

Variables

Note, we get a little complicated here for a while. No need to understand this section completely just now, I just want to explain some things so that when you see them in the examples, you'll have at least seen them before.

In a high level language, sometimes you need to define variables before using them, and sometimes you don't. In assembly language you do need to define your variables. Generally we can use two kinds of variables - global or local. Global variables are accessible to all parts of your program, whilst local variables are only accessible to the routine or function (a part of a program that can be called many times over from other parts of the program) that they are defined in.

Global variables are stored in a section of RAM called the BSS. Nobody we've ever met can tell us why it's called the BSS - it just is. (If anybody knows why, please tell me). When Fantasm is building your program, it calculates the size of the BSS and stores it in the fragment. When the Mac loads the fragment, it sets aside an area of memory big enough to hold all your global variables and then stores the address of this area in the first entry in the fragment's Table Of Contents or TOC. If we then copy this address into a register, it can be used as a pointer to the global variables. We can then access (read from and write to) our global variables as offsets (using labels) from this register. The start up macro performs this function for us (along with another important function of saving all the registers before we start messing with them), and puts the pointer to the BSS into r30. Fantasm can rename registers to anything you like (with the "reg" directive), so if we call register 30 "bss", then we can access global variables with statements such as my_variable(`bss). Note the use of the ` character to identify the name as a register name.

Local variables are stored on the stack. The stack pointer (r2, although Fantasm also knows it as "sp" as well) points to a free area of memory and is generally used to store temporary data such as local variables, return addresses (so a subroutine can return to its caller)

and sometimes parameters (data that subroutines work on; passed to them from the caller; in PowerPC parameters are nearly always passed in registers because there are a lot of registers, and accessing registers is far faster than accessing memory).

Local variables, as previously noted, are accessible only to the current subroutine. In Fantasm 5 we can define local variables with the "local" macro and then access them via the stack pointer - abbreviated to "sp". The code will look something like this:

```
lwz    r3,my_data(sp)
```

For this to work correctly, at the start of the routine we need a list of variable definitions along with a macro that sets up the stack correctly, and at the end of the routine we need a macro that will clean up the stack before we exit the subroutine. These macros are called:

reset_locals - does as it says, resets the local variables macros and need to be used before you use the local macro.

local - define a local variable. This one has a size character tagged onto it, such as local.w - defines a local variable of size word. It is used as: local[.size] variable_name[,number_of_bytes/halves/words] sub_entry - the macro that performs the stack set up at the start of a routine.

sub_exit - the macro that performs the clean up operation at the end of a subroutine.

Thus, a typical subroutine that needs local variables (normally one uses registers because there are lots of them) would look like this:

```
reset_locals
local.w
my_var,1
local.b my_string,256
local.h
my_2d_array,10*10

my_subroutine: sub_entry
some code

sub_exit
```

In the above subroutine, my_var defines space for a 32 bit variable.

my_string defines space for a 256 character string and my_2d_array defines space for a 10 by 10 two dimensional array of halves, or 16 bit values.

Heavy going? Don't worry about it if you find it so, it's always easier to learn by example which is where we're heading. All I'm trying to do is give you a basic grounding from which we can build.

Initialized Data?

Variables are very useful, but sometimes we need pre-defined data, or initialized data as it's called. For example, we may need to define some constants for our program, and what about strings - for example "Fluffy loves socks". How do we get that into memory so we can print it for example?

This is where the data section comes into play. Unlike the BSS which is set up at run time, the data section is stored along with your program on disk - it takes space, but is necessary. The data section is set up by Fantasm (more specifically the linker) and is comprised of all the data you have defined in your source files. At the lowest level, data is defined with the "dc" directives. "dc" is an abbreviation for "define constant". The dc directive allows you to define data that will be available to your program at run time. Example:

```
fluffy_string: dc.b    "Fluffy loves socks"
               align
```

That line defines the string "Fluffy loves socks" and calls it fluffy_string. Just for now, note the "align" directive after the definition - this makes sure the next definition is aligned correctly in memory.

When we need to access this string we can use the label "fluffy_string". Another word for "label" when referring to data and variables is "identifier" which I'll use from now on.

Fantasm takes all your data identifiers and data from all your source files and passes them to the linker which "munges" them all into the data section.

A pointer to each item of data is stored in the Table Of Contents (TOC) and the data identifier for each item is related to the slot in the TOC that points to the correct data. This is necessary because when your program is launched, the data section can go at any address in RAM - it's not always the same address, so the Mac has to "relocate" all the pointers to data.

These are the pointers that are stored in the TOC. Sounds complicated right? The only practical offshoot of it is that when you access an identifier for initialized data, just remember you are getting a pointer to the data.

Another way of looking at it without the technicalities is: The TOC is pointed to by a register called rtoc, which is really just r2.

This register points to a program's Table of Contents. This is simply a table that points to all initialized data (in the data section) that we gave identifiers to. Again, please forgive me but I am oversimplifying things a little, we can have code pointers and pointers to function descriptors in the toc as well, but we need things as clear as possible at this stage.

Thus a fairly typical load may look like this:

```
lwz r3,fluffy_string(rtoc)
```

This will load the address of fluffy_string into r3, because the rtoc contains pointers to the data - so we're loading the pointer to the data into r3.

We could then load the first character of the data as follows:

```
lbz r3,(r3)
```

"lbz" means Load Byte and Zero. (r3) means the contents of the address pointed to by r3, and not the contents of r3 itself.

In this case, the lower byte of r3 would now contain the character "F".

So, lets summarize what we've covered so far:

1. lwz is an instruction that Loads a word into a register.
2. stw is an instruction that Stores a word into memory.
3. li is an instruction that loads a 16 bit value into a register and sign extends it.
4. movei is a macro that loads a 32 bit value into a register.
5. Variables come in two forms; local and global.
6. Global variables are accessible to any part of the program and are stored in the BSS.
7. Local variables are only accessible to the subroutine they are defined in (not strictly true in assembly language by the way - you can do anything you like, but this is the rules for beginners).
8. Local variables are set up with some macros called reset_locals, local, sub_entry and sub_exit.
9. Initialized data is stored in the data section and accessed through the TOC via register rtoc.
10. The TOC contains pointers to data, not the actual data itself.
11. Identifier is another word for "label" when used with respect to data. The word "label" is used for code, "identifier" for data.

Phew, let's give it a go hey?

OK, first make sure Macsbug is installed on your machine. To do this, press the Apple and Power On keys on your keyboard. If Macsbug pops up, lovely jubbly. Type G <return> (<return> means hit the return key on your keyboard). "G" means carry on running, or Go.

If not, if you get the little dialog box with a prompt that looks like this:

```
>
```

then you do not have Macsbug installed. Again type G <return> to get out of the box.

To install Macsbug, drag it over your System folder and restart your Mac.
You can get Macsbug from Apple for free.

Ok, so Macsbug is installed - this will allow us to explore our programs as they are running. Now we need to write that little program above and step through it to ensure r3 get's loaded with the pointer to fluffy_string.

I've uploaded this simple project to the server - download it by clicking here. It's a Stuffit archive, so if your browser doesn't do it automatically, you may need to unstuff it using Stuffit Expander (tm).

```
example1:      entry
                stdx   r1,r2,r3
                lwz    r3,fluffy_string(rtoc)  *r3 points to fluffy_string
                blr    *simple end of our program - branch to link register.
```

```
*****Data*****
fluffy_string: dc.b  "Fluffy loves socks"
                align
```

```
*****End of program*****
```

Lets just run through it. "example1:" is the label, or name of this program. "entry" tells Fantasm where the program starts. "stdx r1,r2,r3" causes the processor to stop in Macsbug (because it's a 64 bit instruction and a 32 bit processor doesn't understand it) so we can see our program. There is a macro that does this, called "Debug" but we won't use that yet. "lwz r3,fluffy_string(rtoc)" we've already covered. "blr" is a PowerPC instruction that means Branch to the address contained in the Link Register. This instruction was covered briefly in Chapter 1. In this case it ends our program.

After you've downloaded and expanded the project, open it from Anvil (it's called "example1_prj"), then build it (APPLE B) and then run it (APPLE R).

Hopefully you should now be in Macsbug. Down the bottom of the screen you should see something like this:

```
020CA238 *dc.l      0x7C22192A      | 7C22192A
020CA23C lwz       r3,0x0004(RTOC) | 80620004
020CA240 blr       | 4E800020
```

The line that reads *dc.l 0x7c22192a is our "stdx r1,r2,r3" and is how we broke into Macsbug (with an illegal instruction). Below that we can see our assembled two lines of program, the lwz and the blr.

If you enter the following command into Macsbug (where <CR> means press the return key):

```
pc=pc+4<CR>
```

we will skip over the illegal instruction to point to our load instruction, lwz. We can run this instruction by holding down the APPLE key and pressing the "S" key. r3 will now contain the address of fluffy_string. We can check by displaying the memory that r3 points to with the following Macsbug command:

```
dm r3<CR>
```

We should hopefully see that r3 does indeed point to the string "Fluffy loves socks".

Note: You will see the string "Fluffy loves soc". This is because the dm r3<CR> command only shows the first 16 characters. To see all the string press the <CR> key a couple of times.

Now type:

g<CR>

and your Mac should run as normal.

Just a quick Macsbug note here. If you have more than one monitor, it may be worthwhile making your start up screen anything other than your main screen. Macsbug always works on the startup screen, so this way you can see Macsbug and whatever your program is doing at the same time.

And that is a lot to take in in one chapter. In the next we'll look at logical operations (sorry, but it has to be done) and a more complex program that utilizes all we've covered here - it may even do something! In the meantime, you may like to have a look at the "PPC_graphics_demo" example that can be found in the examples folder next to wherever you installed Fantasm 5. See if you can figure out roughly what's going on, but don't worry if it just looks scary.

PowerPC Assembly Language Beginners Guide Chapter 3

Logical Operations

AND

Logical operations are quite simple to understand and form a useful tool within the programmers armory. An "and" operation simply says if both A and B are equal to a logical 1, then set the result to a 1. Logical operations work at a bit level - each bit of both operands is logically tested and the same numbered bit in the destination is set or cleared as a result of the logical operation as applied to each bit. We humans can work them out serially; by starting at the first bit of each operand and making a note of the result of each operation. The CPU operates on all bits in parallel, hence all logical operations (in common with most PowerPC instructions) have an effective processing time of one cycle. The exception to this timing are integer multiplies and divides - avoid these like the plague if you can or use the FPU which is a multiply-add unit and can perform single sized (32 bit) multiplies in one clock cycle effectively and double sized (64 bit) multiplies in two.

Back to the logic...

Suppose we "and" 1 and 9. If we look at the number in binary, 1 is 0001 and 9 is 1001. When these two numbers are anded, the processor looks at the numbers like this:

```
3210 <- Bit number
0001 <- 1 in binary
1001 <- 9 in binary
0001 <- result
```

First it will look at both bit 3's. It says I have a 1 and a 0, so I don't have two 1's. Therefore the result is zero. Then it looks at bit 2's, which are both zero, so the result is zero. Bits 1 are both zero, so the result is zero. Bits 0 are both 1. It says if I have a 1 "and" a 1 then the result is 1, so bit zero result is 1.

The result of 9 anded with 1 is 1.

The and operation can be summarised by saying "only if both bits are set will the result be set"

Examine the following examples, to see if you can spot the main use of the "and" instruction.

What is the result of 0xFA anded with 0x0F?

```
0xFA = 11111010
0x0F = 00001111
AND =
00001010 = 0x0A
```

What is the result of 0xF1 anded with 0x0F?

```
0xF1 = 11110001
0x0F = 00001111
AND = 00000001 =
0x01
```

What is the result of 0x1220 anded with 0x00FF

```
0x1220 = 0001001000100000
0x00FF = 0000000011111111
AND =
0000000000100000 = 0x0020
```

The and instruction is mostly used to mask off wanted data in a register. By setting bits in the mask that identifies the bits you want to keep, and then anding this mask with the data, the bits you are not interested in will be set to zero.

For example if you had a routine that returns the ASCII value of a key pressed on the keyboard, and it returned the key in r3. The key can be specified in a byte, but there may be data from earlier processing in the upper three bytes of r3 - so to ensure you don't create errors further in the program, the byte can be masked off with the and instruction as follows:

```
andi. r3,r3,0xff
```

Irrespective of how much garbage is in the upper 24 bits of r3, after this instruction all that will be left in r3 is the byte defining the key press, the lower 8 bits. Note in particular the trinary operands and the dot at the end of the instruction.

If we wanted we could leave the contents of r3 intact and place the results of the AND operation in another register, say r4, with an instruction such as:

```
andi. r4,r3,0xff
```

The dot means that this instruction always affects the condition code register; a note is made of the result of the operation. If for example, the result of the AND was that all bits were cleared, then the Z (or zero) flag in the condition code register was set. We will examine the condition code register in more detail later, but for now just note that there are 8 integer condition register "fields". An instruction followed by a dot means that the condition code register field 0 is affected - it notes the result of the operation. NOTE also that immediate type instructions such as `andi.` always work with 16 bit unsigned immediate data. There are shifted forms of the instructions which will affect the upper 16 bits of a 32 bit register, as an example see the `movei` macro from chapter 2.

NOTE: The syntax when introducing new instructions is of this form:

```
instruction[.] rx,ry,rz
```

where instruction is the mnemonic, `[.]` if present means you can use either the dotted or undotted form where the dotted form will affect the condition flags (`cr0`), `rx,ry,rz` are general purpose registers, `fn` means a floating point register, `ui` is an unsigned integer quantity (normally 16 bit, noted if different) and `si` is a signed integer quantity.

Examples of possible and instructions:

AND - `and[.] rx,ry,rz`

The contents of register `rz` is anded with register `ry` and the resultant 32 bit pattern is stored in `rx`.

AND with complement - `andc[.] rx,ry,rz`

The contents of register `ry` are anded with the complement of `rz` and the resultant 32 bit pattern is stored in `rx`. Complement means the inverse of - for example `%1010` when complemented becomes `%0101`. Not to be confused with two's complement which is how computers subtract via addition. Two complement means to invert the data and then add 1.

AND Immediate - `andi. rx,ry,ui`

The contents of register `ry` are anded with the `ui` and the result stored in `rx`. Note that in this case, the upper 16 bits of the result will be cleared because `ui` is a 16 bit quantity! Note that this instruction is always dotted.

AND Immediate shifted - `andis. rx,ry,ui`

This is basically the same as `andi`, except the `ui` is shifted left 16 bits before being anded with `ry`, so the lower 16 bits of `rx` will always be cleared after this instruction.

OR

The OR instruction works like this:

If either or both of the bits are 1, then the result bit is a 1. The other way of looking at it is "If both bits are a zero then the result is a zero, otherwise its a 1".

Example - OR 1 with 2

1 = 0001

2 = 0010

OR= 0011 = 0x03

In PowerPC, the possible instructions are basically the same as and as follows:

OR immediate - ori rx,ry,ui (note no dot allowed unlike andi. which must have one)
 OR immediate shifted - oris rx,ry,ui (note no dot allowed unlike andis. which must have one)
 OR - or[.] rx,ry,rz
 OR with complement - orc[.] rx,ry,rz

Fantasm uses the ori instruction to provide the "extended mnemonic" nop which stands for NO Operation:

nop is the same as ori 0,0,0

The or instruction is used to provide the useful extended mnemonic mr, which means move from one register to another:

mr rx,ry - move the contents of ry into rx and is the same as or rx,ry,ry

XOR

The eXclusive OR instruction works like:

"If one bit is a 1 and one bit is 0 then the result is 1, otherwise the result is 0".

Example - XOR 1 with 15

1 = 0001
 15 = 1111
 XOR= 1110 = decimal 14 or
 0x0e

XOR 0 with 1

1 = 0000
 0 =
 0001
 XOR= 0001, so the result is 1.

If we XOR the result with 1 again we get a 0 because both bits are now a 1. This is a neat way of toggling a bit, every time a loop executes for example. Initially the bit is set to 1. Each time round the loop, the bit is XOR'd with 1. Every time the loop executes. if the bit is a 1 its set to a 0, and if its a 0 its set to a 1.

What's the use of this? Suppose you want to flash something, say an alien spaceship on the screen between red and yellow. You simply xor the colour control bit with 1 and if it was a 1, use the colour red or if it was a zero use the colour yellow.

Possible XOR instructions:

xor immediate - xori rx,ry,ui
 xor immediate shifted - xoris rx,ry,ui
 xor - xor[.] rx,ry,rz

Miscellaneous logical instructions

NOR - NOT OR

NOT means to invert, so NOR performs an OR operation on the two operands, then inverts the result and stores it in the destination register:

`nor[.] rx,ry,rz` - the contents of `ry` are ORed with the contents of `rz`, then the result is inverted and stored in `rx`.

Fantasm uses the NOR instruction to provide the "extended instruction" NOT:

`not rx,ry` is the same as `nor rx,ry,ry`

NAND - NOT AND

NAND performs an AND operation on the two operands, inverts the result and stores it in the destination register.

`nand[.] rx,ry,rz`

EQV - Equivalent

`eqv[.] rx,ry,rz` - the contents of `ry` are XORed with the contents of `rz`, the result is inverted and stored in `rx`.

Sign extension instructions

Extend sign byte, Extend sign halfword, Extend sign word

`extsb[.] rx,ry`

`extsh[.] rx,ry`

`extsw[.] rx,ry`

These instructions copy the sized data (byte, half or word) to another or the same register and sign extend the result, so

`extsb rx,ry` copies the byte out of `ry` into `rx` and copies bit 7 to bits 8 through 31 of `rx`.

`extsh rx,ry` copies the lower 16 bits out of `ry` into `rx` and copies bit 15 to bits 16 through 31 of `rx`.

`extsw rx,ry` is a 64 bit instruction that is illegal on 32 processors such as 601/3/4. It copies the lower 32 bits out of `ry` into `rx` and copies bit 31 to bits 32 through 63 of `rx`.

NOTE that Fantasm 5 will assemble most 64 bit instructions (i.e. PowerPC 620 processor) just fine - you can turn on warnings about their use from Fantasm's preferences pane.

Count Leading Zeros (Word or Double)

`cntlzw[.] rx,ry`

`cntlzd[.] rx,ry`

These two instructions, one for 32 bit architectures and one for 64 bit (the double form) counts how many zeros from the leftmost bit position until the first binary 1 is encountered. The count is stored in the destination register.

We mentioned about the dotted form of some instructions and that if the dot was used the condition flags would be set. Your first question may be why not use them all the time? The answer is simply that to set the condition register flags the processor needs to do more work. Sometimes you may not care if the result of a logical operation was zero or not - you just want the data. In this case you would not use the dotted form. Some times you do need to know so you can make a decision based on the outcome of the operation; in this case you would use the dotted form.

This will form the basis of the next chapter, where we'll be looking at the processor in more detail and examining the condition register and branch processor in general along with a closer look at the whole architecture and introducing the branch processor instructions. We will then have enough information "under our belts" to be able to produce some rudimentary programmes to introduce more integer instructions.

In the mean-time, it would be worthwhile revising the first three chapters and if you're not subscribed to the Fantasm list server maybe you'd like to consider it. Some very useful discussions crop up from time to time.

Author note: Many people have made me aware that this series was moving rather too slowly for their liking :-) so I have rearranged things somewhat. This chapter is not the one that was planned which was a discussion of data sizes, the toc and bss sections. Instead I have opted for a more practical approach. These things are used in what follows but I figure people are smart enough to figure out what's going on from example - for instance, how to use global variables off of the bss and how the toc points to data in the data section. I will of course fully explain these things in a future chapter.

Some of the text in this chapter is reproduced out of the old beginners guide.

We appreciate all your email, corrections and feedback regarding this series, so don't be shy!

An overview of current PowerPC processors.

601 - This first generally available processor is intended as a bridge between POWER and PowerPC architectures. It has three separate pipelines: The Branch Processing Unit (2 stages), the Integer Unit (3 stages) and the Floating Point Unit (four stages) together with a unified 32k instruction and data cache. A 64 bit data bus and a 32 bit address bus. Speed ranges from 50 Mhz upwards.

603 - This is a true PowerPC implementation designed for high performance and low cost. This has four separate pipelines: Branch Processing Unit (2 stages), Integer Unit (3 stage), Floating Point Unit (Six stage) and a Load/Store unit (five stage). Coupled with separate 8kb data and instruction caches. 64 bit data bus, 32 bit address bus. The data bus can be configured for 32 bit operation. What is confusing is that the 603 is less powerful than a 601 but is available in speeds up to 350 MHz and beyond.

604 - This processor is designed for mid-price workstations. It has Six separate pipelines: The Branch Processing Unit (2 stage), three Integer Units (three stage), Floating Point Unit (six stage) and a Load/Store Unit (five stage), together with separate 16Kb data and instruction caches - it is designed to run upwards from 100Mhz.

604e - Basically a 604 with bigger caches (32K a piece) and some tweaks.

620 - The first full 64 bit implementation. Similar to a 604 except the caches are 32 Kbyte each. It also has an embedded secondary cache controller to drive standard Static RAM chips.

750 - 32 bit modified 603e with direct connect second level cache. Optimized for integer rather than floating point operations. For all intents and purposes it can be considered a tweaked 603e.

The best news is that at last, floating point is an integral part of the specification - that is ALL PowerPC chips have an FPU, so we can start using real numbers rather than integer imitations, making life a lot easier for everybody, and because the FPU's run in parallel with the rest of the processor, it's faster too.

This is RISC isn't it - shouldn't we be a little scared of it?

Well, that's up to you - personally, having used PowerFantasm for the last two years, I'd rather write in PPC any day. Ok, so the transition is a bit traumatic - it's all brand new, but once you get into it, it's great.

Here are the big differences:

1. If you have been used to 68K, you'll know that you can perform operations on data in memory - e.g. `addq.l #1,fred(a5)`. In PowerPC you can only perform operations on data in registers.

This is a real bind, but OK once you get the hang of it.

2. PowerPC instructions very often have lots of operands, and they are backwards compared to 68K.

For example: `add r3,r4,r5`

Adds r5 to r4 and stores the result in r3

3. Flags are NOT implicitly set when you move data. For example in 68K, if you move `w fred(a5),d0`, if fred contains zero, the zero flag will be set. In PowerPC this is not the case - you need to explicitly compare the data, either with a `cmpwi` (CoMPare Word Immediate instruction) or by using a dotted form of an instruction - `addic. r3,r4,r5`.

4. Everything must go through the processor - so no moving memory to memory instructions.

5. The sizes of data is (are?) different:

PowerPC 68K		Description
Byte	Byte	Same as a byte on anything else - 8 bits.
Halfword	Word	16 bits
Word	Longword	32 bits
Longword	Not used	64 bits

That in a nutshell is the main differences. We assume you have read the previous chapters about how data is represented, what logical operations (AND, OR etc) are, and how a computer actually runs. We'll dive straight into the practicalities. We don't doubt for a moment this guide will turn you into an "on the metal" PPC coder overnight - there is a lot you can do with the PowerPC chip - we just want to get you walking, the running is up to you.

The practical basics

Any program can be split into 3 logical sections - initialization, processing, termination.

The initialization stage consists of loading the program into memory, setting its variables to the right starting values, and setting up any storage space, or memory, the program may need.

The processing stage is when the program actually does what it's intended to do and produces its results, and finally the program must exit gracefully from the system.

These three stages can be broken down into smaller and smaller sections until one is finally happy that the "algorithm" or mechanical design of the program can be translated into actual processing statements, or instructions. With that in mind, our first example will be adding two numbers. Before we start, just take some time to scan over the PowerPC instruction set as given in Fantasm's reference manual (LSA0041). Don't just look at the likely candidates for the upcoming example, but take some time to examine all of them. Print it out, take it to school or work, and just browse through it.

We will put two numbers into registers, and add them together - see if you can scribble down the program, then compare it to the one given below, meanwhile we'll have a little interlude in the form of "installing Macsbug".

The version we will be talking about here is 6.5.3 or later which runs just great on PowerMacs.

If you haven't already, put Macsbug in your system folder and reboot. Now by hitting the APPLE key and the Power On/Off key on your keyboard, you should drop into Macsbug. Type "G" return, and you should be back to where you were before entering Macsbug. Now that's installed we can call Macsbug from our programs, in order that at relevant points we can examine the registers and memory.

If you have the Extension Manager on your PowerMac, make a new set called "Programming" which includes a minimal set AND Macsbug - by minimal we mean just the bare essentials - for example you don't need ~Aaron running.

Ok, back to our first example - lets check out the PPC's maths.

Here it is:

```
li    r3,4    *First number is 4.
li    r4,8    *8 is second number
add   r5,r4,r3    *add r3 to r4 and store result in r5
```

Load Immediate (li) is actually an "extended" instruction provided by Fantasm, formed from the "addi" instruction - li is actually addi rx,ry,si where ry is zero, so the instruction

adds the si (signed integer) to zero and stores the result in rx. The signed immediate data is only 16 bits, not the full 32 bits and so is sign extended to 32 bits before the addition is performed as the PowerPC ALU (Arithmetic and Logic Unit) only deals with 32 bit operands (64 bit processors excepted). NOTE: In any trinary operand instruction, if ry is r0, it is taken as zero, zilch, nothing but only some instructions, mainly arithmetic - add and sub - can use this form.

The program then, loads decimal 4 into r3, decimal 8 into r4, then adds r3 to r4 and stores the result in r5.

A quick note about numbers in Fantasm. You may use binary, hexadecimal, decimal and ASCII for normal numbers, Floating point (scientific notation - 1e6, 3.142 etc) can be used for some floating point directives. Binary numbers are preceded by a percent character - %101101. Hexadecimal can be preceded by a dollar character - \$f0fe OR by 0x as in C - 0x1234f0fe. Hex numbers can be in lower or upper case - 0xF0FE. Character constants up to 32 bits can be defined by enclosing the string in double quotes (all strings in Fantasm are delimited by double quotes) - "FRED".

Before we actually try it out, we need to know a little about the practicalities of a PowerPC program for the Macintosh. First off, as you may be aware, every "native" Mac program has a "TOC", or more correctly a "Table Of Contents". The toc is a table of data pointers in the programs data section that points to any initialised data in use. The physical register is called rtoc and is really the PowerPC integer register r2, viz:

```
lwz    r3,fred(rtoc)    *r3 is now pointing to fred which is a pascal type string.
Xcall  DrawString      *Print the string
blr
fred:  pstring "Hello!" *5,H,e,l,l,o is placed in the data section by Fantasm.
```

The physical run time interfacing problems are handled in Fantasm with some handy "macros" (new term I know, I'll explain in a second) that take the stress out of these "interfacing" procedures. These are simply used at the right times, and all will be well (apologies for sounding so patronising but I don't want these terms to get in the way of what we are trying to learn here). For example the first lines of any native program you write should be:

```
Entry *Tell Fantasm where execution starts
start_up *A macro that saves the current machine state. Use tidy_up before exit
```

This will cause Fantasm to insert the macro "start_up" at this point in the source code. When your program finishes, you use "tidy_up" and when you want to call an Operating System function you can use "Xcall <OS function name>".

Note that the PowerPC assembler is "case sensitive" - this means that Xcall is different to XCALL - XCALL will not work. This is because in PowerPC, all Macintosh Operating System functions are called by name. The calling mechanism is case sensitive, so it makes sense that the assembler is also case sensitive. For more information on the PowerPC assemblers' label definitions (instructions, directives etc) see the manuals LSA00040 and 41.

You may have used library functions in Fantasm before - these are pre-assembled code snippets designed to perform a simple function - for example "Getkey" returns a keyboard key (if any). Well a macro is just as handy as a library function, except that a macro is simply text inserted where its name is used. You can examine these macros in Anvil - open the file "LS_ppc_macros.def" in the "Anvil Low Level Defs" folder. Fantasm macros can get incredibly complex (for example we have a set that translates 68K assembly language to PowerPC), but

for the sake of this guide, the only thing you need to know is that we'll be seeing a lot of these three macros - start_up, Xcall and tidy_up.

Armed with this knowledge, we can write the practical version of the add program as follows:

```

        includeh      ls_ppc_macros.def          *include this file from "headers"
        includeh      general_usage.def         *and this one
**Program to add two numbers
        entry          *tell Fantasm where program execution starts
        start_up      *the start_up macro as detailed above - sets up the toc etc
        Xcall  Debugger *call Macsbug so we can see it happen.
**Processing start
        li      r3,4      *First number is 4.
        li      r4,8      *8 is second number
        add     r5,r4,r3   *add r3 to r4 and store result in r5
**Processing end
        tidy_up        *clear up processors registers and get ready to exit
        blr           *back to system.

```

There are two new instructions we are not familiar with. The first line has the instruction includeh - if you've swotted up on Fantasm's manuals you may know that this is not an instruction at all, but actually a "directive" - a command to Fantasm. In this case it tells Fantasm to include a file from the "low level defs" folder - in this case the PowerPC macros so this program may have access to start_up etc.

NOTE: If you use include directives, they should be at the the very start of the file.

The last line of the above program also has a new instruction - blr - Branch to Link Register. This is a register that can contain the return address for a subroutine, or any other piece of code. The start_up macro saves it for us, and tidy_up restores the link register, so when we execute a blr instruction, the processor branches to the contents of the Link Register - in this case, back to whatever launched our program (normally the Anvil).

If you branch to your own subroutine, with a branch and link instruction (bl), you must save the Link register (probably in another register) so you can restore it to return to the caller under the following circumstances: 1. You use the Xcall macro to call an OS function. 2. You branch and link to another subroutine.

If your subroutine does not do either of these, then there is no need to save the link register.

Note that Xcall destroys the Link Register and the Count Register (which we haven't talked about yet, but is included in this discussion for accuracy).

E.g.

```

        bl      my_function1  *Call a routine called my_function1
        add     r3,r4,r5
        the rest of your program
my_function1:
        mflr   r29           *Save the return address (currently in the link register) in r29
        your processing code
        mtlr   r29           *Restore the return address into the link register
        blr    *branch to the link register (back to the caller)

```

The bl instruction branches to a routine and saves the next instruction address in the link register - in this case it is the address of the add following the bl instruction.

Making the project I could have made this incredibly easy for you and simply uploaded a Fant 5 project, but I haven't on the grounds that this is as good a time as any to learn how to create a new project with Anvil.

Here's what we're going to do. First we will create the project. Next we will create a source file and enter the code. Finally we will build the project and then run it and examine the program with Macsbug.

Follow me though:

1. Launch Anvil and from the project menu select "Create New Project".
 - 1a. Click on the little Apple help icon in top right to get the items titles displayed.
2. From the New Project dialog, select the template "PPC Fantasm App" from the "Project template" pop-up menu at the top of the dialog. When we build the project this will give us a standard Macintosh application.
3. Give the project a name in the text box labelled "Target name?" 4. Click the big "OK" button. You will be asked with a file selector where to create the project. Find somewhere, maybe make a new folder for it and click OK. The project will be created and opened. You will note the project window that opens has the build and run icons crossed out in red. This means you can't build the project (there are no files) and the project has not been built, so you can't yet run it.
5. From the Edit Menu hit New to create a new text file - it will be called "Untitled 1".
 - 5a. If you haven't set up Anvil's general preferences to default to PowerPC, select "This file's preferences" from the edit menu and change the language to PowerPC. After this operation Anvil will ask you to save the file - give it a name and save it next to your project file.
6. Enter the program text as above and if you haven't already, save the file and give it a name.
7. Add your new file to project (it will appear in the project window in the _Src area) and then Build the project (Use Apple B, click the Anvil icon in the project window or select Build from the Project menu).

It will build and you will find the target icon is now available. Click it, or hit Apple R to run your program.

If Macsbug is installed you will immediately enter Macsbug, if not you will get an error reporting an unimplemented trap and you will have to reboot, install Macsbug and try again.

Assuming you are in Macsbug, hitting APPLE S three times should display the following lines:

Step (into)

No procedure name

```

004CE9D0  lwz      RTOC,0x0014(SP)| 80410014
004CE9D4  li       r3,0x0004      | 38600004
004CE9D8  li       r4,0x0008      | 38800008
    
```

```
004CE9DC  add      r5,r4,r3      | 7CA41A14
```

APPLE S is the Macsbug command to step an instruction - that is, run just one instruction then stop again. The first instruction `lwz RTOC,0x0014(sp)` is the last instruction of the Xcall macro - you will always see this line if stepping a system call in PowerPC. The next three lines are our program, and the lines following that are the "tidy_up" macro code.

In Macsbug, enter "G" followed by the return key - you will be returned to Anvil. Didn't hurt too much I hope :-)

Now, we need to follow the program through - run it again (APPLE R), and step past the first `lwz RTOC,0x0014(SP)` instruction. Macsbug should now be pointing at the first line of our program - `li r3,0x0004`. To the left of the disassembly, you will see the processors registers. Step over this instruction with S return and examine r3 - it will contain 4. Now step the next instruction, and r4 will contain 8. Now step the final instruction `add r5,r4,r3`, and r5 will contain `0x000C` - which is hexadecimal for 12. If you like, step through the instructions that follow (the tidy_up macro code), and eventually you will come to the `blr` instruction - at this point, when you execute this, your Mac will switch back to 68k code, and you will be in Anvil's code - just type "G return" to run Anvil.

Easy? Any problems? If yes, re - run through the above until you understand what we did. If you are really keen, you can modify the program to your hearts content. Remove the Xcall Debugger line for example (tip - just comment it out rather than deleting it - make the first character of the line a semi-colon or a splat character "*").

A closer look at the architecture.

By now you should be getting the whole point of RISC architectures - the instructions are simple, there's lots of registers and things happen quickly. Whereas in 68k, one tends to use the stack extensively, in PPC, it's better to find your own register convention. For example the following registers are used for:

`sp` - obviously the stack pointer (r1).

`rtoC` - the toc pointer (r2)

`r31` - modify this at your peril if you call any 68K code or an OS function it's best to leave this alone.

Apart from the above registers, all the other are free. However, when passing parameters to a System function (or trap), the parameters are generally passed in r3 to r10, and the results passed back similarly in r3.

For more information we suggest you check with Inside Macintosh PowerPC System Software, but generally the above holds true. Fantasm's reference manual gives you detailed register volatility rules. This is handy when determining if calling an OS function will destroy a register.

One method is to use the low numbered registers as scratch registers, and the high numbered registers as longer term storage. For example, we use r29 to store the LR in when going to a subroutine. If that subsequently calls another routine, that routine will either save r29 in memory first or use r28 to store LR in. The emphasis is on speed, and if you can get away without having to reference memory, then do it.

Stacks are implemented in software, using whatever method you prefer - however, the Update form of instructions are handy for this, as the NEW effective address is stored, not the previous effective address - e.g.

```
68K      move.l  d3,-(sp)
PPC
        stwu   r3,-4(sp)
68K      move.l  (sp)+,d3
PPC
        lwz   r3,(sp)
        addi  sp,sp,4
```

Note the lmw instruction. This moves registers from the processor to memory quickly:

```
        stmw   r3,0(fred)      *save r3 to r3 in memory at location fred
        lmw   r3,0(fred)      *load r3 to r31 from memory at fred
fred:
        ds.l   4*32           *reserve 128 bytes of storage in the data section
```

Instruction set

The instruction set is detailed in Fantasms reference manual - we will not replicate it here, but will note specific practices.

The "carry", "overflow", and "extended" option bits: The standard PowerPC arithmetic instructions do not set the carry bit or test for overflows. The "c" and "o" suffixes are used to designate the instruction forms which modify the carry and/or overflow bits, as in "add carrying" (addc), "add carrying with overflow enabled" (addco), and "add carrying with overflow enabled and CR update" (addco.). The "extended" arithmetic instructions include the carry bit in their calculation, to implement multiple-precision arithmetic. The extended instructions include "add extended" (adde), "subtract from extended" (subfe) and "add to zero extended" (addze).

The "record" bit: Unlike the 68K processor which nearly always set the condition codes depending on the outcome of an operation, in PowerPC we use special "record" versions of the arithmetic instructions to set Condition Register field 0 (CR0). Most arithmetic instructions have a "record" form indicated by appending a period (".") at the end of the instruction name, as in subtract from (subf.).

Immediate and "shifted immediate" values : Some of the instructions have an "immediate" form where one of the operands is contained within the instruction word. (This differs from the 68K where Immediate is an addressing mode and the information follows the instruction.) Since immediate data is limited to values that can fit within the instruction word, immediate values are usually limited to a 16-bit halfword. The PowerPC also supplies "immediate shifted" instruction forms that take a 16-bit immediate value and shifts it left by 16 bits into the upper half of a word, allowing the loading of fullword (32 bit) immediate data with two instructions - LIS and ORI for example.

A practical example.

In this section we'll dissect the "PPC_GRAPH_DEMO" program supplied on the Fantasm 5 CD. The main aim of this example is to show just how a PowerPC program is structured. We have 2 source files, 1 globally included file (the BSS offsets) and a Build Control File. The two source files are simply the main source file "PPC_graph_demo.s" and the initialisation file

"graph_demo_init.s".

The aim is to open a window, use QuickDraw to draw some nice shapes, and then quit.

The first thing we do is equate some registers to names, so as to make the code more readable:

```
param1: reg    r3      *Set up the names of the regs used for parameter passing
param2: reg    r4
param3:        reg    r5
param4: reg    r6
bss:    reg    r30     *The register we use for global data
```

These registers are used as parameter holders during system calls using the "Xcall" macro.

Next, the program proper starts:

ppc_graph_demo:

```
ENTRY *Prog starts here
start_up *save all the regs and set up r30 for global
la      r3,qd(`bss) *get the address of the QD array into r3
addic   r3,r3,206-4
Xcall   InitGraf      *Init managers
Xcall   InitFonts
Xcall   InitWindows
Xcall   InitMenus
Xcall   TEInit
li      `param1,0
Xcall   InitDialogs
Xcall   InitCursor
**Open our window and copy its viewrect.
bl      graph_demo_init *initialise and open a window and get its viewrect
                                *into viewrect_1(bss)
```

The ENTRY directive tells Fantasm that this is where the program starts .

Next we use the "start_up" macro to save the PowerPC registers, and set r30 to point to the BSS section. Following on is the normal Macintosh initialisation - we could have used a library function here (init_mac), but thought it better to show the process. Finally we branch and link to graph_demo_init which is in the initialisation source file. graph_demo_init carries out two functions. Firstly it opens our window, and secondly it sets up two rectangles that we will be drawing into:

```
**File:graph_demo_init.s
param1: reg    r3      *Set up the names of the regs used for parameter passing
param2: reg    r4
```

```

param3:      reg      r5
param4: reg      r6
bss:  reg      r30          *The register we use to point to "global"
variables

graph_demo_init:
    mflr      r29          *save return address
    bl        open_window
    mtlr      r29
    blr

open_window:
    mflr      r28          *save return address from link register
    li        `param1,128  *window resource id is 128
    li        `param2,0    *clear param2 - window storage - let the OS find
some
    li        `param3,-1   *behind no other windows(i.e. in front)
    Xcall     GetNewCWindow *Note NewCWindow, else we could have problems
                                *with the colors.

    stw      `param1,window_1_ptr(`bss)
**get the viewable rectangle (top,left,bottom,right)
    la       r3,16(r3)     *windowptr+16=viewrect (la is "load address")
    la       r4,viewrect_1(`bss) *Storage is in the bss section
    lfs      f0,(r3)       *32 bit move into f0
    stfs     f0,(r4)       *32 bit store into viewrect_1
    lfs      f1,4(r3)
    stfs     f1,4(r4)

**And copy to our second rectangle as well
    la       r4,viewrect_2(`bss)
    stfs     f0,(r4)       *into viewrect_2

    stfs     f1,4(r4)

**set the port to our window
    lwz      `param1,window_1_ptr(`bss)
    Xcall     SetPort
    lwz      r10,white(rtoc) *r10 points to colour white
    mtlr     r28          *restore the return address
    blr      *and branch to it

****

    global   graph_demo_init
    extern_data  white

```

Note the use of the FPU (stfs - store floating single) to transfer the 8 byte rectangle definition into viewrect_1 and 2.

Now we have a window and know its coordinates (viewrect) we can start drawing.

```

**First lets set the foreground colour to white
    lwz      `param1,white(rtoc)
    Xcall     RGBForeColor *That should do it

```

This piece of code sets the current pen colour to white. Now we fill the window with horizontal lines by drawing each line and decrementing r22 until it is zero:

```

**now a simple horiz test
line - draw r22 white lines
    la    r3,viewrect_1(`bss)    *top,left,bott,right
    lhz   r20,2(r3)              *left of window
    lhz   r21,6(r3)              *right of window
    lhz   r22,4(r3)              *bottom of window into r22
**use MoveTo and LineTo to draw the line
line_loop:
    bl    draw_line              *draw a line(r20 to r21 at
y position r22)
    subic. r22,r22,1             *up 1 line
    bne   line_loop              *and if line y isn't zero draw another line.

```

We then repeat the process, but this time we change the current pen colour as well and do it 50 times:

```

**now we'll do the same, but change the colours dynamically this time and
**fill the window 50 times
    li    r26,50                 *do it all 50 times
outer_loop:
**reset the x and y's
    la    r3,viewrect_1(`bss)    *top,left,bott,right
    lhz   r20,2(r3)              *left of window
    lhz   r21,6(r3)              *right of window
    lhz   r22,4(r3)              *bottom
**Draw line and alter the components of the colour
line_loop_2:
    bl    draw_line              *draw this line
    lwz   r23,white1(rtoc)        *r23 points to our colour that we are altering
    lhz   r24,(r23)              *get the red value
    subic. r24,r24,64             *subtract 64 from the red
    sth   r24,(r23)              *save the new colour back in memory
    lhz   r24,2(r23)             *get the green value
    subic. r24,r24,32            *subtract 32 from the green
    sth   r24,2(r23)             *save the new colour back in memory
    lhz   r24,4(r23)             *get the blue value
    subic. r24,r24,128           *subtract 128 from the blue
    sth   r24,4(r23)            *save the new colour back in memory

    lwz   `param1,white1(rtoc)
    Xcall RGBForeColor           *Set new foreground colour to white1
    subic. r22,r22,1             *up 1 line
    bne   line_loop_2           *and if not top of window (line=0) draw next line in new
                                *colour.

    subic. r26,r26,1             *do it all r26 times
    bne   outer_loop

```

Next we go from drawing lines, to a little scroll test. Rather than put the code in line, we have used a scroll routine called:

```

    bl    clear_window           *clear the window out by scrolling

```

This looks like this:

****Clears our window by first scrolling diagonally, and then virtically (virtically?).**

```
clear_window:
    mflr    r29                *save return address in r29
**First lets set the foreground colour to white
    lwz     `param1,white(rtoc)
    Xcall   RGBForeColor      *That should do it
    la     r20,viewrect_1(`bss) *r20 points to viewrect_1
    lhz    r22,4(r20)         *bottom of rect for use as a loop count
**Scroll diagonally
*extern pascal void ScrollRect(const Rect
*r, short dh, short dv, RgnHandle
*updateRgn)
scroll_diag_loop:
    la     `param1,viewrect_1(`bss) *top,left,bott,right
    li     `param2,1             *dh = 1 = scroll horizontal +1
    li     `param3,1             *dv = 1 -- scroll vertical by 1 as well=diagonal scroll
    li     `param4,0             *no updaterng
    Xcall   ScrollRect          *scroll by 1 pixels
    subic. r22,r22,1             *Decrement loop count
    bne    scroll_diag_loop     *and branch if not zero to scroll again
    la     r20,viewrect_1(`bss)
    lhz    r22,4(r20)          *bottom
**
Now Scroll down
scroll_down_loop:
    la     `param1,viewrect_1(`bss) *top,left,bott,right
    li     `param2,0             *no dh this time
    li     `param3,1             *just dv
    li     `param4,0             *no updaterng
    Xcall   ScrollRect          *scroll by 2 pixels
    subic. r22,r22,1             *Decrement loop count
    bne    scroll_down_loop     *and branch if not zero to scroll again
    mtlr   r29                  *get return address in link register
    blr    *and return to caller.
```

Note the way the Pascal header translates to PPC - very easily. The parameters go left to right into r3 onwards (up to and including a maximum of r10).

When clear_window returns, the window will be cleared and we can draw some ever larger circles just as easily:

```
**now lets draw a circles
    li     r28,3                *do the zoomy circles 3 times
rgb_zooms:
    lwz     `param1,red(rtoc)
    Xcall   RGBForeColor      *Set new foreground colour
    bl     draw_circles       *draw a zoomy circle in red.
    lwz     `param1,green(rtoc)
    Xcall   RGBForeColor      *Set new foreground colour
    bl     draw_circles       *draw a zoomy circle in green
    lwz     `param1,blue(rtoc)
```

```
Xcall  RGBForeColor    *Set new foreground colour
bl     draw_circles    *draw a zoomy circle in blue
subic. r28,r28,1
bne    rgb_zooms
```

PowerPC Assembly Language Beginners Guide

Chapter 5

The Macintosh

This is the trickiest chapter by far. I will attempt to explain just what a Macintosh is (from a programmers perspective), how it is programmed from a low level point of view and where some potential problems can occur.

The Macintosh, or more lovingly, the Mac, debuted way back round about 1984 (which is about two centuries in computing time). At the time it was heralded as a complete and utter breakthrough. Today, it can be classified as a high powered, multi-media workhorse. Some would argue it is not a multi-tasking system, but from a programmers point of view it is very multi-tasking. The kind of multi-tasking I am talking about is that of being able to run multiple processes sharing a finite amount of hardware resources. For this very reason, even a game can't just ditch the OS and get on with it. For example; you may grab a serial port and start sending data.

But if another process running in the background asks for use of the same serial port, the OS has no way of knowing you have grabbed it. The result is garbled data. With the advent of real time systems such as Open Transport Networking and multithreaded processes, even an assembly language programmer has to follow the rules. Granted that an assembly language programmer can do things a C programmer simply can't. Even in this day and age, certain parts of the OS can only be talked to with some assembly language "glue". Of course, the assembly language programmer doesn't need this glue, and so gets faster execution.

For the reasons outlined above, you need to know how to talk to to the OS, and what facilities it provides. Fortunately under PowerPC, accessing the OS is far simpler than it is for 68K, where you normally push parameters on the stack, but not always. A case in point is the memory manager when called from 68K code. It generally takes a pointer in a0, and any size data in d0.

In PPC, all parameters are passed in registers. The first integer parameter goes in r3, the next in r4 etc. Floating point parameters are passed in f1 onwards. Any budding Mac programmer NEEDS to get hold of Inside Mac Toolbox essentials and Overview. These are totally necessary reading. Believe me, if you break the rules on the Mac, even though your program may run, your users will give you a very hard time :-)

Good, that's the OS introduction out of the way. From the above you will hopefully see why an understanding of the OS is necessary. So what is the MacOS? As noted previously, the Mac has a long and somewhat uneven heritage.

Many projects have been started at Apple and only half heartedly incorporated into the OS (PowerTalk, GX printing etc.). Many other projects have been highly succesful and become "core" - for example the Sound Manager. Thus we have an OS with many "core" components and

many "not so core" items. This really doesn't matter to us. What technologies you decide to use is up to you. The important thing is that you do need to use some of the OS. Why? There have been many different motherboards, processors and chips used in the Mac over the years. It's not like the Amiga with a standard chipset - you simply don't know what motherboard your pride and joy will end up running on. This may not sound like a whole load of fun, but don't worry - you need to use the OS yes, but you don't need to do everything through it. I know you may be thinking calling the OS is slow because of all those parameters you have to pass and set up. True. The trick is to use the OS just where you really need to, or at least to use the OS to get the machine into the state you need it. Just bear in mind that it is a multitasking environment and all will be fine.

Now to make this easier for all of us, I'm going to show you two ways of calling the OS, and from then on, use the second. First the underlying theory.

On the PowerMac, all OS functions are exported from fragments (typically shared libraries). This means that your PowerPC program needs to be linked (at run time) with the OS exports. Then to call an OS function, your code has to set up the TOC to that of the shared library, find the address of the function, and then branch to it. Of course, it also has to set up a stack frame, and save some important registers (like our toc for example!). Here's the base, raw code to do it:

```

lwz    r12,the_function(rtoc)  *load transition vector
stw    R2,20(sp)               *save my RTOC
lwz    r0,0(r12)               *get callee address
mtctr  r0                      *prepare branch
lwz    R2,4(r12)               *set callee RTOC
bctrl                      *bsr to callee

lwz    r2,20(sp)               *get my toc back

```

Replace the text "the_function" with whatever OS function you are calling. The name of the function must have previously been imported via Fantasm's import directive. So, let's take BlockMove as an example - the code to call BlockMove (moves a chunk of memory) would be:

```

ifnd   BlockMove               ;if BlockMove hasn't been defined
import BlockMove               ;import it
endif                                     ;of import check
lwz    r12,BlockMove(rtoc)     *load transition vector
stw    R2,20(sp)               *save my RTOC
lwz    r0,0(r12)               *get callee address
mtctr  r0                      *prepare branch
lwz    R2,4(r12)               *set callee RTOC
bctrl                      *bsr to callee

lwz    r2,20(sp)               *get my toc back

```

BlockMove is defined as:

```
BlockMove(srcPtr, destPtr, byteCount);
```

So, we move our srcptr into r3 (where the data is now), we load r4 with our destptr (where we want the data copied to) and we load r5 with the number of bytes to copy. (As you can see, BlockMove would be better called "BlockCopy" but there you go...).

So, if we wanted to move 1000 bytes from fred to harry the complete code would be:

```

lwz    r3,fred(rtoc)    *ptr to fred
lwz    r4,harry(rtoc)  *ptr to harry
li     r5,1000          *1000 bytes to copy
ifnd   BlockMove       ;if BlockMove hasn't been defined
import BlockMove       ;import it
endif                          ;of import check
lwz    r12,BlockMove(rtoc) *load transition vector
stw    R2,20(sp)        *save my RTOC
lwz    r0,0(r12)        *get address of BlockMove
mtctr  r0               *prepare branch
lwz    R2,4(r12)        *set BlockMove's RTOC
bctrl                          *branch and link to BlockMove

lwz    r2,20(sp)        *get my toc back
carry on with your code
fred:  ds.b    1000
harry:  ds.b    1000

```

As you can see, that's a whole lot of code to type every time we want to call something in the OS! It's also very error prone. So we don't do it that way. You can if you want, but it's not recommended - I just wanted to show you the mechanics.

We roll all the common code into a macro, and use that instead:-

```

lwz    r3,fred(rtoc)    *ptr to fred
lwz    r4,harry(rtoc)  *ptr to harry
li     r5,1000          *1000 bytes to copy
Xcall  BlockMove

```

Easier? Course it is. The only requisite is that you must include the right .def file into your project either as a globinc, or include the file into your source file with an includeh directive. If you look in the Anvil folder "Anvil low level defs" you'll find lots of these files. How do you find the right file? Easy; use Anvil's search all files in folder feature. Open any file from the low level defs folder with Anvil. Now bring up the find dialog and type in "blockmove" in the find field, next click the "Search all files in folder" check box and then click OK. Within a few seconds, Anvil will have found the file that contains BlockMove; in this case, "memory.def". Def files are Fantasm's low level equivalent of C's header files.

Fantasm 5.1 expands on this concept by removing the need to include the low level def file through the use of zillions of macros - one for each OS function. In the above case, blockmove would be called as:

```

lwz    r3,fred(rtoc)    *ptr to fred
lwz    r4,harry(rtoc)  *ptr to harry
li     r5,1000          *1000 bytes to copy
OSBlockMove    r3,r4,r5

```

It doesn't look much different right? True, but what you get is error checking in that the macro knows how many parameters to expect, and so can fail if you pass not enough or too many parameters. Also it means you don't have to get the parameters in the right registers

```
OSBlockMove r3,r7,r8
```

Would be perfectly acceptable. The macros also provide the same facilities under 68K assembly language too, so you may want to take a look at this when 5.1 is published.

Note about BlockMove - if you only have a small amount of data to copy, do it yourself. By the time you've called BlockMove, you could've done it already. For large amounts of data, BlockMove is hard to beat.

Now we know how to call the OS. Good. You are now asking yourself "I wonder how many functions are in the OS?" The answer is "Thousands" and I can pre-empt your next question "So how do I find out which one I need?" with the following answer.

Download Inside Mac for starters. They are all free from Apple (go to our links page where you'll find links to them). At the back of each chapter is a list of functions described in that chapter along with the parameters.

Alternatively, all good booksellers will sell you the volumes in paper form or you can get them all on CD from Apple. If you are really serious about it you may want to buy either Think Reference from MacTech (which is the one we use) or Macintosh Toolbox Assistant from Apple (\$89 last time I checked).

You just type in the first couple of letters of the function you are interested in and it'll find the function for you. You'll get a description along with things to watch for and of course the required parameters and any return data.

So, what kind of OS functions are you likely to need? Well first off, any application HAS to do certain initialization, otherwise you'll call an OS function and it'll crash. Luckily, to save you time, a library function called InitMac is supplied. Call this at startup and you have no problems.

```
bl init_mac
```

Make sure you have added the Application library to your project, else it won't link (it'll fail with a "Where is this InitMac??" error (or words to that effect)).

After this, you are in a position to get on with you program. Nearly every type of program must have a window to draw into. You can't just draw "anywhere". You must draw into a graphic port. More often than not these days, you want to use color, so it needs to be a color graphic port - a CGrafPort. Windows can be created in one of two ways - either manually or you can get one from a resource. Either way is fine - you end up with the same result. From a resource is the quickest and least code intensive of the two. If you open Resedit, create a new file and then create a "WIND" resource, you'll find you can edit your window graphically. Be sure the initially visible checkbox is set and leave the ID at 128 for the purposes of this example. If you now save the file and then add it to your Anvil project; the window you have defined will be copied to your application when built.

The function we need to use to get the window is called GetNewCWindow and looks like this:

```
pCWindow = GetNewCWindow(windowID, wStorage, behind); windowID is the resource ID of the window. 128 in our case.
```

wStorage. You can tell the window manager where to put the window record, or if you set it to null (0) the storage will be allocated for you.

behind is the windowptr (what this call returns) of the window you want this window to appear behind. If you set it to -1, the window is placed at the front.

Now we can load the window with the following code:

```
li      r3,128  *the window ID in the resource fork
li      r4,0    *Let the OS allocate storage for it.
li      r5,-1   *We want it to the front
Xcall   GetNewCWindow
```

After this code, r3 will either contain a valid pointer to the window or zero if the call failed for some reason. You should then store the ptr to the window somewhere safe, such as in a global variable. Note that this call gets the window, but does not set the port. So the next thing we need to do is set the current port.

```
cmpwi   r3,0
stw     r3,my_window_ptr(`bss)
beq     error
```

Note the optimization of storing the window_ptr during the check for a valid pointer. This means that we can store a zero as the ptr (if there was an error), but heck, if we have an error anyway it doesn't really matter. The ptr to the window actually points to the graphics port (CGrafPort) for this window, so if we pass that to SetPort we then have a valid drawing environment:

```
lwz     r3,my_window_ptr(`bss)
Xcall   SetPort
```

And to prove it, we can now print something:

```
lwz     r3,my_string(rtoc)      ;a pascal type string - pstring directive
Xcall   DrawString
```

Finally, we can wait for the mouse button to be pressed before quitting:

```
wait:   Xcall   Button
        cmpwi   r3,0
        beq     wait
```

And to quit, we call the macro "tidy_up" (assuming "start_up" was called at the start).

```
error:  tidy_up
```

If you were to run this program, you'd get a window, and it would quit when you pressed the mouse button, but you wouldn't see any text. Why? Well, we haven't told the OS where we want to draw, so it does it at 0,0 (x,y coords). Because text is printed from the bottom up, our text is drawn out of view at the top of the window. We need to move the drawing coordinates (or the pen position) to a suitable location before drawing:

```
li      r3,4    *X coordinate
li      r4,20   *Y coordinate
Xcall   MoveTo
```

Now, we will see the text.

Looping

One of the things you need to do in any programming language is change the flow of instructions depending on the result of an operation. Consider the change to our program below:

```

**lets set up our x and y coordinate variables
    li    r23,4  *x
    li    r24,20 *y
**lets set up our loop counter
    li    r25,10
**lets draw some text
draw_loop:
    mr    r3,r23  *x
    mr    r4,r24  *y
    Xcall MoveTo  *move the pen
    lwz   r3,my_text(rtoc)
    Xcall DrawString  *Draw the string
    subic. r25,r25,1  *decrement loop counter
    addi   r23,r23,4  *increment x coordinate
    addi   r24,r24,4  *increment y coordinate
    bne   draw_loop  *if our loop counter isn't zero, goto draw_loop

```

Can you visualize the result of this? Whilst thinking about it, note again, I have placed two instructions at the bottom between the subtract with record instruction (subic.) and the conditional branch (bne). The adds in-between these two instructions are effectively "free". The processor would take five cycles to determine whether to take the branch or not, so by opting for a subic. rather than using the counter register (which may at first seem the obvious choice) to control the loop, I have incremented my printing coords "for free". Any integer instruction with a dot after it will affect the condition flags field 0. Any conditional branch without a cr field assumes cr0. We could write bne cr0,draw_loop and it would mean the same thing.

Note that a straight subi can't be used to set the cr0 field of the condition code register. One must use a subic.

Anyway, back to the program - it prints ten strings, each offset in x and y slightly. OK, cool. Now how about we want it continually printing these strings until we press the mouse button. What we need to do is: after printing the strings, erase them, check the mouse, and if not pressed reprint them, erase them, check the mouse, etc.

How do we erase them? Well the obvious choice is the OS function EraseRect.

It takes a rectangle defined as four 16 bit values of top, left, bottom, right. This definition is pretty much a Mac standard as far as rectangles go, so you may as well get it into your head now. Top, left, bottom, right.

You just need to learn it.

Now we could say, OK, I know the size of that window so I can define the rectangle to erase as a constant set of data with a dc.h directive. Wrong :-). Yes, you can do this, but what happens if the user changes the size of your window? If you remember back a few paragraphs,

I said that a windowptr is really just a CGrafPort pointer. If we look at the CGrafPort structure, we can find something that is useful. Actually, something that may be of benefit here is to examine just how we find out about a given structure. In this case we are talking graphics. This means that books like Inside Mac Memory probably won't help much. Inside Mac Toolbox Essentials might as it covers all the really important things, and a graphic ports are pretty important. So, I load up my Inside Mac CD (you can buy them on CD, or if you have a "burner" you can make your own IM CD after downloading them from Apple. Tip, just write them as a session, that way you can add them to your CD as they are published/updated. So, I load in my IM CD and open TB Essentials.

I go to the WindowManager section and find the definition of a CWindowRecord. Sure enough, the first entry is the graphics port, but this document doesn't expand on the graphics port structure. It does however tell me that the structure is defined in Inside Macintosh: Imaging. So, now I load that one up, goto the table of contents and immediately find the definition I'm looking for (actually I just type "cgrafp" into Think Reference :-)).

```
CGrafPort =
RECORD
    device:                Integer;                {device ID for font selection}
    portPixMap:            PixMapHandle;            {handle to PixMap record}
    portVersion:          Integer;                {highest 2 bits always set}
    grafVars:             Handle;                {handle to a GrafVars record}
    chExtra:              Integer;                {added width for non-space characters}
    pnLocHFract:          Integer;                {pen fraction}
    portRect:             Rect;                  {port rectangle}
    visRgn:               RgnHandle;              {visible region}
    clipRgn:              RgnHandle;              {clipping region}
    bkPixPat:             PixPatHandle;           {background pattern}
    rgbFgColor:           RGBColor;              {requested foreground color}
    rgbBkColor:           RGBColor;              {requested background color}
    pnLoc:                Point;                 {pen location}
    pnSize:               Point;                 {pen size}
    pnMode:               Integer;               {pattern mode}
    pnPixPat:             PixPatHandle;           {pen pattern}
    fillPixPat:           PixPatHandle;           {fill pattern}
    pnVis:                Integer;               {pen visibility}
    txFont:               Integer;               {font number for text}
    txFace:               Style;                 {text's font style}
    txMode:               Integer;               {source mode for text}
    txSize:               Integer;               {font size for text}
    spExtra:              Fixed;                 {added width for space characters}
    fgColor:              LongInt;               {actual foreground color}
    bkColor:              LongInt;               {actual background color}
    colrBit:              Integer;               {plane being drawn}
    patStretch:          Integer;               {used internally}
    picSave:              Handle;                {picture being saved, used internally}
    rgnSave:              Handle;                {region being saved, used internally}
    polySave:             Handle;                {polygon being saved, used internally}
    grafProcs:            CQDProcsPtr;           {low-level drawing routines}
END;
```

So, we can see that at offset 16 in the cgrafport is the rectangle (portRect) that defines the current size of the CGrafPort. Thus we should pass the address of this rectangle to the EraseRect function. Now, it doesn't matter what size the window is, we will always erase the whole visible part of it (assuming you haven't messed up the current clipping rectangle, but we haven't come to that yet)

That may sound complicated, but the code is trivial:

```
lwx    r3,my_window_ptr(`bss)  *our windowptr
addi   r3,r3,16                *point to the portrect
Xcall  EraseRect
```

See! This will erase the window. Here's a useful MacsBug tip in case you aren't aware of the power of MacsBug. It contains many templates for popular Mac data structures. So, if you have r3 pointing at a CGrafPort, in MacsBug you can type:

```
dm r3,cgrafport
```

And MacsBug will display the grafport with all it's field names and contents. "dm" is the MacsBug command to display memory. sm is the command the set memory.

Right, so now we can erase the window. All we need to do now is change the mouse button wait loop to branch back to the start of the drawing code rather than just waiting. Here is the program in total as it looks at this stage:

```

        includeh      windows.def
        includeh      quickdraw.def
        includeh      quickdrawtext.def
        includeh      events.def

bss:    reg    r30
chapter5:  entry
        start_up
        bl     init_mac
        li    r3,128  *the window ID in the resource fork
        li    r4,0    *Let the OS allocate storage for it.
        li    r5,-1  *We want it to the front
        Xcall GetNewCWindow
        cmpwi r3,0
        stw   r3,my_window_ptr(`bss)
        beq   error
        Xcall SetPort

mouse_loop:
**lets set up our x and y coordinate variables
        li    r23,4  *x
        li    r24,20 *y
**lets set up our loop counter
        li    r25,10
**lets draw some text
draw_loop:
        mr    r3,r23
        mr    r4,r24
        Xcall MoveTo
        lwz   r3,my_text(rtoc)
        Xcall DrawString
        subic. r25,r25,1
        addi  r23,r23,4
        addi  r24,r24,4
        bne  draw_loop
**Erase the window
        lwz   r3,my_window_ptr(`bss)
        addi  r3,r3,16      *point to the portrect
        Xcall EraseRect
        Xcall Button      *Check the mouse button
        cmpwi r3,0
        beq   mouse_loop
error:  tidy_up *end of program
*****
**Data
my_text:      pstring "123"  *The text we print
        align

```

```
**Linkage
    extern init_mac      *This is a static library function
```

Hopefully, by now you are getting into the swing of calling the OS. I'm not going to dwell on it too much longer. You just have to learn the various functions available. Obviously in the rest of this series we will be seeing more OS functions, but I won't be detailing them too heavily. The Mac OS is big with a capital B. You just have to learn as you go.

NOTE: We used EraseRect to clear the window. We could have used PaintRect to achieve the same result. The difference is that EraseRect is quicker than PaintRect.

Colors

The Mac when drawing uses two colors: The foreground color and the background color. EraseRect works with the background color. There are two OS calls to set these colors - RGBForeColor and RGBBackColor. These both take a pointer to three 16 bit unsigned values. Each 16 bit value specifies the red, green and blue intensity of the colour. So 0xffff,0xffff,0xffff is white, whilst 0x0,0x0,0x0 is black - red, green and blue. 0xffff,0,0 is max red, 0,0xffff,0 is max green.

Armed with this information we can start doing some funky color stuff. By altering the foreground and background colors we can alter the color of the text and the background of the window. Suppose we just incremented the value of the red component of the background color on each loop (and the other two components were set to zero). What would be the visual effect? Let's try it and see. We need to define some new data - the background color - we'll call it my_bg_color. Then we need some code after we erase the rectangle to change the background (bg) color:

```
**Change the bg color
    lwz     r3,my_bg_color(rtoc)

    lhz     r4,(r3) *get red
    addi    r4,r4,220 *add 100 to it
    sth     r4,(r3) *store red
    Xcall   RGBBackColor r3
```

Of course, the red slowly fades up to maximum red and then switches back to black. But what would be the result of the following code?

```
**Change the bg color
    lwz     r3,my_bg_color(rtoc)

    lhz     r4,(r3) *get red
    addi    r4,r4,220 *add to it
    sth     r4,(r3) *store red

    lhz     r4,2(r3) *get green
    subi    r4,r4,280
    sth     r4,2(r3) *Store green

    lhz     r4,4(r3) *get blue
    subi    r4,r4,180
```

```
sth    r4,4(r3)    *Store blue
Xcall  RGBBackColor  r3
```

Believe me, you don't want to work through too many steps. The end result is almost a random color fade and switch affect - but of course it isn't random. Best tried on a monitor capable of millions of colors.

And that's it for this chapter. The above code snippet is pretty unoptimized as far as PPC code goes - can you optimize it? Answer next chapter (tip: what if the color component we are loading to modify isn't in the level 1 cache and hence isn't immediately available?). Another question: Can you change the colour changing code so that abrupt color changes do not occur - it's all nice and smooth? Final question: How could we change the background color without erasing (in 256 indexed color mode only) - Tip check in the examples folder of your Fantasm 5 CD?

Don't worry if you find these questions baffling - we'll cover them all next time along with more application goodies such as menus and events with the emphasis being on fun.

Till then, Code On!

The project for this chapter can be downloaded from here (Fantasm 5 project)
- 8k.

Copyright Lightsoft 1998.

PowerPC Assembly Language Beginners Guide. Chapter 6

In this, and the next few chapters, we will be writing a larger Mac application in PPC assembly language. We will be making deliberate mistakes to highlight some easily made errors and problems.

Answers to questions posed in Chapter 5

In the last chapter I left you with some questions. The one people most found baffling was how to optimize the code that changes the three background colour components. Here is the original unoptimized code:

```
**Change the bg color
lwz    r3,my_bg_color(rtoc)

lhz    r4,(r3) *get red
addi   r4,r4,220    *add to it
sth    r4,(r3) *store red

lhz    r4,2(r3)    *get green
subi   r4,r4,280
sth    r4,2(r3)    *Store green
```

```

lhz    r4,4(r3)      *get blue
subi   r4,r4,180
sth    r4,4(r3)      *Store blue
Xcall  RGBBackColor  r3

```

The answer is given by Fantasm's Stall Warning Generator. If we switch it on for the red component calculation, we will get dependency warnings for r4 at the addi instruction. You can do this by modifying the code to read:

```

swg_med      *switch stall warning generator to medium sensitivity
lhz    r4,(r3) *get red
addi   r4,r4,220      *add to it (SWG Warning here)
sth    r4,(r3) *store red
swg_off     *swg off

```

Why is the SWG giving a warning? Because if the processor can't get at the contents of r3 (which is pointing to my_bg_color) on this clock cycle and put it in r4 as an unsigned 16 bit value (lhz) there will be a delay before the add can process because the add needs the contents of r3 as one of it's operands. i.e. the processor will stall.

If we bear in mind that the PPC can issue memory requests and then get on with other things, then we can prevent this stall by moving the add further down the instruction stream, so even if there is a delay getting the red component of the colour, by the time we need to data for the add it should be available.

Fine. So how do we move the processing (the add) further down the line? Does this work?

```

lhz    r4,(r3) *get red
nop
nop
nop
addi   r4,r4,220      *add to it (SWG Warning here)

```

Nope. All that does is introduce a three clock cycle delay which doesn't achieve anything (apart from wasting three clock cycles). What we need is useful processing inbetween the load and the calculation. How about we issue other memory requests whilst we're waiting for this data to arrive? Can we do that? It would be cool if we could. Well, is the PPC a powerful chip? Course it is. Have a look at this:

```

**Change the bg color
    lwz    r3,my_bg_color(rtoc)      *The colour we are changing

**get the red,green and blue components
    lhz    r4,(r3) *get red
    lhz    r5,2(r3)      *get green
    lhz    r6,4(r3)      *get blue

**now the processing
    addi   r4,r4,220      *add to it
    sth    r4,(r3) *store red

    subi   r5,r5,280
    sth    r5,2(r3)      *Store green

```

```

subi    r6,r6,180
sth     r6,4(r3)      *Store blue
Xcall   RGBBackColor *r3 points to the color
    
```

By rearranging the code and using two more registers we have eliminated any stalls that may occur if the data isn't immediately available. The highly astute among you may question a possible stall at the Xcall instruction because we are storing r6 in 4(r3) immediately before the call.

There will be no stall here. Why?

Two reasons. Firstly, the reading from the background color in the preceding code will have ensured the data is in the level 1 cache, so there will be no delay writing it. Secondly, Xcall will run through at least four instructions before the data is needed (would be five in compiled code because functions are normally called via a branch and link (bl), whereas Fantasm does it in-line and saves two instructions per OS call).

Just for your information, the possible setting for the SWG are as follows:

SWG_OFF	TURN STALL WARNING GENERATOR OFF (PPC)
SWG_LOW	STALL WARNING GENERATOR SENSITIVITY TO LOW. (PPC)
SWG_MED	STALL WARNING GENERATOR SENSITIVITY TO MED. (PPC)
SWG_HIGH	STALL WARNING GENERATOR SENSITIVITY TO HIGH. (PPC)

Please note that the SWG is inoperative in demo versions of Fantasm as the demos are distributed as 68K builds only. Due to the heavy workload required for the SWG to operate (it has to emulate the processor) it is coded in PPC asm.

The other question involved how to prevent abrupt colour changes. Well the answer to this one was simply avoid the dramatic change that occurs when a colour component switches from a large value (i.e. 0xffff) to a small value (i.e. 0) or the other way round. The last question was how to speed up the colour changing without having to fill the whole rectangle. The answer was to use Color Look Up Tables (CLUT's) in 256 color mode - an example of this is given on the Fant5 CD in the example "Clut Fade" so I won't reproduce it here.

Other matters

We need to point out that MacsBug versions prior to 6.5.4aX will simply not work on MacOS version 8.00 or later.

Macintosh Applications

The question most frequently asked by beginners is "How does a Macintosh Application fit together? How does it run?".

In common with most Graphical User Interface (GUI) operating systems, the Mac runs via "events". Every time the user does something such as clicking the mouse button or pressing a key, the OS generates an "event" and places it in the event queue. If the application in the foreground - the one the user is interacting with, examines this event queue it can find out what the user is doing by processing the events as they appear in the queue. It is a mistake to say the OS "sends" an event to the application - it doesn't, it simply places the events

in the queue. It is up to the application to get the events out of the queue. There are lots of different events that can be placed in the queue - Apple Events for example, but luckily there are just a few that "really matter".

These, with the possible exception of the diskette insertion events, are the major events you need to handle in order that your application will run and behave under the MacOS.

```

mouse button presses
mouse button releases
character key presses
character key releases
character key repeats
update requests
diskette insertions
activate window requests
    
```

The mouse and keyboard events should hopefully be self explanatory - you need to know when the mouse is clicked or held down and what keys the user is pressing.

The update requests are put in the queue when the OS thinks one of your windows needs to be redrawn (for example because another window has moved, or closed).

The activate event is placed in the queue when the OS thinks you need to activate (or deactivate) a window because the user has clicked in another window or application. Activating a window generally means showing any controls (scrollbars for example) associated with the window, activating any highlighting visible, activating the caret (if you have one) etc. Deactivating is the opposite - hiding everything (unlike Windows 95 which thinks it's fine to have scrollbars active on windows in the background!).

Whenever you pull an event out of the queue, the format of the event "record" always follows this structure:

Size Bytes	Name	Offset	Description
2	what	0	Type of event (0=null)
4	message	2	Depends on event type
4	when	6	Ticks since system startup (60ths of a second)
4	where	10	Mouse position in global coordinates (y/x)
2	modifiers	14	State of Apple, option, ctrl etc keys

Thus we can see that an event record is 16 bytes in size.

Event types (the what) are declared as follows:

nilEvent	0	no event matched eventMask
mouseDown	1	mouse button got pressed
mouseUp	2	mouse button got released
keyDown	3	character key was pressed
keyUp	4	character key was released
autoKey	5	key repeated because the user held it down
updateEvt	6	window must be redrawn
diskEvt	7	diskette was inserted
activateEvt	8	window was activated or deactivated
	9	(not used)
networkEvt	10	network event (obsolete in System 7)
driverEvt	11	I/O device driver event (system use)
app1Evt	12	available for application use.(obsolete)
app2Evt	13	available for application use (obsolete)
app3Evt	14	available for application use
app4Evt	15	used by <u>MultiFinder</u> / Switcher (obsolete)
osEvt	15	operating-system event (System 7)
kHighLevelEvent	23	high-level event (System 7)

Thus we can draw up a plan for a standard Mac application:

1. Initialise.
2. Process events.
3. Quit.

Simple huh?

WaitNextEvent

How do we get the events out of the queue? We use an OS function called WaitNextEvent. WaitNextEvent takes four parameters and returns a Boolean value. The boolean return value is true if there is an event for us or false if no event. In this case the type of event will be null.

The high level definition of WaitNextEvent is

<u>Boolean</u>	WaitNext Event (<i>eventMask, theEvent, sleep, mouseRgn</i>)
<u>short</u>	<i>eventMask</i> ; Event mask
<u>EventRecord</u>	* <i>theEvent</i> ; Event record
<u>long</u>	<i>sleep</i> ; Number of ticks to sleep
<u>RgnHandle</u>	<i>mouseRgn</i> ; Region of null mouse events
	returns Return Code; 0=null event; 1=event returned

eventMask is a 16 bit value specifying which events you are interested in - pass -1 to get all events.

theEvent is a pointer to a 16 byte eventrecord.

sleep is the maximum number of ticks you application agrees to let the OS (other applications) have control for (cooperative multitasking).

mouseRgn specifies a region inside of which mouse movement does not cause mouse moved events. Pass null if you don't want any mouse moved events.

If you remember back to chapter 5 we talked about calling OS functions - we could do it three different ways - the hard way for masochists, coding it all by hand, the easier way by

using Xcall and the easy and error proof way by using OScalls. In chapter 5 we used Xcall in our code. Now we will switch to using the OS calls found in the file Universal_OS_calls_plus.def. This new file is available from the Updates area. It isn't included in the demo distribution or on Fant5 CD's prior to 1st May 98. This is Fantasms definition of WaitNextEvent (taken out of Universal_OS_calls_plus.def):

```
OSWaitNextEvent:    macro    *mask,eventrecord,time,mouse rgn (all reg)
    if 68k
    clr.b    -(sp)
    move.w   \1,-(sp)    *mask
    move.l   \2,-(sp)    *event record
    move.l   \3,-(sp)    *sleep
    move.l   \4,-(sp)    *rgn
    dc.w    _WaitNextEvent
    move.b   (sp)+,\5
    else
    map_in_4   \1,\2,\3,\4
    SysCall WaitNextEvent
    cmpwi    r3,0
    map_out   \5
    endif
```

Using this method we can call WaitNextEvent as:

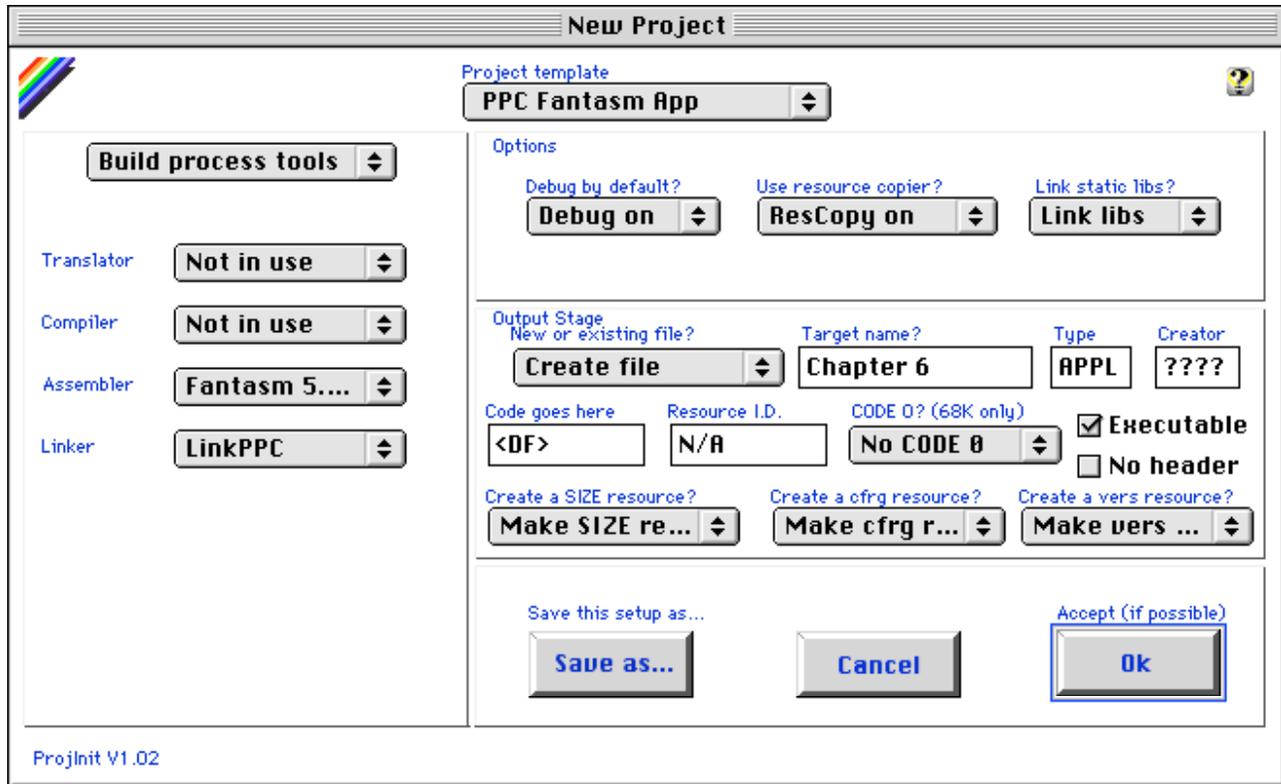
```
OSWaitNextEvent r3,r4,r5,r6,r3
```

This means we place the four arguments in r3-r6 and expect the boolean return value (either 0 or not 0) in r3. To be able to do this we need to make Universal_OS_calls.def a Globinc - this means add the file to our project and then move it into the Global Includes or _Globincs area of the project window (See below).

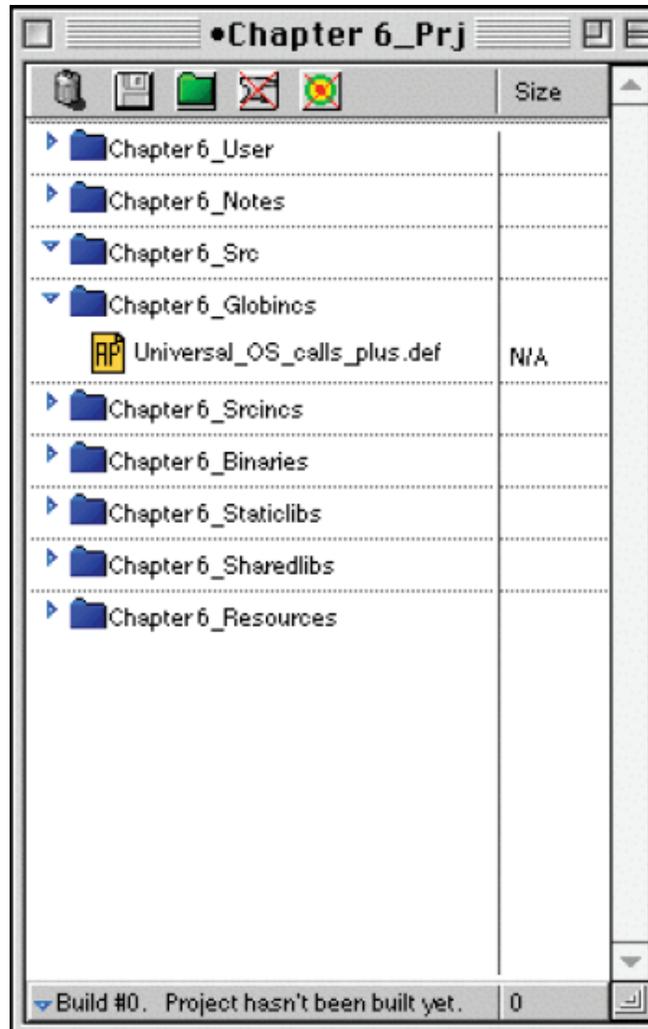
You may note that the definition of OSWaitNextEvent is a macro. Macros are a way of replacing one instruction (in this case OSWaitNextEvent) with many instructions. Macros can call other macros (in this case, the WaitNextEvent macro calls map_in_4 and map_out - these are other macros) and can take parameters, referenced as \1, \2, \3 etc. For more on macros please consult your documentation.

The Project

That gives us all the theory we need to get on with writing a Mac App. First, lets create the project. Run Anvil, and then from the Project Menu select "Create New Project". A dialog box like the one below will open (we've switched on the item labels by clicking the little help icon). From the Project template menu select PPC Fantasm App and change the file name to whatever you want - in our case we called it "Chapter 6".



Now click the OK button which will bring up a file selector box. Select (or make) a folder to create this project in and then save. A new project window will open in Anvil. Because we will be using `Universal_OS_calls_plus.def` we need to add it to the project and make it a globinc - this means `Universal_OS_calls_plus.def` will be included in every one of our source files automatically. To do this, click on the little disk icon at the top of the project window - looks like [Image]. From the file selector navigate to the folder called "Anvil Low Level Defs" and select `Universal_OS_Calls_Plus.def`. Anvil will place the file in the `_Src` area of the project. You need to drag it into the `_Globincs` area so the project window looks like this:



Now we can start writing our program. From Anvil's File menu select New. This will create a new source file in memory and open a window. If you haven't set Anvil's default language to PPC (via the General Preferences option) then you will need to set this file's language to PPC. Save the file as "Main.s". We will try to write this program as modular as possible, so our main file is going to call three routines: Init, events, terminate. This will develop into a reasonably sized program, so structure is all important here. It will seem as if we are generating a lot of files, with very little in them. This is true because we are architecting a large project by dummifying pieces of code. This is a good practice to get into.

For now we know we need to call at least three routines, so as well as our main file, we will also generate three other files; Init.s, Events.s and Term.s.

Our main file needs to look like this:

```

**Chapter 6 - example Mac App.
main:  entry
      bl  init
      bl  events
      bl  terminate
*****
      global  main
      extern  init,events,terminate

```

From that we can see the programs top level structure - it initialises, handles events and then quits. Notice that:

- a). All branches and labels are colored blue by Anvil (as a default - you can change the colors).
- b). We have declared main as being global and it's also the entry point for the program - this is where it starts running.
- c). We have declared init,events and terminate as extern. This means these items are external to this file. They can be found in other files.

Now we can create the other files - even though we have no idea what code will go in them yet, we can still create them and define some structure - init.s for example looks like this:

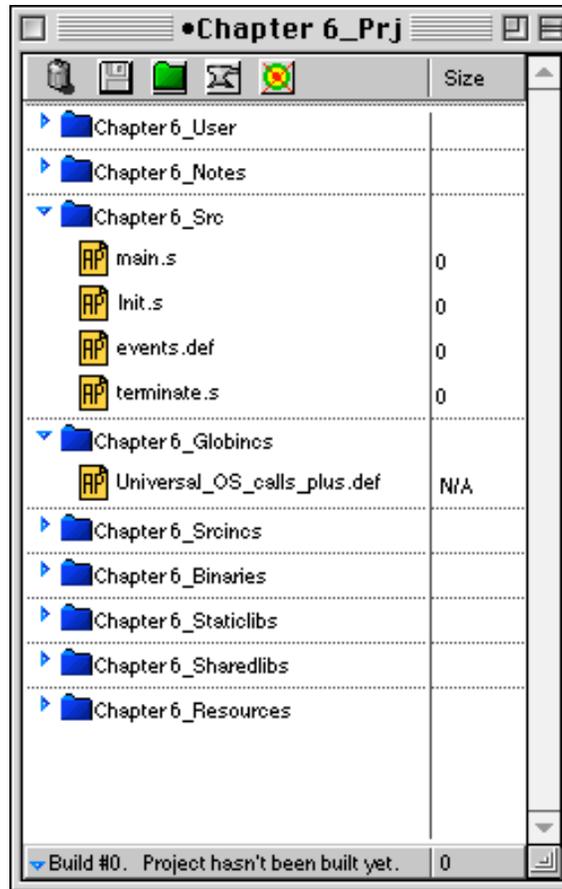
```

**Chapter 6 - Example Mac App
**Initilise
init:
 blr
*****
      global  init

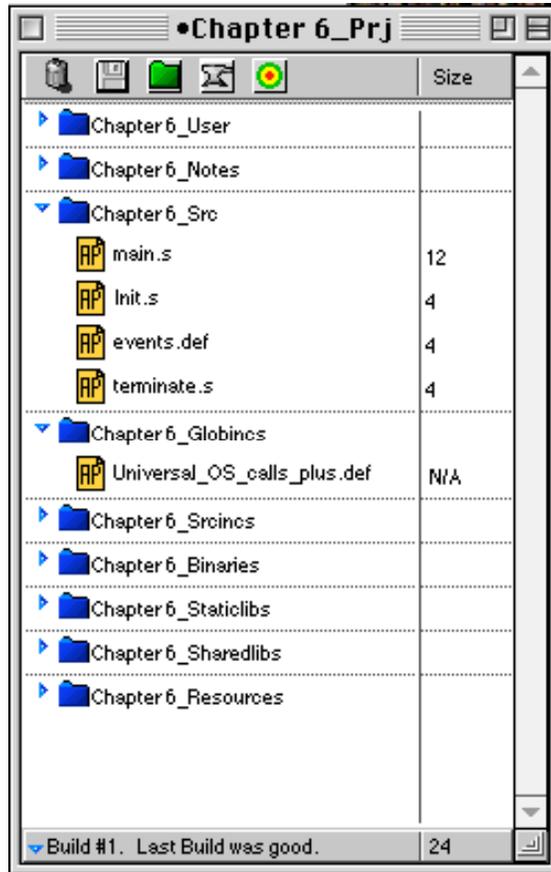
```

Do the same for Events.s and Term.s

Now we can add these files to our project. If you were "canny" you may have created a folder for your source code - this is a good idea. We add the files to the project the same way we added Universal_OS_Calls_Plus.def - by clicking the little disk icon in the project window. You should end up with a project looking like this:



And then if you build the project it'll look like this (DO NOT RUN IT AFTER BUILDING!
There is no terminate routine):



Now it's time to start writing some code. First we need to initialise the Macintosh as all good Mac apps must. Open the file `Init.s` and change it to call `init_mac`. This is a library routine that does all the initialising for us and saves some typing. There is a problem in that to call something, we normally use the link register (LR), and the link register is currently holding our return address to `main`! So we need to save it somewhere. How to save the Link Register? Here are a some solutions...

1. We have 32 integer registers available, so we'll save it in a register rather than in memory - lets say that `r27,28` and `29` will be link register save areas. We need to copy the link register into `r27`. We do this with a `mflr` instruction which means Move From Link Register, and we can restore it with a `mtlr` instruction - I wont insult your intelligence with the expansion of that mnemonic. The problem with that is you have to keep track of how many subroutines you have called, to be sure of saving the LR in the right register. Fast but difficult to maintain possibly.
2. The easiest way by far of saving the link register as you go into some code, and restoring it for returning is to use the macros `sub_in` and `out`. These work by saving the link register and restoring it for you. To be able to do this, you need the file `LS_PPC_Macros.def` as a Globinc in your project. These macros require three instructions to either save or restore the link register. You can use these routines anytime.
3. You can push it onto the stack, and then pop it off when needed - this takes two instructions for each push and pop to the link register - first you need to move the link register into a General Purpose Register (GPR) then you need to push the register onto the

stack (and the other way round to pop off the stack). To use this method you need to use `Universal_OS_Calls_Plus.def`.

4. If a routine doesn't call any other code, then there's no need to save the LR!

We will use the second method - it's slower, but as this is for beginners, we want to keep it simple. If you are quite happy with the other methods, then do it as it's faster!

To use `sub_in` and `sub_out` we need to make `LS_PPC_Macro.def` a globinc - this file contains lots of useful little macros to make PPC assembly language easier. Browse through it. Follow the procedure we used for `Universal_OS_calls.def` above.

The whole point of that discussion was to highlight that in assembly language, there are no rules. You can do things as you see fit and that you are comfortable with. Maintaining good structure however is paramount, irrespective of the language.

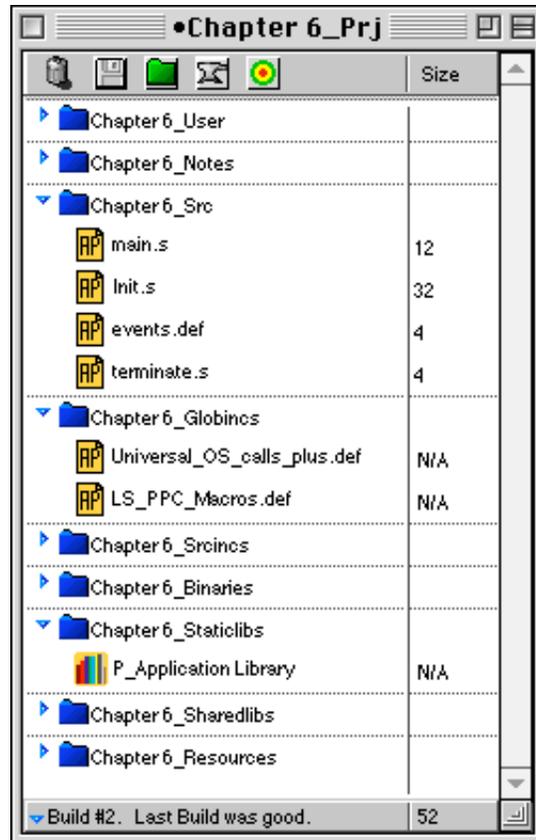
So, back to our initialisation. We have made `LS_PPC_Macros.def` a Globinc, and we've modified `Init.s` to call `init_mac`. `Init.s` should now look like this:

```

**Chapter 6 - Example Mac App
**Initiliasse
init:
    sub_in
    bl init_mac
    sub_out
*****
    global  init
    extern init_mac

```

If we now build this project, we will get errors from the linker. It'll say `Init.s` wants to link to `init_mac` but the reference doesn't exist. The linker is complaining because it can't find the code for `init_mac`.



init_mac is a library function, in this case held in the library called P_Application_Library. For the linker to find the function we need to add the library to our project. After adding the library, if you want to find out what functions it contains, just double click it out of the project window. Anvil will ask the Librarian to open the library and display it's contents. To get information about a function, click the function name in the Librarian's window.

A common mistake is to mis-spell a library functions' name (labels in PPC are case sensitive). Suppose we had added P_Application_Library to our project but the code read:

```
b1 Init_Mac
```

This would fail the same way as if the library was not present. The Linker would complain.

"Init_Mac" is NOT the same as "init_mac" in PPC because of case sensitivity.

So, we're initialised. It's easy to spot an App which has forgotten to call init_mac - it crashes a lot. Normally when you try to open a window, or call QuickDraw - it'll crash almost instantly. The last thing we need to do in this section is to be able to terminate correctly. The OS provides a function called ExitToShell. This will immediately terminate your application and is the recommended way to quit. We need to modify terminate.s to call ExitToShell as follows:

```

**Chapter 6 - Example Mac App
**Terminate
terminate:
    OSExitToShell
    blr    *this blr is never executed!
*****
    global terminate

```

All it will do is boot up and then quit, but, and this is the important bit, we have defined the structure of a useful project. One we can add pieces of code to, within a defined structure.

Now, lets take some time out to think about a typical Mac App. What are the elements of the interface?

The main elements include menus, windows and dialog boxes along with more minor details such as the mouse cursor and maybe even sound. Now what services does a typical Mac App require? Filing system services are probably way up on the agenda - the ability to read and save files is important. We need to look at all these things.

You may notice not once have we specified what this app does; for the reasons of this tutorial it is irrelevant.

In the next part we will expand our application to be able to handle simple events with specific emphasis on how to run the menus. We will look at initialising and dynamically changing menus to suit the current context along with acting on menu selections.

Postscript: Some are worried this series is going the wrong way - concentrating too much on the Mac OS and application world, rather than specific low level techniques such as writing to video memory etc. Please do not worry, we will come to that. The information is already out there in the example applications we provide with Fantasm. In the mean time if you do have any specific questions about this area, please direct them to our coding support

Copyright Lightsoft 1998.

PowerPC Assembly Language Beginners Guide

Chapter 7

This chapter will examine the creation of Menus, Dialog Boxes and Event Loops. It will also give a brief guide of how to use ResEdit (a utility for creating and manipulating Resources).

Overview of Chapter 6

If you can recall the last chapter, we started to create our first real Mac Application. Did you have any problems with that? Was there any mistakes in it? We stated that there would be errors in it, so did you spot them?

Did you notice that the macros `start_up` and `tidy_up` were missing?

Let us remind ourselves of these macros:

`start_up`

This macro saves the general purpose registers (GPRs) and sets up the stack and BSS pointer to `r30`.

`tidy_up`

This macro restores the GPRs and the stack.

So now lets modify the `main.s` file to encompass the above macros.

```
** Chapter 6 - an example Mac Application
```

```
main:
    start_up
    bl  init
    b   event
    bl  terminate
    tidy_up
*****
global  main
extern  init,event,terminate
```

Now that we are up to speed, how will we go about initialising the Mac App. We need to set up a few resources; we will go through each one, step by step. We will start by a setting up a window, then we will create a menu bar (using ResEdit) and tie them in to our program. Once that it complete we will set up an events loop and finally we will set up a Dialog and Alert box to demonstrate the difference.

So to summarise

- * Set up a Window
- * Create a Menu and Menu Bar
- * Implement the Menu Bar into the program
- * Create an Events Loop. We will set up Mouse Events to Select Options from the Menu Bar

- * Create a Dialog Box and an Alert Box
- * Implement these into the Program

Setting up a Window

We first saw this in Chapter 5. Now we will set up the window in the `init.s` file. We will use the `GetNewCWindow` call. Remember that we need to store the pointer to the window which is returned by `GetNewCWindow`. So in the `Chapter6_BSS.def` file we will reserve space for this variable

```
win_ptr_1:    rs.w    1
```

Then in `init.s` we create the window with the following code:

```

** Chapter 6 - Example Mac App
** Initialise

bss:  reg r30

init:
    sub_in
    bl  init_mac

***** Open Window *****
    li    r3,128    * The window identifier
    li    r4,0      * Let the OS allocate storage
    li    r5,-1     * Put the Window to the Front
    Xcall GetNewCWindow

    cmpwi r3,0     * Check the return pointer
    stw   r3,win_ptr_1(`bss) * store it
    beq   terminate * if fail then terminate
    lwz   r3,win_ptr_1(`bss) * program properly
    Xcall SetPort

```

Notice the `SetPort` call. This tells the program that the window you have just defined is going to be made active - i.e. any drawing commands are to be carried out in this port.

So far, so good ("so what!!" I hear you cry), now that we have set up the window, let's set up the menu bar.

Setting up a Menu Bar

Setting up the menu is separated into a few different sections. Firstly we have to define the MENU resource using ResEdit (Thank you Lord for small mercies). If you have never used ResEdit or heard of it, you have led a very sheltered life, but anyway ResEdit is a Visual tool for creating Resources. Lets make a menu and see how easy it is.

If you haven't got ResEdit then you will have to Add the Chapter6_menu resources to your project. Try and get a copy of ResEdit and use it, it is a VERY important tool.

Load ResEdit. Create a New File by selecting File, New... Select a file name and place to save the file. An empty window will then open, the title bar will be the name of the file you have just created. I know this is "sucking eggs", but I have to be sure we have covered every step.

Then select the Resource menu option. From there select the Create New Resource option. This will open a window which has a list of resource names. Notice that the name of the resources are like mnemonics. We will select a MENU resource.

This will open a window where it will show the user one 'tab' of a menu. The user will be given the option of entering a Title for the menu or Selecting the Apple Menu.

This is the first selection, so select the Apple Menu Symbol. The press <CR>. You will notice that it goes to the next line. So what we will do is create an option to show a Dialog Box (which will eventually be our about box) and another option to create an Alert Box, so that we can see the difference.

So in the Text field enter My Demo Alert Box and then press <CR>. It will go to the next line of the menu. Select the separator option to put a line underneath. Then close the window and that menu is set up.

So that is the Apple Menu done, we need to make a File menu next. We need to make another MENU resource (like we did before). Notice how there is now a window with a picture of the first menu with an identifying id value underneath which starts at 128 (well that's no surprise!!). The next MENU resource will have an id of 129, and so on if you made more MENU resources.

Instead of selecting the Apple Menu, the Title of this menu is File. Press <CR> to go to the next line. The option is NEW. So enter into the Text field New. This time though we also want a keyboard simulation of this option (AppleKey + N), so in the Cmd-Key: field enter N.

Enter some more options (what ever you want!!), but make sure you have an OPEN, a SAVE and a QUIT option in your menu.

Once you have done that close the window and save it.

It is now time to set up the menu.

How this is done is by using the GetMenu, InsertMenu and DrawMenuBar calls. The GetMenu command searches the open resource list and read the predefined menu from the resource and returns a handle. This handle is used to add the menu to the Menu Bar using the Insert Menu command. Finally, once all the menus have been inserted into the Menu Bar, it is drawn by the DrawMenuBar call.

Here are the definitions of each of the calls:

```
hMenu = GetMenu(menuID);  
    menuID;   resource ID of a 'MENU' resource  
    returns   handle of menu read from resource (0 = not found)
```

```
InsertMenu(theMenu, beforeID);  
    theMenu   handle of a menu to insert into the menu bar  
    beforeID  0 = normal menu, -1=pop-up menu
```

```
DrawMenuBar();  
    no parameters
```

So if we look at the code we will see how we get each menu (Apple Menu first), then Insert each menu (the left most menu first, so we have The Apple Menu first). Once we Insert each menu, the Menu bar is then drawn so that it appears on screen.

Note: The Apple Menu is a special beast, using the above method will only give you the Dialog box and Alert box in the Menu. To add all the other applications to the menu you must use AppendResMenu. This call is detailed below:

```
AppendResMenu(theMenu,MenuType);  
    theMenu   handle of a menu to be appended  
    MenuType  the type of menu to be added 'DRVr' or 'FONT'
```

The following code loads and displays our menu bar:

***** Setup Menu *****

```

li    r3,128          *Load in First menu (Apple Menu)
Xcall GetMenu
stw   r3,menu_ptr_1(`bss) * returns menu pointer

li    r3,129          * Load in Next Menu
Xcall GetMenu
stw   r3,menu_ptr_2(`bss) * returns menu pointer

lwz   r3,menu_ptr_1(`bss) * Add Menu to Menu Bar
li    r4,0
Xcall InsertMenu

lwz   r3,menu_ptr_1(`bss) *Add the rest of the Apple Menu
movei r4,"DRVR"
Xcall AppendResMenu

lwz   r3,menu_ptr_2(`bss) *Add the next menu to Menu Bar
li    r4,0
Xcall InsertMenu

Xcall DrawMenuBar      * Now draw the Menubar
                          * to screen

```

Right, that sets up the menu but it doesn't do a fat lot. Why?

The answer is fairly simple! Mac OS responds to EVENTS. Anything from pressing a key to clicking the mouse button. This was covered in Chapter 6. These events are in a queue and the OS acts upon the next one along using Event Handlers (please note this is a major simplification, this guide is showing you how to use EVENTS, not the internal workings of the MacOS). As an interesting note, Windows (I've just brought up my lunch) 95 works in a similar way, so crack events here and you can code Windows 95 apps (is that a bonus or a curse?) .

At the moment our application does not process any events so how can we open a menu, if we are not looking for a mouse click, or a menu selection etc.

Now Try imagining a Event routine, what should it do?

- * Get an Event
- * Compare the Event with an Event Type (ie. is it a mouse event?)
- * Go off and process that Event if you have routines to process that type of Event
- * Go back to the top of the Event Loop

Note: If you go off and process an event you must return to the top of the Event Loop afterwards.

That's fairly straight forward. The call that we use was introduced the the last chapter.

** Chapter 6 - an example Mac Application

```
main:
    start_up
    bl  init
    b   event
    bl  terminate
    tidy_up
*****
global  main
extern  init,event,terminate
```

So lets go to the code. We will firstly set up the parameters then call WaitNextEvent. If there is a null event returned we will return to the top of the Event Loop. If it returns an event we will process it, if it is a mouse event or an update event (these are the only events we will be covering in this lesson) else we will return to the top of the Event Loop.

* Chapter 6 Mac App

```
bss: reg r30
```

```
event:
```

```
***** Set up Events Loop *****
```

```

li    r3,-1          * send me all events
la    r4,event_record(`bss) * pointer to the event_record
movei r5,0xFFFFFFFF * allow processor to have system
li    r6,0           * set the mouse_rgn to no drag
Xcall WaitNextEvent

cmpwi r3,0          * if null event return to top of loop
beq   event

lhz   r25,event_record(`bss) * load event_record into register so it
                                * can be compared

cmpwi r25,mouse_event * if mouse event goto the mouse related
beq   mouse_routines  * routines . This is a mouse button down
                                * event

cmpwi r25,update_event * if update event goto update related
beq   update_routine  * routines

b     event          * else return to the top of the loop

```

As you can see the Events Loop is a fairly simple routine. But this still doesn't open our menu and let us select options from it, so now we have to process our mouse events.

There are a few different mouse events, like:

- * Mouse Button Down
- * Mouse Button Released
- * Mouse Moved

We are only concerned (in this example) with the mouse clicking on the menu bar and then selecting an option from the menu. So here is the plan of action. When we get a mousedown

event, find out where the mouse was pressed, was it on the menubar? This is done using the FindWindow call. If we clicked in the menu bar area we will jump to the menu routines.

Lets examine the FindWindow call:

```
iWinPart = FindWindow(thePoint, &whichWin);
                thePoint    global screen co-ordinates
                whichWin    receives pointer to window
                returns     code indicating window part
```

The code looks like this:

```
mouse_routines:
    lhz     r3,10+event_record(`bss) * load mouse co-ordinates (offset of 10)
                                           * in the event record
    lhz     r4,win_ptr_reply(`bss) * load pointer to window
    xcall   FindWindow

    cmpwi   r3,1
    beq     menu_routines *if location is menubar goto menu routines
```

Again a simple compare of the returned value to branch to the next section. Right, this is where it gets a bit more involved. Once you have clicked on the menu the program must do a few things first before it can process your selection of menu.

Firstly we have to discover what menu was selected, we will use the MenuSelect call to find this out. This call reads in the global mouse co-ordinates and so opens the menu that you have selected (if you clicked the menu bar) or selects the option from a menu that you clicked (if you clicked an option in a menu). Lets examine the MenuSelect call:

```
lMnuAndltn = MenuSelect(startPt);
                startPt    mouse position in global co-ordinates
                return     32-bit integer (HiWord = menu id,
                               LoWord is item no.)
```

Once we find out what selection has been made, we process the code for that option. If you look at the code below (the heading is MENU STUFF), you can see how we call the ExitToShell if the menu selection is Quit. Notice how we use the StandardGetFile call to open a standard 'Open File' dialog box. And yes, StandardPutFile opens a standard 'Save File' dialog.

```

void StandardGetFile(fileFilter, numTypes, typeList, &reply);
fileFilter    an optional file filter function (0 standard filter)
numTypes     number of file types to be displayed (-1 all files)
typeList     list of types to be displayed
&reply      address of a reply record

void StandardPutFile(prompt, defaultName, &reply);
prompt       the prompt message in the save box
defaultName  initial file name
&reply      points to the reply record

```

Right, I will let you mess about with the other standard calls, what we will look at now is how we call the Dialog Box and the Alert Box.

Calling an Alert Box

Calling an Alert Box is a single call. The Alert Box you display will have to be created in ResEdit. Examine the resource file for this project to look at an example Alert box.

To select the Alert Box we simply check to see what menu item has been selected from the Apple Menu and if it is not any of the other options then we process the Alert box routine. the call is called Alert. This is called then the code returns to the event loop.

```

dItem = Alert(alertID, filterProc);
alertID    ID of an 'ALRT' resource
filterProc address of custom event filter (0 = standard)
returns    dialog item number

```

Calling a Dialog Box

This is a more involved process, the program must firstly call GetNewDialog which creates a dialog from the specified DLOG resource. That only makes the dialog, the program must make the dialog appear using the ModalDialog call which begins user interaction with the dialog. Then we finally call DisposeDialog to close the dialog and release the related memory. Lets look at the commands:

```

theDialog = GetNewDialog(dlgRsrcID, dStorage, behind);
    dlgRsrcID  resource ID of DLOG resource
    dStorage   address of a dialog record NIL to allocate one
    behind    window plane -1 = in front
    returns   address of a Dialog Record

void ModalDialog(filterProc, &itemHit);
    filterProc  address of custom event filter NIL = standard
    &itemHit    receives number of selected item

void DisposDialog(theDialog);
    theDialog  the dialog window to close

```

Below is the code concerning the Dialog Boxes, Alert Boxes and Menu Selections. Read this carefully!!!

***** MENU STUFF *****

menu_routines:

```

lwz    r3,10+event_record(`bss)  * Offset 10 is where the global mouse
Xcall  MenuSelect                * mouse co-ordinates are

```

```

andi   r4,r3,0xffff             * get lower word for menu item
srwi   r3,r3,16                 * shift high word into r3 for menu id

```

* r3 contains menu res id

* r4 contains menu item number

```

cmpwi  r3,128                   * if menu id = 128 (base id) goto apple stuff
beq    apple_menu

```

```

cmpwi  r3,129                   * if menu id = 129 goto File menu stuff
beq    file_menu

```

```

b      event                    * any other menu, so return to event loop

```

```
*****
***** Apple Menu Stuff *****
```

```
apple_menu:
```

```
  cmpwi r4,2          * if 2nd option do dialog box
  beq   dialog_box
  cmpwi r4,1          * if NOT 1st option
  bne   more_apple_menu * go process any other apple
                          * menu selections
```

```
* Else do Alert Box
```

```
  li    r3,128        * ID of ALRT resource

  li    r4,0          * standard event filter
  Xcall Alert
  b     event         * return to event loop
```

```
dialog_box:
```

```
  li    r3,128        * id of DLOG resource
  li    r4,0          * allocate storage for Dialog Record
  li    r5,-1         * make dialog foremost window
  Xcall GetNewDialog
  stw   r3,mydialog(`bss) * store address of Dialog Record

  li    r3,0          * address of custom filter (0 = standard)
  la    r4,another_item_num(`bss) * receive number of selected
  Xcall ModalDialog   *(this gets trashed so be careful)

  lww   r3,mydialog(`bss) * address of dialog record
  Xcall DisposeDialog * kill that dialog

  b     event
```

more_apple_menu:

```
lwz    r3,menu_ptr_1(`bss)    * pointer to the window
; menu item is already in r4    * the item from the menu
la     r5,item_name(`bss)     * the item name as a pascal string
Xcall  GetMenuItemText
```

```
la     r3,item_name(`bss)     * the string of the item to call
Xcall  OpenDeskAcc
```

```
b      event                  * return to event loop
```

file_menu:

```
cmpwi  r4,2                  *File Menu Option 2
beq    menu_open
```

```
cmpwi  r4,4                  *File Menu Option 4
beq    menu_save
```

```
cmpwi  r4,6                  *File Menu Option 6
beq    menu_quit
```

```
b      event                  * back to event loop
```

***** File Menu Options *****

menu_new:

menu_open:

```
li     r3,0                  *file filter (no filter needed)
li     r4,-1                 *show all files
li     r5,0                  *list of types to display
la     r6,reply_record(`bss) *load address of reply_record
Xcall  StandardGetFile
```

```
b      event
```

```
menu_close:
```

```
menu_save:
```

```
lwz r3,prompt_string(rtoc) * load the prompt string
lwz r4,default_string(rtoc) * load the default name string
la r5,reply_record(`bss) * the address of the reply record
Xcall StandardPutFile
```

```
b event * retrun to event loop
```

```
menu_quit:
```

```
b terminate
```

```
***** DATA *****
```

```
default_string: pstring "Untitled.txt"
```

```
prompt_string: pstring "What do you want to call it?"
```

```
align
```

And Finally.....

If you look above you can see how we call to any external program through the Apple Menu. We firstly use the GetMenuItemText call to find the name of the program we wish to run, then we use OpenDeskAcc to run the program.

Lets examine the commands:

```
drvRefNum = OpenDeskAcc(daName);
           daName  p-string name of desk accessorie to open
           returns driver reference number if successful
```

This in itself will not make the program run, it will only run once our program becomes suspended (this is also an event) and this will only happen after we have processed an UPDATE event (I bet you thought I was never going to get to that didn't you). So lets look at the steps you need to go through to process an update event. Firstly we save the current GrafPort using GetPort, then we make our GrafPort the current one with SetPort. We then call for an Update of our window using BeginUpdate. We then redraw the bottom right corner of the window if it has been resized using DrawGrowIcon, next we update any visible controls using DrawControls. Then we end the update using EndUpdate and finally restore the current GrafPort.

There are a few calls there, but I will document only BeginUpdate and EndUpdate.

```

void BeginUpdate(theWindow);
    theWindow window to be updated

void EndUpdate(theWindow);
    theWindow window call by the previous BeginUpdate

```

This will then update the window and so run any call to execute external programs. Lets look at the code:

```

***** UPDATE EVENT *****
update_routine:
    lwx     r3,win_ptr_2(`bss)    * Save Current GrafPort
    Xcall   GetPort

    lwx     r3,win_ptr_1(`bss)    * Load the Window Pointer
    Xcall   SetPort

    lwx     r3,win_ptr_1(`bss)    * Start the Update of the Window
    Xcall   BeginUpdate          * we have selected

    lwx     r3,win_ptr_1(`bss)    * Updates the Sizing Icon in the
    Xcall   DrawGrowIcon         * corner of the window

    lwx     r3,win_ptr_1(`bss)    * Draws the Windows Active and
    Xcall   DrawControls         * visible controls

    lwx     r3,win_ptr_1(`bss)    * End the update
    Xcall   EndUpdate

    lwx     r3,win_ptr_2(`bss)    * Set the saved Grafport back to the
    Xcall   SetPort              * the active one.

    b       event

```

Take your time with this lesson, there is a lot of information here that needs to be learnt thoroughly. This has given us a base from which to make a Mac Application.

In the next Chapter we will look at other events, and get onto mouse cursors in different areas of our window (as we never got round to it in this chapter). We will look at stretching (resizing) windows and other events. We will decide exactly what sort of Application it will be eventually. We will add to the menus and modify the Help Menu.

Copyright Lightsoft 1998.

PowerPC Assembly Language Beginners Guide

Chapter 8

Basic Optimisations

This chapter moves away from application specifics and examines the processor and assembly language programming techniques in more detail. This section discusses a generic PowerPC processor.

The PowerPC architecture is described as "super scalar". This means in simple terms that the processor can execute more than one instruction at once. A very basic PowerPC processor will have at least one integer unit which handles the integer operations; addi for example, one floating point unit which deals with the floating point operations and one branch unit which, can you guess?, deals with branch instructions.

Up until now we have considered instructions coming from memory one at a time as needed for execution; this can be considered the *Instruction Stream*. However, this isn't how it happens in practice. The processor has an "instruction queue", abbreviated to IQ. Generally the instruction queue hold about 8 instructions waiting to be "despatched" for execution. As instructions are dispatched from the queue all the remaining instructions move towards the front of the queue and new instructions are fetched from memory.

Instructions can be dispatched from the first four locations in the queue (termed IQ0 to IQ3). On every clock cycle the processor tries to dispatch as many instructions as possible from the queue, so if the branch unit and integer unit are free but the floating point unit is not it would be possible for the processor to dispatch one integer and one branch instruction from IQ0 to IQ3 if those types of instructions are present.

This knowledge provides the assembly language programmer with a valuable optimisation technique; that of interleaving branch, floating and integer instructions to ensure an even mix of instruction types in the instruction queue (and indeed is why Anvil categorises instruction by group for syntax colouring purposes).

Generally instructions execute in one cycle but there are exceptions. Conditional branches can take anywhere from five down to zero cycles. If you remember back to the beginning we said that condition code flags were not implicitly set when data is moved or the result of a calculation become available. There's a very good reason for this, and it's not because the designers were lazy, or they just wanted to save silicon.

If you know anything about traditional CISC (Complex Instruction Set Computers) such as the 68000 family you may know that there is one set of condition flags; the flags typically indicate zero, carry, overflow etc. These are the flags that are tested when we execute a conditional branch. For example a subtract might be followed by a conditional branch if the result is zero (the zero flag is tested). The PowerPC processor being superscalar may actually try to execute a conditional branch *before* the flags it needs are ready (remember the dispatching discussion at the start of this chapter). The general rule of thumb is that a conditional branch needs to be placed about five instructions down from the instruction that affects the flags. This is possible because the programmer has to explicitly use instructions that change the flags. Using this technique the PowerPC processor can effectively process conditional branches in zero cycles! To further enhance performance instead of having just one set of flags we have eight; they are referred to as the cr flags.

This means conditional branches can be tagged with the specific set of flags they need to test. In Fantasm we identify the required set of flags with the identifier crx where x is a number between 0 and 7. If you don't specify a cr then Fantasm defaults to 0 for integer instructions and 1 for floating point instructions (Oh yes, the FPU supports its own conditional branches! fcmp). Typical usage would be:

```
bne cr3,fred whereas
bne fred
```

really means bne cr0,fred Having all these cr fields means we can do some funky stuff:

```
cmpwi cr0,r3,0
cmpwi cr1,r3,10 3 other instructions
ble cr0,fail
bgt cr1,fail
```

By doing the compares well before we need to determine if the data is in-bounds we effectively get the two conditional branches for free (or looking it the other way, we get the 3 other instructions for free). The condition flags can be set manually with the move to condition register instruction which can manually set flags in the cr register (naturally there's also a move from condition register instruction allowing you to read them). This can be used to great advantage to set cr flags well before they are needed without doing a compare - for example the MacOS BlockMove call is optimised in this way to calculate if the data is aligned and if not how many halves and bytes it needs to move before and after it can move words or double words (see FPU below). Very efficient.

So, by not implicitly setting flags the PowerPC designers gave us a very efficient method of dealing with conditional branches which always make up a large proportion of any program. Another optimisation the processor can make is speculative processing. This is looking down the instruction stream for conditional branches and trying to decide whether they will be taken. If so then the processor can start getting instructions from the new address (that which would be executed if the branch is taken). If it turns out that the processor was correct and the conditional branch is taken then the instructions are already to run. If however the processor is incorrect then all the new instructions have to be discarded and instructions fetched from the address following the conditional branch instruction. This may not seem such a good idea but later PowerPC processors have very efficient mechanisms for recovering if the processor predicted incorrectly. The programmer can help this speculative processing by giving a hint to the processor that the conditional branch will normally be taken. A typical use for this is in conditional branches at the end of a counting loop (wherever possible one should use the ctr register for this, but sometimes its not possible). Normally the branch will be taken back to the start of the loop so this would be a good place to provide a hint. Generally in Fantasm the "-" character is used if the branch is backwards and the "+" character if the conditional branch is forward (but see the Fantasm reference manual!). So:

```
loop: subic. r3,1 ;sets cr0 flags
      some instructions bne+ loop ;test the cr0 flags - specifically the zero flag
```

Is the correct way to optimise this loop (if you can't use the count register). Remember that the further up the instruction stream you can put the instruction that sets the cr flags, the faster it will run (with a ceiling of about five instructions).

Useful tip: Anvil's Help menu will contain language help for the current language in use. In assembly language it's very useful if you've forgotten the mnemonic for a certain

instruction. I for example can never remember some of the more useful rotates; so a quick search for what I want in the Help window generally provides the answer.

FPU

The PowerPC processor also benefits from having a Floating Point Unit (FPU) bestowed upon it as part of the architecture specification. This means that all PowerPC processors MUST have a floating point unit and as such programmers can rely on it being there. You may think of the FPU as being just a number cruncher where in fact it's also a great mover of data. Most PowerPC Macs have 64 bit wide data busses. The integer unit caters for integer sized data and as such its registers are 32 bits wide. The FPU on the other hand caters for floating point sized data and as such its registers are 64 bits wide (it can work with both 32 bit, or single sized data, or 64 bit, or double sized data). Note the correlation between the bus width of 64 bits and the FPU register width of 64 bits.

Thus the largest amount of data we can move with the integer unit is 32 bits, or four bytes, but with the FPU we can move 64 bits or 8 bytes. The FPU provides instructions for both reading and writing data from memory and luckily provides both 32 bit and 64 bit derivatives of nearly all of its instructions. In Fantasm the size of the data to be worked on is typically specified by appending either "s" or "d" onto the end of the instruction. For example:

```
stfs f0,(r5)
```

Stores the single sized data (32 bits) in f0 in the address pointed to by r5

```
stfd f0,(r5)
```

Stores the double sized data (64 bits) in f0 in the address pointed to by r5

It's a similar situation for loading data into an FPU register.

```
lfs f0,(r5)
```

Loads the single sized data (32 bits) at the address pointed to by r5 into floating point register 0.

```
lfd f0,(r5)
```

Loads the double sized data (64 bits) at the address pointed to by r5 into floating point register 0. Examine this loop:

```
line_loop: lfd  f0,(r6)
            addi   r6,r6,8 stfd  f0,8(r5) bdnz   line_loop
```

These four instructions copy data from the address at r6 to the address at r5. I've introduced another operation here. We've seen the bdnz instruction before. It decrements the count register and branches if it isn't equal to zero. The stfd is a variant of store floating double in that it automatically adds 8 to r5 and stores the result back in r5 before each execution. This means that in one instruction we store the data AND update the pointer by the size of the data; in this case it's a double so that's 8 bytes. You may realise that for this to work we need an extra instruction before executing the loop; the one that decrements r5 by 8 before we start! Thus the full code looks like:

```
subi   r5,r5,8 line_loop:
lfd    f0,(r6) addi   r6,r6,8 stfd  f0,8(r5) bdnz   line_loop
```

Alignment

Takes on a whole new meaning when you start moving 8 bytes at a time! If either your source or destination address are not octal (8 byte) aligned when you start moving data 8 bytes at a time the code will run slower than cold treacle. The PowerPC processor rarely has hardware support for mis-aligned addressing and so the OS takes over and has to move the data as best it can (this means slowly). Octal alignment means that the address you are reading from or writing to produces a remainder of zero when divided by 8. Everybody should deliberately try misaligning their data now and again just to get a feel for it; it's easily spottable once you seen it once!

*Copyright ©Lightsoft Software (Tools) 2000.
Reproduction in whole or part prohibited without permission.*

HEX AND SUCH

A Beginner's Guide to the Abyss of Hexadecimal Numbers

by ProZaq

Are you a "newbie"? As long as you're interested in not only computers but also in what's making computers work the way they do, then you'll definitely need to learn and master the meaning of a couple of basic expressions/ terms/ concepts. Take, for example, the hexadecimal number system; it doesn't matter if you want to learn the basics of programming or if you want to write programs for Macs or PC's or you just wanna cheat on some computer games; you have to learn and master it in order to be able to "exploit" it. And as you learn more you will notice that all these concepts are interrelated and one can be manipulated to change the other.

In this file I shall try to explain the following topics: binary and hexadecimal numbers, bytes/words/longs, ASCII characters, strings, HexEditors, the hardware components of a computer, and debuggers. If you find that you are not familiar with an expression take a look in the "The Computer's Hardware Components" chapter.

Binary, Decimal, and Hex Numbers

Oh boy! Where do I start? Well, at the very, very, very beginning...

If I remember my IT classes well, the whole fame about binary numbers and calculations with binary numbers goes to an English fellow named George Boole. He developed amongst others Boolean Algebra. Remember all those horrible hours you had to spend in algebra class learning formulas like: $a(b+c) = a*b + a*c$? Well you have him to thank for it. He also developed a type of logic where he used ones and zeros to represent the logical flow of an operation, which is the kind of logic that every personal computer chip uses today.

You know how everyone is always saying that computers are all about ones and zeros? Well that's because everything in computers narrows down to being a one or a zero (an electronic current or the lack of it).

But what on earth is the binary number system? Well, let's try to define the decimal number system first (the one we use in every day mathematics) since we're more familiar with it.

The decimal number system is based on the number 10. Twas the name "Decimal"; which means "tenth" in Latin (doesn't "mal" mean "multiply" in German?). You have the numbers zero through nine. When you start counting from zero up, you hit nine. And what happens when you hit ten? You reset the value of the rightmost column (set it to zero), and carry a one into the next column. At one hundred you reset the two rightmost columns and carry a one into the next one. And so on. So as you notice you carry numbers at the powers of ten. Like $10^1 = 10$ (^ means raised to the power), $10^2 = 100$, $10^3 = 1000$, $10^4 = 10\ 000$, $10^5 = 100\ 000$, $10^6 = 1\ 000\ 000$ etc.

Let's break the number "9876" into columns representing the numbers at which the carrying occurs. The "thousands", "hundreds", "tens", and "ones" column.

Thousands	Hundreds	Tens	Ones
(10^4)	(10^3)	(10^2)	(10^1)
9	8	7	6

As you might have noticed, in order to get the number nine thousand eight hundred and seventy six you multiply the value of each column with the appropriate multiple of ten then add the values together ($9 \cdot 10^4 + 8 \cdot 10^3 + 7 \cdot 10^2 + 6 \cdot 10^1$).

In the binary number system we only have two numbers to work with instead of ten as we had in decimal. One and zero. So this means, that instead of carrying numbers at the power of ten we carry numbers at the powers of two; namely: $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$ and $2^8 = 256$.

When dealing with binary a lot of times the value of all eight columns of numbers are shown even if it is zero. Makes the calculations easier. For example, one in binary has the value 1 but can also be written as 00000001.

Here is a little chart showing the numbers one to sixteen in binary:

Value of column:

126	64	32	16	8	4	2	1	= value of each column added up
0	0	0	0	0	0	0	0	= 0
0	0	0	0	0	0	0	1	= 1
0	0	0	0	0	0	1	0	= 2
0	0	0	0	0	0	1	1	= 3
0	0	0	0	0	1	0	0	= 4
0	0	0	0	0	1	0	1	= 5
0	0	0	0	0	1	1	0	= 6
0	0	0	0	0	1	1	1	= 7
0	0	0	0	1	0	0	0	= 8
0	0	0	0	1	0	0	1	= 9
0	0	0	0	1	0	1	0	= 10
0	0	0	0	1	0	1	1	= 11
0	0	0	0	1	1	0	0	= 12
0	0	0	0	1	1	0	1	= 13
0	0	0	0	1	1	1	0	= 14
0	0	0	0	1	1	1	1	= 15
0	0	0	1	0	0	0	0	= 16

Here's an other approach in trying to explain how binary works. Try adding up the values of the columns where there is a one. In ten for example (00001010) there is a one in the two's and the eight's column. Thus when these values are added together (two plus eight) we get

ten. The same goes for fifteen, there's a one in each column so, eight plus four, plus two, plus one equals fifteen.

OK, now we've reached the hexadecimal numbers. Well, for these suckers we carry at powers of sixteen. With other words we count from zero to fifteen before resetting the first column and increasing the next. The slight problem of only having ten numbers in our everyday number system is compensated by using six alphabetical letters to represent the numbers ten through fifteen. Thus the numbers used in the hexadecimal number system have the following notation:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

And once fifteen is reached, the next number (as always) is represented by resetting the first column and increasing the next. Meaning that sixteen in hex is "10".

If you have managed to get this far you've done a good job. And if you still have difficulties understanding what the different number systems are all about then I'll let you in on a big secret. Only a very few people convert between number systems in their head.

Most of us mortals rely on something called the "Scientific calculator". This makes life a lot simpler than trying to convert numbers in you head. I always use a calculator simply because it's just so much faster. I believe that if you know the principles behind the different number systems and you have access to a calculator that converts between these then you're set.

So now you know what different number systems are. But when it comes to writing them down some difficulties may arise. It's obviously easy to distinguish numbers represented in binary. Just to be on the safe side, however, it's a convention to put a "%" sign in front of binary numbers. On the other hand "123" can be a number represented in both hex and decimal form. If it's a decimal number it's simply one hundred twenty three. But if it's a hexadecimal number then it has the decimal value of 291, two hundred ninety one. Big difference there! So how do you distinguish between hex and decimal numbers? Well the most common way is to represent hex numbers by putting a dollar sign, "\$" in front of the number.

In the programming language C you represent decimal numbers using the "0x" prefix. In assembly language it is common practice to use the "#" sign when representing decimal numbers. I tend to be very lazy so when I want to represent decimal numbers I just don't bother using any signs, but for hex numbers I always use the "\$" sign. For example: #12345 (decimal) is \$3039 (hexadecimal); and \$ABCDEF (hexadecimal) is 11259375 (still decimal if no sign is used).

Through the course of this file I will use this method of notation. I might, however, refer to hexadecimal numbers without the \$ sign if I think that it's obvious what I mean.

Bytes, Words, and Longs

Now that you know what hex is, there is a need to discuss the length of a number. The length of numbers have a large part when it comes to writing programs. By using numbers with different lengths the programmer can manipulate data much more easily. Another benefit of numbers with different lengths is that numbers that are small can be stored in a small place in the memory instead of occupying an unnecessarily large one. This is not much of a problem now with the increase of of both RAM and HardDisk sizes, but back in the days of C-64's and

before, when programmers only had so much RAM to work with, it was very important whether a number took up 1 or 4 bytes.

Anyway, in assembly language for the 68k Macintosh processors we talk about bytes, words and longs. A byte is two digits long and is between 00 and FF (0 to 255 in dec). A word is 4 digits long and is between 00 00 and FF FF (0 to 65535 in dec). Finally a long is made up of 8 digits and is between 00 00 00 00 and FF FF FF FF (0 to 4294967295 dec).

With other words:

byte:	\$00	- \$FF	#0 - #255
word:	\$00 00	- \$FF FF	#0 - #65535
long:	\$00 00 00 00	- \$FF FF FF FF	#0 - #4294967295

As you can see a byte takes up one fourth of the memory a long does. This principle will be discussed further in the chapter dealing with HexEditors. I think it might be a good thing for you to learn how many digits a byte, a word and a long has. I will use these expressions later on. I chose to use these expressions (and not including floats and doubles) because I feel that even an experienced person can get far with only these three length-notations.

For those interested the programming language C uses the following expressions to refer to the length of numbers:

```
char          c = 'A'; // 1-byte long by definition (in C++).
short int    si= 1;   // minimum range +/-32767.
short        s = 2;   // short same as short int.
int          i = 3;   // minimum range +/-32767.
long int     li= 4;   // minimum range +/-2147483647.
long         l = 5;   // long same as long int.
float        f = 10.1; // min 6 digits (decimal) precision.
double       d = 11.2; // min 10 digits (decimal) precision.
long double  ld= 12.3;
```

```
unsigned char    uc; // unsigned integers can only store
unsigned short int usi; // positive numbers.
unsigned int     ui;
unsigned long int uli;
```

```
signed char      sc; // signed integers can store positive
signed short int ssi; // or negative numbers.
signed int       si2;
signed long int  sli;
```

(Information taken from "C Reference Card" by Argus Software Engineering)

ASCII Characters

With the arrival of networks reaching from one country to the other arose the problem of character mapping. When you push the letter "a" on your keyboard, the hardware components of the computer send a number value to the processor which represents the letter "a". But how on earth would a computer in Yugoslavia, configured to deal with the Yugoslavian alphabet, be able to interpret letter "ä" which is fairly common in the Swedish language. To eliminate the problem a new standard for keyboards, the American Standard Code for

Information Interchange (ASCII) was adopted in most places. What this means is that (in theory at least) all alphabetical characters will appear the same way no matter where you are in the world.

Unfortunately this only works in theory, since different keyboards have different mapping of different keys and have different ways of showing different letters etc... The good news is that just like you didn't have to know how to convert hex numbers in your head, it's enough that you know that ASCII refers to the numerical values of the different characters on your keyboard that the computer can interpret as such.

Now you know that when you push a key on the keyboard, the corresponding number value is sent to the processor (well in reality it's interpreted by the OS and sent to the active application). So, what is this number value? Well, every character on the keyboard is represented by a different number. For example the English lowercase alphabetical characters range from \$61 to \$7A (a-z). Notice that when it comes to computers there's a definite difference between lowercase and uppercase letters. Thus the uppercase English letters are represented by the numbers \$41 to \$5A (A-Z).

It is important to realize that every ASCII character (every character on the keyboard) can be represented by a number that's the size of a byte. Meaning a number between 1-255, \$1-\$FF. Thus the current standard of keyboard maps can only handle 255 characters.

But that's of no real importance either. The most common ASCII characters and their values in both hex and decimal form are available in the included file "ASCII.txt"

Now then, we know that ASCII characters are represented by numbers. For example the capital letter "A" is represented 65 (\$41). "B" is 66 (\$42) and "C" is 67 (\$43). So the letters "ABC" could be represented by the ASCII values 65 66 67 (or in hex 41 42 43). And this brings us to our next topic, strings.

Strings

The expression "string" refers to a sequence of keyboard characters. For example "Hello world!" would be a string. Notice that the computer doesn't care about the space between the two words, it looks upon the sentence as only one string of characters. This leads to the problem of representing strings. Imagine how a string would look like in the computer's point of view. It would be a sequence of numbers stored somewhere in the memory. And unless you inform the computer how to interpret the beginning or end of the string, it will not know where the string ends.

There are currently two standard ways of representing strings. The C way and the Pascal way. I'll start with the C way, it's easier. Basically after the last character in the string there is a zero-byte. This means that a value of zero marks the end of the string. For example:

H	E	L	L	O	_	W	O	R	L	D	!	•
72	69	76	76	79	95	87	79	82	76	68	33	00
\$48	\$45	\$4c	\$4c	\$4f	\$5f	\$57	\$4f	\$52	\$4c	\$44	\$21	\$00

Keeping in mind that the size of an ASCII character is that of a byte (max 255) we notice that using the C method the length of the string is actual increased by one byte; the zero-byte on the end. When a program is in need of using the above string, it needs to know the

memory address of the first character, and it knows that it has hit the end of the string when the value of the character is zero.

The Pascal method is a bit different. It stores the number of characters in the string as the first byte. The example above would be portrayed like this in Pascal notation:

```

•   H   E   L   L   O   _   W   O   R   L   D   !
12  72  69  76  76  79  95  87  79  82  76  68  33
$0c $48 $45 $4c $4c $4f $5f $57 $4f $52 $4c $44 $21
    
```

As you might have noticed there are 12 characters in the string (including the "_" and the "!" signs). So using the Pascal method, the program would read the first byte of the string and thus determine the length of it.

This whole concept will be developed further in the next chapter.

Hex Editors

NOTICE: When dealing with hex editors you are going to be changing real files on your computer. By changing just one byte in a file you can corrupt it to the extent that it will not be usable any more! So always make sure that you are working on a BACKUP of the file.

The easiest thing to do is to create a folder where you copy all the files that you want to change with the HexEditor.

Remember how all data processed by the computer is made up of a one or a zero? Well, the same principle holds true for files stored on the hard disk, on a floppy disk, on a CD-ROM, or on any other storage media. But because hexadecimal numbers are easier to deal with than binary numbers, we have programs that can read the content of any storage media as pure hexadecimal data. These programs are called HexEditors. Using the above idea, any file containing data that is stored on a media can be opened and it's contents will be represented as hexadecimal numbers. And it does not matter whether the file is an application program or just a simple text file, since ALL files are at their "lowest level" made up of binary numbers and can thus be viewed by a HexEditor.

The first thing you have to do is to find yourself a HexEditing program. It doesn't matter which computer platform you have. HexEditors exists for PC's, Mac's, Unix's, even C-64's. Once you've found a HexEditor open up any backup file with the program. I have a Mac and I use HexEdit 1.0.7, a freeware program by Jim Bumgardner. If I open an application file I get something like this:

```

000000: 00 01 00 00 00 00 17 A1 00 00 16 A1 00 00 00 B5 .....
000010: 66 69 6E 65 20 50 42 55 6E 6C 6F 63 6B 52 61 6E fine PBUUnlockRan
    
```

Please note that you WILL get something completely different, since the chances of us opening the same file is very slim, and different HexEditors present the information in different ways.

Let me explain the above. To the left you have the Offset column. "Offset" refers to the distance of a data from the first byte in the file. Since the offset here starts at zero we know that we are dealing with the beginning of the file. Also notice that the offsets are displayed as hex values. A good HexEditor should be able to display the offset as decimal numbers as well.

In the middle you have the Hex column. This is where all the hexadecimal data can be found. If you converted all these numbers to binary, you'd have a representation of the binary information of the file as you would find it on the Hard Drive.

Finally on the right side is the ASCII column. This is an ASCII representation of the Hex values. This means that each hex number is looked up on an ASCII table and it's ASCII value is displayed in this column.

OK, now what? Well, as an example I'll describe the use of HexEditors as a way to cheat on computer games.

Off course you can not use a HexEditor to cheat on a game while you are playing it. Those situations will be dealt with in the next chapter. What you can do with a HexEditor, however, is to change saved games. I mean, think about it. What is the program actually doing when it is saving a game? It saves all the data about the game to a file. Like where you are positioned on the map, what items you carry, how many monsters are gonna attack you etc... In this example I will use Realmz, a shareware game for the MacOS.

The first thing you have to do is to find where on the HardDrive the game saves it's files. Some games allow you to save wherever you want, while others will only allow you to save into a certain set of game folders (usually 1-10 or something like that). So search through the game's folders (directories as they are also called), and look for a file that has the same name as your saved game.

The next step is to find the document in which the game stores the information you want to change. For example Realmz is a Dungeons & Dragons game for the Mac where you can create your own characters. The attributes of the characters, such as it's strength or stamina, are saved in a file that has the same name as the character.

Let us presume that I have a character called Pro. His attributes are stored in the file called "Pro". I want to change my character's strength. I want to make him stronger so that he can cause more damage with each hit. The first thing I would do is to run the game and see how strong he is at that particular time. This will be the value that the game stores in the "Pro" file. He has a strength of 105. So I convert this number to hex, which gives me \$69.

And then I set out to look for the hex byte \$69 in the saved file. To make things easier I look for the hex word "00 69" since the possibility of the string "00 69" appearing several times in the file is smaller than that of the string "69". (Read "Note on HexEditors and numbers" for more information regarding this.) When I've found this value I change it to whatever I want it to be and then I save my work.

The problem might arise that "00 69" appears in more than one places in the file. The easiest (and most dangerous way) is to change all the values to the value you want. By doing this, however, you might have changed values which are very important to the program and might cause it to freeze. By using a trial and error method you can try to change a different value every time and see if the value you changed was the correct one. The most effective method, however, is to look at "00 69" in a context. Meaning, look at the other numbers around it.

For instance, if you recognize the number after "00 69" as the movement points of the character then there's a good chance that you're on the right track.

Note for Macintosh users: The MacOS divides up a file into two parts, the data fork and the resource fork. Without getting too much into programming, here's what the purpose of these two forks are. The resource fork should contain information such as how a window looks like, where it is located, how the menus look like etc. With other words information used by the Operating System. The data fork should be used to store the information used by the user's.

For example, in a word processor file the resource fork might contain information regarding the size of the window, while the data fork might contain the actual text written by the user. However, the programmer is not obliged to follow these criterias. They are only suggestions made by Apple. So, when you are looking at a file with a HexEditor on a Mac, be sure to check both forks of the file for the information you are looking for.

Note On HexEditors And Numbers

I find it appropriate to give a bit of a revision of numbers and strings.

To use the example from above, let's presume that my character had the strength of \$69. What we don't know is how the program stores this number. It might store it as a byte, a word, or a long (see chapter about bytes, words and longs for more info about this). Using common sense, if my character has a strength of \$69 and is considered very very strong than the program will probably save the value as a byte or a word. It's completely useless for it to store it as a long (although it might happen). If, however, we regard the characters experience point, its obvious that it is a lot larger than the range of a word, so it HAS to be stored in a long (or something larger). This might help you when searching for a value in the HexEditor.

Another thing that needs to be discussed is that the length of a number has to be even. A programmer deals with blocks (units) of memory. The program in it's turn reserves these block once it's launched. The smallest block a programmer deals with is a byte. This means that no matter how much the programmer wants it, he/she can never store the number "1" just like that.

If it is to be stored in the memory it will be stored as "01". However, if the programmer assigned the number to be a word it will be stored as "00 01". And if it was assigned to be a long it will be stored as "00 00 00 01". The computer doesn't care what number is stored in the variable. It only cares about the length of the variable. Thus if the computer stores three longs with the values \$1, \$22 and \$333 respectively then it will look like this once you open the file with a HexEditor:

```
00 00 00 01 00 00 00 22 00 00 03 33
```

Lets say you want to change the \$333 part to \$433. A good HexEditor might allow you to search for "333" but remember that the smallest unit is a byte. When you are changing "00 00 03 33" to "00 00 04 33" it's pointless to change all 8 digits. It's enough if you change the 3'rd byte ("03" to "04"). Notice, however, that you can't just change 3 to 4. You have to change "03" to "04". A good HexEditor should actually not allow you to change one digit at a time.

It should require you to change one byte, 2 digits, at a time. If you are confused then re-read this chapter, and the previous chapter dealing with lengths of numbers. This is important stuff, and it's very important that you know it well!

The Computer's Hardware Components

Now we have covered a lot of track. You should know what the different number systems are, you should have an understanding of different programming expressions and you should know how to use a HexEditor. In order to understand how a Debugger works, however, we need to dive into the hardware components of a computer. Do not worry, I will keep it simple. I will only talk about the most important parts of the computer. As a matter of fact you will most likely recognize and already know the function of some of these components.

- The Motherboard - This is that green board within your computer covered with circuits where all the hardware is placed. Everything from the diskdrive to the microphone is somehow connected to the Motherboard.

- Memory - The part of the computer where data is stored.

- RAM (Random Access Memory) - This is the temporarily storage facility of the computer. It is loaded full with information when you turn on the computer and it is emptied when you turn your computer off.

- PRAM (Parameter RAM) - Very much like ordinary RAM with the exception that there is a special battery in the computer providing the PRAM with enough electricity to keep the information in it even when the rest of the computer is turned off.

- ROM (Read Only Memory) - This is a storage unit where information can only be read from. With other words the computer can read anything in the ROM but it can not change anything there. Thus the ROM usually stores all the information the computer needs to be able to start when you press the "On" button. When you think about it, a compact disk (CD) is also a read-only unit. The computer can read the information on it, but it can't store stuff on it. Thus the name CD-ROM.

- Storage Media - For example, the HardDrive, a floppy disk, a CD or a Zip disk etc. These are accessories to the computer on which information is stored "indefinitely". That is "indefinitely" in the sense that the information will still be there, even when the computer off. This does, however, not keep the computer from replacing the information on the media. So it can freely read from it and write to it. It can even replace existing data with new data.

- The Processor - This is the brain of the computer. All data is sent to be calculated in the processor.

- Registers - These are blocks of memory within the processor where data is stored for a brief period of time, waiting to be processed.

- Busses - Circuits, on the motherboard, where the information travels from one component of the motherboard to the other.

- The Sound Card - This is like a small mother board with it's own processor, busses and registers capable of converting binary information to sound waves.

- The Graphics Card - Same as the sound card except it displays information as the graphics on the monitor.

And that's all you need to know for now.

Debuggers

I gave this whole chapter a lot of thought and decided on the following. I will only give a general description of a debugger, and some theoretical uses for it. For those interested I have included a file called "MacBug". I wrote this file a while back and it is not designed to be read by beginners. However, some people might find it handy. There are a quite a lot of other files dealing with MacBug that might be a lot more useful. So if you are really interested, read a few of those as well!

There are a lot of different debuggers out there, for all computer platforms. I use a Mac along with Apple's own free debugger called MacBug.

A debugger is a program that allows you to take control of the complete computer. This is done, by "stopping" the processor. When you activate a debugger, it stops the processor from executing any commands of the program you are running at the moment. The whole concept of a debugger is to help software developers look for mistakes in their programs or to see if it executes in a proper way.

As you may know, when a program is launched, the Operating System loads the program from the HardDrive to the RAM. This is done because the RAM is a lot faster than the HardDrive and most other storage medias. Then the processor jumps to the part of the RAM where the code of the program is stored, and it starts executing each of the commands. So, when the debugger is started the processor stops executing these commands. It then allows the user to check the values of the certain hardware components. This way the user can detect any mistakes in the program or just check the current state of the hardware components. The user can even step through the code of the program. This means that they can look at each command that makes up the program and see what it does. As I said before, debuggers are largely used by programmers trying to figure out why their program won't work properly.

For you, the main advantage of a debugger will be that it allows you to change the data in most hardware components, including the RAM. Since the program is loaded into the RAM when launched, and it does all the calculations in the RAM all the data/variables/information it may use will most likely be stored somewhere in the RAM. Thus the debugger can be used to change any of these.

Since this file has had a general undertone of being an aid for cheating on computer games I decided to include a way to use MacBug to cheat on games while you are actually playing them.

I will first summarize the theory and then go into the specifics. In order for you to be able to follow it through you will have to know at least the basic commands and functions of MacBug. If you are using a different debugger, then the theory will most likely be the same but the commands will be different.

WARNING: When you are changing memory contents or changing anything in a debugger for that matter, you CAN cause very large damages to your computer! The incorrect use of a debugger can cause the computer to freeze and cause information to be lost! Several other damages can also occur. Thus I advise you to become familiar with your debugger before you attempt to change anything with it. Read any related files, read the manuals and do some minor experimenting before you try to change stuff directly in the memory!

So first, the theory. I launch the game as a start. By opening up any saved games, I force the game to load anything it might have saved on the HardDrive (and that is of use to me) to the RAM. Then I stop the game by starting the debugger, I find where in the RAM the game is stored, I find the information I want to change and then I change it.

OK, and now for Practice:

Here's the scenario: I'm playing Heroes of Might & Magic II and I want more creatures in my armies. I open up the hero's preference window and see that my hero has 25 Minotaurs, 53 Dwarfs, 32 Griffins, 9 Skeletons and 2 Dragons. Thus I know that the computer keeps track of how many creatures I have and that means that the number of creatures must be stored somewhere in the RAM.

The first thing I have to do is to find out where in the RAM the game is located. The first step is to drop into MB (this is done by holding down the apple key and pressing the power button on the keyboard).

The second step is to issue the "hz" (heap zone) command that lists all the currently active applications and their locations in the RAM. I got this:

```
Heap zones
#1 Mod      7206K    00002800 to 0070C34F SysZone^
#2 Mod        6K    00008D60 to 0000A88F ROM read-only zone
#3 Mod        48K    001301F0 to 0013C1EF
#4 Mod       128K    004475B0 to 004675AF
#5 Mod     29560K    0070C350 to 023EA69F Process Manager zone
#6 Mod      9737K    010A6830 to 01A28EFF "Heroes II" ApplZone^ TheZone^ Targ
```

As you can see Heroes II starts at memory location 010A6830 and ends at 01A28EFF (all in hex of course).

The next step is to use the "find" command and find the number of creatures that make up my army. See, it is very likely that a game stores relevant data close to each other. So I presume that the program stores the number of creatures I have, in a specific block of memory in the RAM. If I can find this block of memory, I will be able to change it's content, thus changing the number of creatures. In some ways it's like finding information with a HexEditor. Except you're looking for data in the RAM and not in a file.

In order to be able to use the "find" command I have to be able tell the following things: the start of the memory address, how many bytes the debuggers should search for, and what to search for. Unfortunately I don't have all the criteria. I have to find out how many bytes Heroes II occupies. This can easily be done by subtracting \$010A6830 from \$01A28EFF. This subtraction gives me \$009826CF. If you want you can do this calculation directly in MB, just type "01A28EFF-009826CF". Now I have all the stuff I need to use the find command.

In this example I issue "f 010A6830 009826CF 00190035"

The "f" stands for "find". This tells MB to use the find command. "010A6830" is the address of the memory where Heroes II starts. "009826CF" stands for the number bytes Heroes II occupies in the memory. It tells MB the number of bytes I want to search for, from the initial address.

"00190035" stands for 25 Minotaurs and 53 Dwarfs. #25=\$19 and #53=\$35. Since I've done this before I know that Heroes II stores the the number of creatures in word sized blocks of memory. If I didn't know that I would have had to search for "0019" first (or "19") and look at it in it's context. When I issued the find command I got this:

```
Searching for 00190035 from 010A6830 to 01A28EFE
0118EE3E 0019 0035 0020 0009 0002 0000 0003 0100 ...5.....
```

The first hex long represents an address in the memory. The following four longs (16 bytes) are the values of the data contained in the RAM starting from that address. The following 16 characters are the ASCII representations of these values.

Now, if I convert the first five words to decimal numbers I get: 25, 53, 32, 9 and 2. That's a perfect match of the number of creatures I have in my army. Thus there is a fairly good chance that Heroes II keeps track of my army starting at address 0118EE3E. When you are doing something like this on your own and you don't think that this is the location of the memory that you are looking for, you can continue searching by hitting return until you get a message saying that it could not be found.

Then comes the dangerous part, I have to change the blocks of memory. I issue the following command, "sw 0118EE3E 00ff" (sw stands for set word). This changed the word at the memory address 0118EE3E from "0019" to "00ff". If I now issue the "dm 0118EE3E" command (dm stands for display memory) I see that the value at address 0118EE3E has changed to:

```
0118EE3E 00FF 0035 0020 0009 0002 0000 0003 0100 ...5.....
```

So I return to the game by issuing the "g" command. Apparently nothing has changed. But if I close the preferences window and force the game to actually check how many Minotaurs my army has (by checking the number stored in the RAM), then I can see that the game in fact thinks that I have 255 Minotaurs! Cheat accomplished. Now I just have to repeat the above procedures for all the other creatures.

NOTE: when you are changing the contents of the memory, make sure that you use the appropriate addresses, meaning the ones you get when you issue "hz" and the find command. Do NOT use the memory addresses I used! They are purely examples and WILL NOT work on your computer!

The most common ASCII characters (to be viewd in Monaco)

32 ' '	\$20	33 '!'	\$21	34 '"""	\$22	35 '#'	\$23	36 '\$'	\$24
37 '%'	\$25	38 '&'	\$26	39 '''	\$27	40 '('	\$28	41 ')''	\$29
42 '*'	\$2A	43 '+'	\$2B	44 ','	\$2C	45 '-'	\$2D	46 '.'	\$2E
47 '/'	\$2F	48 '0'	\$30	49 '1'	\$31	50 '2'	\$32	51 '3'	\$33
52 '4'	\$34	53 '5'	\$35	54 '6'	\$36	55 '7'	\$37	56 '8'	\$38
57 '9'	\$39	58 ':'	\$3A	59 ';'	\$3B	60 '<'	\$3C	61 '='	\$3D
62 '>'	\$3E	63 '?'	\$3F	64 '@'	\$40	65 'A'	\$41	66 'B'	\$42
67 'C'	\$43	68 'D'	\$44	69 'E'	\$45	70 'F'	\$46	71 'G'	\$47
72 'H'	\$48	73 'I'	\$49	74 'J'	\$4A	75 'K'	\$4B	76 'L'	\$4C
77 'M'	\$4D	78 'N'	\$4E	79 'O'	\$4F	80 'P'	\$50	81 'Q'	\$51
82 'R'	\$52	83 'S'	\$53	84 'T'	\$54	85 'U'	\$55	86 'V'	\$56
87 'W'	\$57	88 'X'	\$58	89 'Y'	\$59	90 'Z'	\$5A	91 '['	\$5B
92 '\'	\$5C	93 ']'	\$5D	94 '^'	\$5E	95 '_'	\$5F	96 '`'	\$60
97 'a'	\$61	98 'b'	\$62	99 'c'	\$63	100 'd'	\$64	101 'e'	\$65
102 'f'	\$66	103 'g'	\$67	104 'h'	\$68	105 'i'	\$69	106 'j'	\$6A

107 'k'	\$6B	108 'l'	\$6C	109 'm'	\$6D	110 'n'	\$6E	111 'o'	\$6F
112 'p'	\$70	113 'q'	\$71	114 'r'	\$72	115 's'	\$73	116 't'	\$74
117 'u'	\$75	118 'v'	\$76	119 'w'	\$77	120 'x'	\$78	121 'y'	\$79
122 'z'	\$7A	123 '{'	\$7B	124 ' '	\$7C	125 '}'	\$7D	126 '~'	\$7E
127 ''	\$7F	128 'Ä'	\$80	129 'Å'	\$81	130 'Ç'	\$82	131 'É'	\$83
132 'Ñ'	\$84	133 'Ö'	\$85	134 'Ü'	\$86	135 'á'	\$87	136 'à'	\$88
137 'â'	\$89	138 'ä'	\$8A	139 'å'	\$8B	140 'â'	\$8C	141 'ç'	\$8D
142 'é'	\$8E	143 'è'	\$8F	144 'ê'	\$90	145 'ë'	\$91	146 'í'	\$92
147 'ì'	\$93	148 'î'	\$94	149 'ï'	\$95	150 'ñ'	\$96	151 'ó'	\$97
152 'ò'	\$98	153 'ô'	\$99	154 'ö'	\$9A	155 'õ'	\$9B	156 'ú'	\$9C
157 'ù'	\$9D	158 'û'	\$9E	159 'ü'	\$9F	160 '†'	\$A0	161 '°'	\$A1
162 '¢'	\$A2	163 '£'	\$A3	164 '§'	\$A4	165 '•'	\$A5	166 '¶'	\$A6
167 'ß'	\$A7	168 '®'	\$A8	169 '©'	\$A9	170 '™'	\$AA	171 '´'	\$AB
172 '¨'	\$AC	173 '≠'	\$AD	174 'Æ'	\$AE	175 'Ø'	\$AF	176 '∞'	\$B0
177 '±'	\$B1	178 '≤'	\$B2	179 '≥'	\$B3	180 '¥'	\$B4	181 'µ'	\$B5
182 'ð'	\$B6	183 'Σ'	\$B7	184 'Π'	\$B8	185 'π'	\$B9	186 'j'	\$BA
187 'ª'	\$BB	188 'º'	\$BC	189 'Ω'	\$BD	190 'æ'	\$BE	191 'ø'	\$BF
192 '¿'	\$C0	193 '¡'	\$C1	194 '¬'	\$C2	195 '√'	\$C3	196 'f'	\$C4
197 '≈'	\$C5	198 'Δ'	\$C6	199 '«'	\$C7	200 '»'	\$C8	201 '…'	\$C9
202 ' ' '	\$CA	203 'À'	\$CB	204 'Ã'	\$CC	205 'Ö'	\$CD	206 'Œ'	\$CE
207 'æ'	\$CF	208 '–'	\$D0	209 '–'	\$D1	210 '“'	\$D2	211 '”'	\$D3
212 '‘'	\$D4	213 '’'	\$D5	214 '÷'	\$D6	215 '◇'	\$D7	216 'ÿ'	\$D8
217 'ÿ'	\$D9								

End Notes

In conclusion I hope to have given an insight to how different principles of computer technology work. I'd like to point out to some more advanced readers that I am aware of the fact that I have generalized and simplified some concepts. I did this only when I felt that the theory was more important than a detailed explanation. I also hope to have made some of you interested in learning about programming and more advanced topics of IT and computer technology.

Good luck

ProZaq
1999.12.28

Werd to mSEC, and everyone else who's ever helped me out! It's people like you who make it worthwhile!

PPC Cracking v2.0

by ProZaq

Table of Contents:

- Introduction
- Tools
- The PPC Processor in General
- Using MacsBug in a PPC Environment

- Cracking
- Hints for Using MacsBug in a PPC Environment
- Using a PPC Disassembler
- Using The Hex Editor
- Note About FAT Binary Applications
- End Notes

Introduction

My aim in this tutorial is to introduce the general techniques for cracking PPC software for people who are familiar with the 68k processor structure. I will not go into detail about PPC assembly nor will I give any practical examples. You shouldn't need those! The purpose of this file is simply to explain the techniques used for cracking PPC software.

If you are new to cracking then I strongly recommend that you go through my other files on cracking (The Ultimate Mac Cracking Guide) because (and trust me on this one) PPC cracking does get ugly! You can find them in various issues of HackAddict or you can download them from my homepage:

www.geocities.com/area51/rampart/4007/Welcome.html

In those tutorials I describe in great detail what cracking is all about. And the truth is that exactly the same principles apply to PPC cracking. The only problem is that there aren't too many tools available to help you on your quest to crack PPC programs. So you'll have to improvise quite a lot.

Trust me, unless you're a programmer or a practiced cracker you'll have no clue as to what I'm talking about most of the time in this file!

Tools

As always I've keep the price of the tools required down to a minimum (\$0) and you shouldn't have any problems finding these programs:

1. MacsBug - The cracker's best friend! There's no need to go around finding "demos" of the other debuggers when g'old MacsBug works just fine!
2. ResEdit - All serious Mac user's best friend! Unfortunately ResEdit is not capable of disassembling the PPC assembly code, so bugger Super ResEdit, any old version will do just fine.
3. Any hex editor capable of reading both data and resource forks - I always use HexEditor by Jim Bumgardner (which is freeware), but I guess the data fork editor extension of ResEdit would do just as well...
4. A PowerPC Disassembler:
 - PPCdissasembeler 2.0 by Alain Birtz - This is sorta like the Code editor part of Super ResEdit. The good part about this particular program is that it displays the meaning of the PPC assembly mnemonics.

- Janus 0.1 by Peter Creath - This can sorta be used to find out information about the a-traps used by the program. I don't use it much. It has a rather nasty interface, and don't give all that much useful information...

- PEF Viewer v1.0d8 - Apple's own PPC disassembler. It has a nice interface and is the most usable of them all. Unfortunately not even this one can save any changes to the coding itself.

- MacNosy - Pay for it if you want, use it if you want. It's a lovely tool, but I must say that you can crack just as well without it.

6. Wetware - You will definitely need your head! So make sure you're in immediate possession of it whilst cracking PPC software!

The PPC Processor in General

They claim it's fast, and there's no doubt about that. They claim it's got a lotta registers, well 32 general purpose ones actually (plus all the FPU and 64 bit ones). And they claim it's better than the 68k processor, uhm... whatever. Sure it's got all those registers, but they are general purpose ones. And in practice you only use about 20 of them... Lemme warn you straight away: don't mess with the first three registers (r0, SP and TOC)! They hold information that are essential to the execution of the program such as return tables and lovely things like that.

I guess the biggest changed to get used to is that there are no address or data registers! Any of the registers can be used as either data or address registers. Meaning that it can hold a number or it can serve as a pointer to a location in the memory. It's up to you to figure out the purpose of the register. Something I noticed rather early on is that I was forced to use the "dm" (display memory) command all the time, just to make sure that the damn register wasn't used as a pointer.

Another thing that shocked me was that all PPC commands are 4 bytes (8 digits) long. Oh well... this isn't the place to talk about the speed of executing 4 digit commands...

I will unfortunately not include a list of the assembly commands and what they mean in this tutorial. I guess it's not too important that you know exactly what each command means, but it might be useful to get a hold of a definition of all the different mnemonics, just in case... I can on the other hand recommend the reference manual to PowerFantasm or some files from Apple regarding programing in PPC assembly.

This on the other hand is important to know: every subroutine starts with the "mflr" command and ends with the "blr" command. Apparently a lot of times there are several "blr" commands within a subroutine (sort of a contradiction of terms huh?). This way the program doesn't use the BRA command to branch to the end of the subroutine but ends it straight away once it's done it's purpose. I guess this is how they try to make up for the speed loss of the 4 byte long commands ;-). Another PPC assembly command that might be useful to recognize is the "bl" command. This is the common branch command that is know in 68k terms as Branch-To-Subroutine (BSR).

The BRA command has the hexadecimal value 4800xxxx and the NOP command has the value 60000000. So whenever you want to replace a conditional branch command with a BRA command, you change it's leftmost 4 digits to "4800" and keep the rest as it was. For example if you wanted a BEQ command represented by 41820034 to always branch you'd change it to 48000034

(remember not to change the rightmost 4 digits, as that will seriously fuck things up!). And you'd just replace the whole thing by 60000000 if you didn't want it to branch.

Another thing to notice when looking at PPC assembly code is that in simple arithmetic the registers are used in reverse order. For instance, here's an example of a simple addition in 68k:

```
add.w    d1,d0
```

As you all know the values of d1 and d0 would be added up and stored in d0. In PPC the above example might look something like this:

```
add     r3,r4,r5
```

However, the execution process is reversed. The above command adds r5 to r4 and stores their sum in r3.

Now then, A-Traps. Well they are known as system vectors/ symbols in PPC programing (but I'll just stick to the expression "a-trap" to be nostalgic). Apparently this is supposed to be a much faster alternative to patching the A-Traps, but for crackers they are a real pain in the ass!

Also if you take a look at the resource fork of a PPC file with ResEdit you will find that all the usual resources are there with the exception of the CODE resource. And if it is there then it's usually just some warning that the program won't run on 68k platforms. So where is the actual code? You guessed it; in the data fork. That's why you need a hex editor if you want to change it.

Using MacsBug in a PPC Environment

The PPC part of MacsBug is structured the same way as the 68k part. To the left you have the stack, below it the current application name, and below that you have the registers. In the middle you have the assembly commands, and to the far right you have the command's hexadecimal value.

Unfortunately MacsBug is not developed to deal with PPC as well as it does with 68k assembly, but it is still very usable! The command used to break at a specific a-trap, is the "tvb" (TVector Break) command. Notice, however, that while in 68k it was enough to type "atb modal" to set a break for the ModalDialog a-trap, in PPC you have to type out the whole name of the a-trap. So if you want to be dropped into MacsBug at every ModalDialog a-trap, you have to issue the command "tvb ModalDialog".

The equivalent of "atc" is the "tvc" command.

Another thing that sucks is that whenever MacsBug breaks for an a-trap it does not actually break before the trap but breaks at the first command inside the trap. So while in 68k you were dropped inside the program itself at the command branching off to the a-trap's subroutine, in PPC you're dropped into the actual a-trap, and have to get out of it first (remember that all subroutines end with the "blr" command).

Once you're out of the a-trap though, you can continue tracing/ stepping through the code as if you always did while cracking 68k programs.

Cracking

You start off exactly like when you were cracking 68k programs. Issue the "tvb" command just like you would with the "atb" command.

Once you're dropped into MacsBug, remember that you are actually inside the a-trap subroutine so you need to trace through the code until you see the "blr" command. Everything that comes after that belongs to the program you're trying to crack. Remember that the PPC mnemonic for the "BSR" command is "bl". You can step into/ trace over any subroutine just like you could in 68k code using the "s" and the "t" commands.

Once you've found the conditionals you're looking for, the same principles apply in PPC code as they do for 68k code. Let's say that the below code won't branch but it would be "nice" if it would:

```
01A03244 beq      $+0x0024          ; 0x01A03268 | 41820024
```

The "0x01A03268" part tells us (just like in 68k) that if the conditions would have been met it would have branched to the command that's in the memory at address "01a03268" ("0x" prefix is used to represent hexadecimal numbers). Now then, luckily the Program Counter (a special address register that keeps track of which command is to be executed next) works the same way in PPC assembly as in 68k. Thus, if you want the above command to branch, simply set the Program Counter to 01a03268. You would achieve this by typing "pc=01a03268".

On the other hand, you might be in a situation where you want the conditional not to branch. So simply jump over the command. How? Well since in PPC all commands are 8 digits long it makes things simpler than they are in 68k. If you ever have to jump over a command, type "pc=pc+4" (the current pc address plus 4 will give you the address of the next command in line).

Hints for Using MacsBug in a PPC Environment

A very useful feature of MacsBug is the disassemble command, "il". Since there are no fabulous tools to use for disassembling PPC software, you might as well use the Debugger to do it. So how do you use the "il" command? If you want to disassemble from the current address just type "il". This can be very useful when you're trying to figure out what would happen if a conditional didn't branch.

If you want MacsBug to disassemble the assembly commands from a specific address you'll have to use the "il-c" command. Simply because if you don't MacsBug will disassemble the PPC commands in 68k mode and you'll get some very interesting results.

I also found it useful in some cases to figure out what happens in the subroutine before it hits a specific a-trap (especially when I was cracking software protected with hardware keys). This can very easily be achieved by typing "il-c pc-x" where "x" is any multiple of 4 (remember that every PPC command is 4 bytes, 8 digits, long). And if you recall your hexadecimal rules then hex 10 equals dec 16. So it's very convenient just to disassemble using "x" as multiples of hex 10 (meaning 4 assembly commands at a time).

To clear things up, if you want to disassemble from the previous assembly command you type "il-c pc-4". If you want to disassemble starting from the 4th last command just use "il-c pc-10". You'll get the hang of it!

Another very useful command in PPC cracking is the break command, "br". If you care to read the help section of MacsBug you'll notice that the "br" command in a PPC environment causes an error. So use the "brp" command instead! Both "br" and "brp" can be cleared with the "brc" command.

You'll find break points to be a very useful alternative method to the "tvb" command. Remember how when you're using the "tvb" command you are actually dropped inside the a-trap? Well once you find your way out of it, you can simply set a break point at the a-trap's subroutine and clear the TVector break. This way you'll still break every time for the a-trap, but before you actually enter the trap.

A very simple way of using the break command is with the help of the Program Counter. If you want to set a breakpoint at the previous assembly command type "brp pc-4"; if you want a break at the current command type "brp pc"; and if you want a break at the next command in line type "brp pc+4" (useful when you wanna get outa long loops). Rather simple, and saves you a lotta time!

Using a PPC Disassembler

In theory, there's no "real" need for any PPC disassembler when you think about it. I mean, you can find out most of the things you want using MacsBug, and since you can't change the program with PPCdisassembler or PEF Viewer anyway (nor Nosey for that matter), there's not much point in using them. Well except maybe for one thing. And that is that PPCdisassembler tells you what the different assembly mnemonics mean (for example "addi" stands for "add immediate"). So if you're in doubt or despair, run around in circles, scream and yell! No, but whenever you can't figure out the meaning a command, you can always look up the command in PPCdisassembler and it'll explain it to you. Oh, yeah, don't forget to turn on the "add meaning" option in the Preferences. You might also want to set the origin to zero, to make the offsets of the commands realistic.

There is another use for the disassemblers. And that is the following. In 68k, when you were looking for the resource ID's of dialogs or windows you used the CodeEditor part of Super ResEdit. You can use the disassemblers to do this. Just search for the hex value of the dialog ID you're looking for and see if there's a branch command right after it somewhere. Since the first 2 bytes of the PPC assembly command are used for the mnemonic, don't just search for "80", search for "0080". This will greatly speed up your search. [If you're using PPCDisassembler when searching for "0080" for example, make sure that you mask it with "FFFF" (open the "search" dialog and you'll understand what I'm talking about).]

If you're looking for the part of the code that passes a 4 letter resource to the memory, then you have to realize the "limits" of PPC. One of the limits is that you can only pass 2 byte long digits into a register with one call. So pushing the letters "PREF" into a register would require two commands. A typical example of how this is could be done:

```
lis    R4,$5052    * load immediate shifted, 'PR'
li     R4,$4546    * load immediate, 'EF'
```

Using The Hex Editor

Right, so you finally found the conditional you need to change. What do you do? First of all, write down the hexadecimal value of the command you wish to change, along with the values of the next or previous four commands as well. Then, since all assembly information is stored in the program's data fork, open the program with your hex editor. Search for the hex values you wrote down, and replace the undesired one with 4800xxxx (BRA) or 60000000 (NOP) depending on what you need.

Quit the hex editor, launch the program you're trying to crack, and watch your computer freeze! Well that is if you fucked up somewhere on the way ;-). If you didn't you should have a fully functional cracked program!

A little hint here, in order to avoid unwanted freezes, when you're searching for the the hex values, make sure that there's only one set of those in the data fork! Since you can't really work with offsets as you could in the 68k environment, you are forced to go trial and error. I think it's obvious that if you write down many hex values of assembly commands, then the chance of them occurring in that specific order several times is reduced! Really, this is where the wetware part of the whole thing kicks in! Use your head!

Note About FAT Binary Applications

FAT applications work on the principle, as you all know, that it'll run on both PPC and 68k computers. It achieves this by having the 68k code in the CODE resource and the PPC code in the data fork. There are two freewares out there called Strip 68k/ PPC by Phase Consulting. What they do is that they take away either the 68k or the PPC code from a FAT application. So if you feel more comfortable cracking 68k apps, then use the above program to remove the PPC coding from it!

NOTE: When I stripped the PPC code from a larger application I was faced with all sorts of error dialogs.

End Notes

And that's about all you need to know. If there's any need for it or if I'm extremely bored I might write a beginners guide to PPC cracking, but until then you'll just have to learn how to crack in 68k properly first! ;-)

1998.11.10 - updated 1999.05.16
ProZaq

Basic Mac PPC Cracking

by Dot Com December 12, 1997

I have read many cracking tutorials and have noticed they are pretty hard to understand. I first learned cracking thru Smeger's excellent text, "The Kool Krack Tutorial". The learning curve was pretty tough at first but I will attempt to get you rolling as quickly as possible thru examples. In this part we will use Nosy to bypass a typical serial number dialog box and other annoying alerts. There are many different approaches to bypassing a dialog box but this is one basic method I use the most often because its damn easy.

Software you will need

You will need the following software before you begin this tutorial:

Resorcerer 2.0
 Nosy II 8/97
 Macsbug 6.5.4a3c1 THINK Reference 2.0
 WebCollage 1.0

Resorcerer is a resource editor like ResEdit. I prefer Resorcerer because of its excellent search features. We will use Resorcerer to modify code and checkout dialog boxes. Nosy is a PPC disassembler and will convert all assembly code into something we can read and understand. Macsbug is a debugger and disassembler that will allow us to see what code is executing. You will not need Macsbug in this lesson but get it anyway. THINK Reference is a handy utility that tells you all about Mac Toolbox Traps (more on this in a bit). You will also need to locate a copy of StarNine's WebCollage 1.0 from your local warez site (it may also be downloadable from <http://www.starnine.com>). This is the app that you are going to attempt to crack.

Basic Assembly Language

I have to admit that I know nothing about assembly language (I dont even know that the hell it is!) but I have not found it necessary to know a whole lot, so I wont bore you with all that stuff. There are really only 4 pieces of assembly knowledge that you really need to know (at least for now) with the first being conditional branches.

Conditional branches are lines of code that compare values and go to or "branch" to another line of code. If your familiar with the old BASIC language its like an IF THEN statement. IF <serialnumber> = <12345> THEN <linenumber>. There are two types of branches we need to know which are BNE (branch if not equal) and BEQ (branch if equal). The machine language equivalent codes for these are 40 and 41. What the hell is machine language you say? Hell if I know! This will make sense later so bare with me. We use branches to force the program to do something different from what it normally does, kinda like a detour. A conditional branch statement looks like this:

```
4182 0014 bc IF,cr0_EQ,laq_3
```

so this line of code branches if the value cr0 is equal and goes to procedure laq_3 (in Macsbug it will be a numeric value or line number, more on that later)

The next type of assembly we need to know is NOP which stands for No Operation. This is basically a line that does nothing..our macs just skip over this line. The machine language equivalent is 6000. We use this to delete lines of code. A NOP statement looks like this:

```
6000 0000 nop
```

A line that loads a procedure or subroutine is also a branch but I feel its a little different than a normal branch. A BL or "branch load" is just like a GOSUB in ole BASIC and it looks like this:

```
4BFF FFDD bl proc13
```

so this line of code will run all the code in proc13.

And last but not least we need to know LI. What does LI do? I have no idea, but it means "Load Immediate" I believe. All I know is that they are useful for seeking out Dialog box ID numbers. A LI statement looks like this:

```
38A0 06A4 li    r5,$6A4
```

Using Resorcerer

Upon running WebCollage Editor we of course get the typical register me dialog box as shown in Figure 1.

Well since we dont seem to have an authorization key handy I guess that where gonna have to bypass this dialog box somehow and get the program to load up for us. Most (not all) dialog boxes in an application have a Dialog ID number. We will use Resorcerer to locate this number. Load up Resorcerer and load in WebCollage Editor and select the DLOG Resource and we see Figure 2.

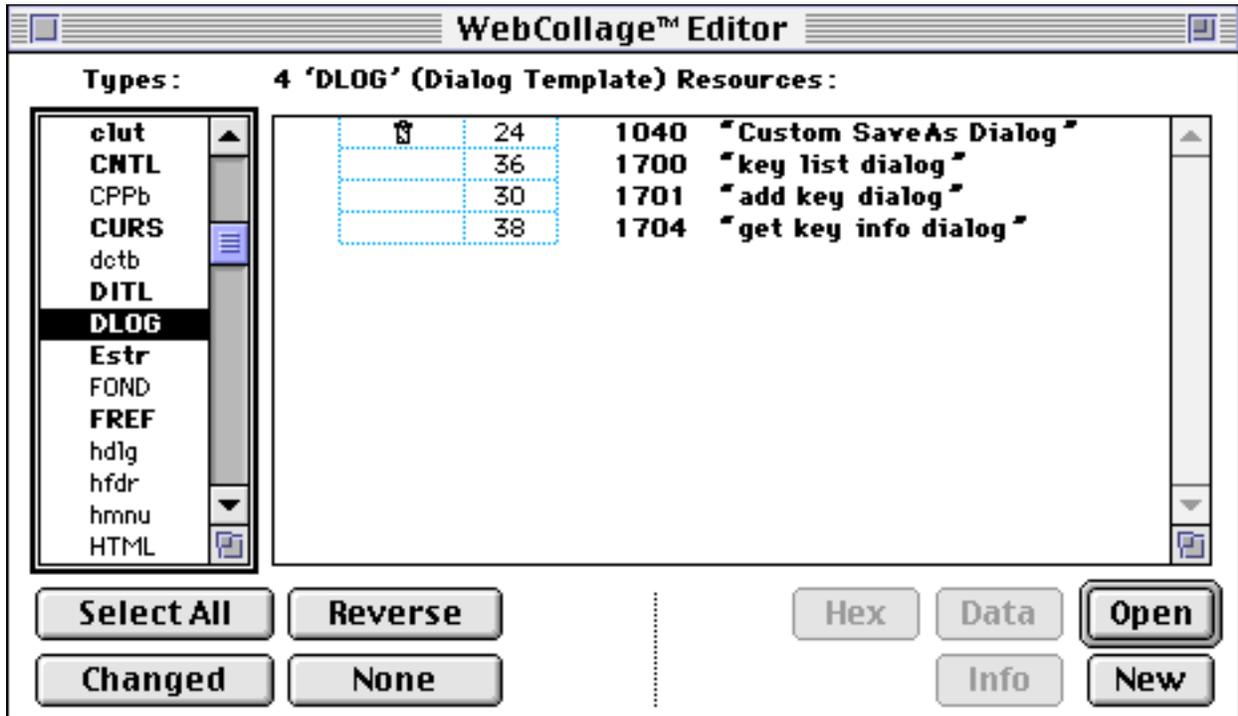


Figure 2

We find that Dialog number 1701 "add key dialog" is the problem. Sometimes they dont label there dialogs as did StarNine so you will have to load up each dialog and find the right one. Now we need to convert the ID number into Hex so we can locate it in Nopsy. What is Hex anyway you say? Well you guessed it, I dont know. I guess its just numbers that assembly code can recognize, who knows? Anyway go to the Edit Menu and select "Value Converter" and type in 1701 in the Long field as seen in Figure 3.

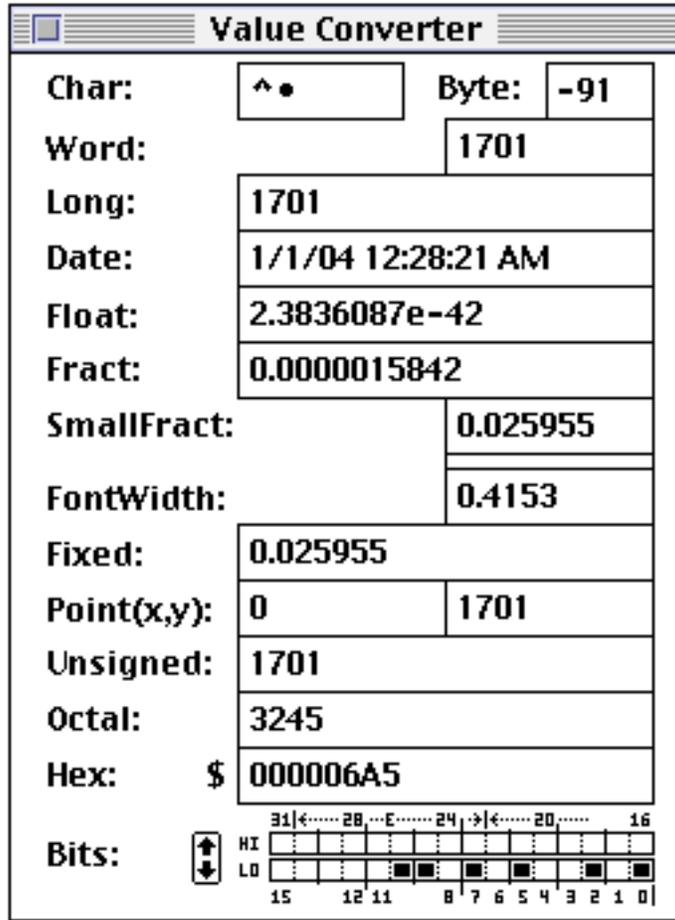


Figure 3

Now in Hex, 1701 = 06A5. Now its time to load up Nosy and snoop around the Dialog traps and try to locate the annoying procedure that calls Dialog 1701.

Toolbox Traps

Smeger described Toolbox Traps better than I can:

The Toolbox is a set of routines that mac programmers can use to simplify common tasks, making writing code really simple 'cause you don't have to do anything. A trap is a system routine that performs some sort of action, such as drawing a menu bar or a window. Traps are stored within a program as a single instruction. When the trap is called, the program will perform the trap, then continue execution normally.

Got That? Good. For a complete listing of Toolbox Traps and what they do I use the THINK Reference app. You should be able to find it no problem. The trap calls we are interested in are any traps relating to Dialog Boxes such as GetNewDialog, GetDialogItem, etc. GetNewDialog seems to be the best one and I use it everytime.

Using Nosy

Nosy is a great program that will disassemble the Data Fork (where PPC code is located) into a format that we can read. First duplicate WebCollage Editor and rename the copy to just Editor (Nosy only accepts 20 character filenames) and load it into Nosy. Select the <DF> when Nosy asks you to select a resource. Press Continue for the Treewalk options and let Nosy explore. Nosy will take a few minutes and disassemble the program. The time it takes varies on the filesize. Sometimes Nosy will not be able to disassemble part of a program and Macsbug will have to be used instead which will be covered in Part 2 of this series. In this case Nosy loads the Editor fine and we get a window displaying all the Code Blocks as in Figure 4. Keep scrolling down and you will find all the Toolbox Traps used in the application.



Figure 4

Scroll down until you find .GetNewDialog_GL_. Select it and press _ R (also under the Display Menu: Show Refs to). This will show all procedures that reference the GetNewDialog trap. We now get the window in Figure 5.



Figure 5

Well we lucked out as only 3 procedures have new dialog calls. Sometimes there can be 20 or more and you will have to snoop thru them all. Anyway lets check out proc3233. Select it and press _ D (also under Display: Code Blk) and we get Figure 6. Scroll down a bit until you find a BL for the .GetNewDialog_GL_.

```

proc3233
8C8A8: 7C7A 1B78          mr      r26,r3
8C8AC: 3861 013C          300013C la      r3,wkg_3(SP)
8C8B0: 4801 2541          109EDF0 bl      .GetPort_GL_
8C8B4: 8041 0014          lwz     RTOC,20(SP)
8C8B8: 2819 0000          cmpli  cr0,0,r25,0
8C8BC: 4182 000C          108C8C8 bc     IF,cr0_EQ,mkg_1
8C8C0: 3860 06A5          li      r3,$6A5
8C8C4: 4800 0008          108C8CC b      mkg_2
8C8C8: 3860 06A5          mkg_1  li      r3,$6A5
8C8CC: 3880 0000          mkg_2  li      r4,0
8C8D0: 38A0 FFFF          li      r5,-1
8C8D4: 4801 3C5D          10A0530 bl      .GetNewDialog_GL_
8C8D8: 8041 0014          lwz     RTOC,20(SP)
8C8DC: 7C7E 1B78          mr      r30,r3
8C8E0: 281E 0000          cmpli  cr0,0,r30,0
8C8E4: 4082 0020          108C904 bc     IF_NOT,cr0_EQ,mkg_3
8C8E8: 4801 3961          10A0248 bl      .ResError_GL_
8C8EC: 8041 0014          lwz     RTOC,20(SP)
8C8F0: 7C7F 1B78          mr      r31,r3
8C8F4: 7FE0 0735          extsh. r0,r31
    
```

Figure 6

Notice above the first GetNewDialog we have a:

```
8C8C0: 3860 06A5      li      r3,$6A5
```

Thats our man as we see Hex ID 6A5 being loaded just before the dialog call. Bypassing this routine should solve the problem right? Nope. Notice the branch at address 8C8BC: bc IF,cr0_EQ,mkg_1. If we reroute that branch to mkg_1 it still loads the dialog and there are no branches at the start of proc3233 to bypass all of this. So what do we do now? We bypass this whole procedure altogether. Scroll back up to the top of proc3233 and we see the following:

```
;-refs - proc3232 proc3250
```

This shows all the references to this procedure. Lets _ D proc3232 and we see that nothing is apparant but the BL to proc3233. What we want to find is a conditional branch before loading proc3233. Its not here so lets checkout the only ref to proc3232 which is proc3231. Well no conditional branches in proc3231 as well but an interesting LI line referencing \$6A4.

If we type 6A4 into the Value Converter in Resorcerer we find that its doing something with Dialog 1700 "key list dialog", definetly something we want to avoid. The only ref to this procedure is proc20, lets check it out. Well, proc20 is a big one. Do a search for proc3231 with a _ F. Type in proc3231 and it we find the contents in Figure. 7.

```

proc20
1D50: 3860 004C          li      r3,76
1D54: 4808 D7B5          108F508  bl      proc3267
1D58: 6000 0000          nop
1D5C: 7C60 0735          extsh.  r0,r3
1D60: 4182 0048          1001DA8  bc     IF,cr0_EQ,lau_9
1D64: 3860 FFFF          li      r3,-1
1D68: 3880 004C          li      r4,76
1D6C: 38A0 0000          li      r5,0
1D70: 4808 AA21          108C790  bl      proc3231
1D74: 6000 0000          nop
1D78: 7C60 0735          extsh.  r0,r3
1D7C: 4182 002C          1001DA8  bc     IF,cr0_EQ,lau_9
1D80: 4806 A54D          106C2CC  bl      proc2438
1D84: 6800 A39D          100C2CC  br-    proc2438
1D88: 3860 014D          li      r3,$14D
1D8C: 3880 0000          li      r4,0
1D90: 4809 EC21          10A09B0  bl      .NoteAlert_GLL
1D94: 8041 0014          lwz     RTOC,20(SP)
1D98: 4806 A599          106C330  bl      proc2439
1D9C: 6000 0000          nop
1DA0: 3800 0002          li      r0,2
1DA4: 981D 0064          stb     r0,$64(r29)
1DA8: 8062 8AE4          -751C  lau_9  lwz     r3,pU_1771C(RTOC)
1DAC: 8063 0000          lwz     r3,0(r3)
1DB0: 8063 0000          lwz     r3,0(r3)
1DB4: 4809 D445          109F1F8  bl      .SetCursor_GLL
1DB8: 8041 0014          lwz     RTOC,20(SP)
1DBC: 807D 006C          lwz     r3,$6C(r29)

```

Figure 7

If we keep clicking the Find button we see that this is the only occurrence of proc3231 in this procedure. Well there it is; a conditional branch above the call to proc3231:

```
4182 0048      1001DA8      bc          IF,cr0_EQ,lau_9
```

Notice the NoteAlert in Figure 7, the LI above it calls \$14D. Go back to the Value Converter in Resorcerer and type in 14D and we get 333. Select the ALRT (Alert dialogs are located here) resource type and double click on 333 and we get a nice ALRT saying that "None of the authorization keys are valid. Please contact StarNine for more information". Well this has got to be the right place. It looks like if we force the application to reroute to lau_9 then we should avoid the serial number dialog box and the ALRT. All we have to do is change it from a BEQ to a BNE. How do we do that? Easy. First lets examine the code around the line we want to change:

```

4808 D7B5      108F508      bl          proc3267
6000 0000          nop
7C60 0735          extsh.      r0,r3
4182 0048      1001DA8      bc          IF,cr0_EQ,lau_9
3860 FFFF          li          r3,-1
3880 004C          li          r4,76

```

See the number 41820048? That is the assembly code for this line. What we are going to do is search the <DF> in Resorcerer for this line. What we find is that there are many 41820048's in the <DF> so we need to copy down the surrounding code so we make sure we are in the right place. The surrounding code would be :

```
600000007C600735 41820048 3860FFFF3880004C
```

Modifying Code in Resorcerer

To make the change we will use Resorcerer. In Resorcerer and select the <DF> resource type and do a `_ F` and copy in the code with **NO** spaces in between the codes as in Figure 8.

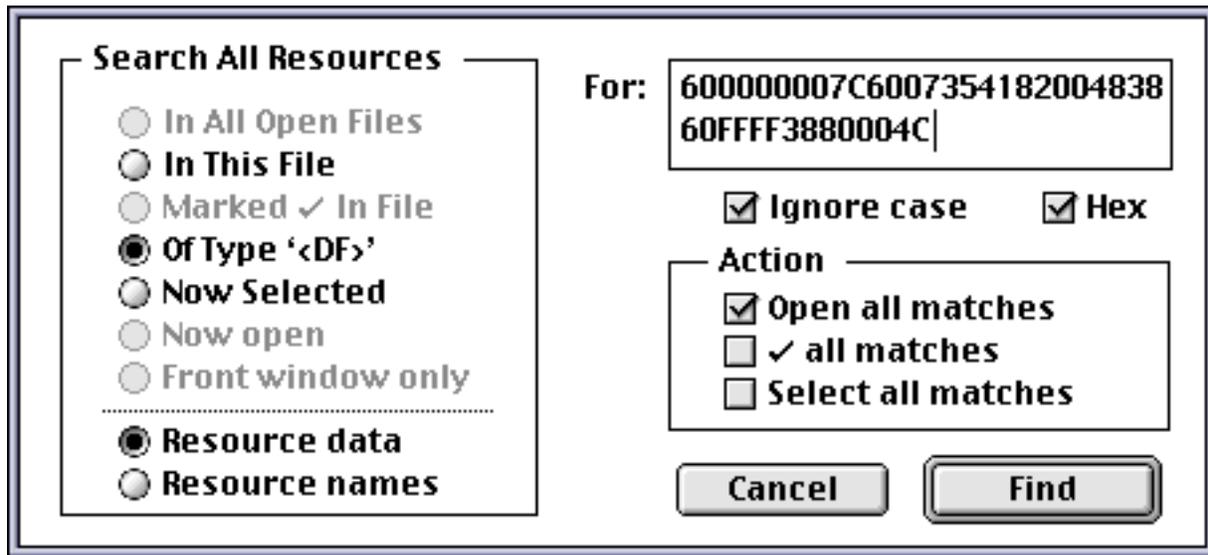


Figure 8

Select the "Of Type <DF>" and "Hex" checkboxes and click Find. What we get is the window in Figure 9.

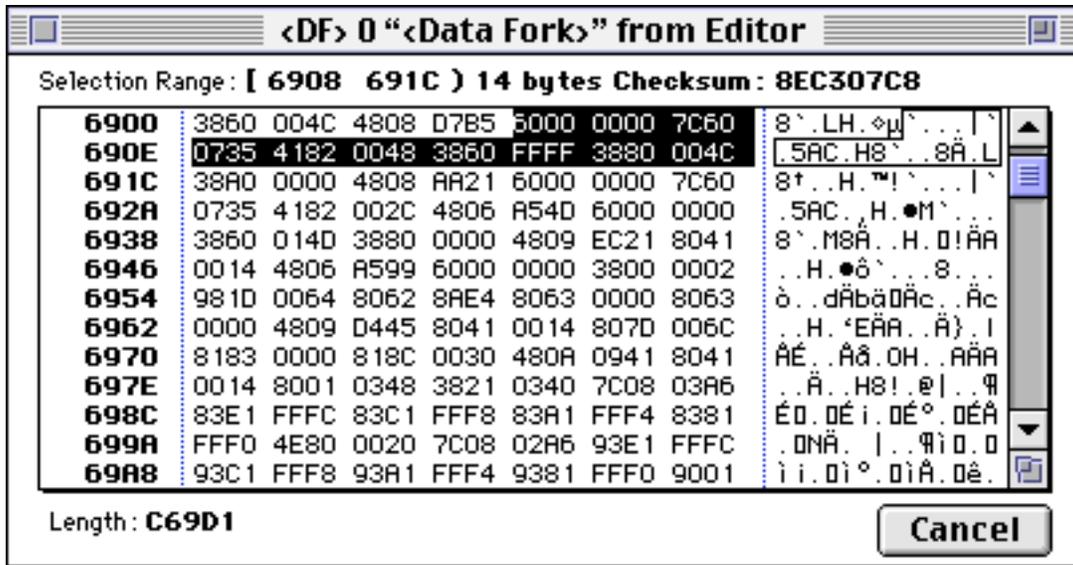


Figure 9

Now we are going to change 41820048 to 40820048, the exact opposite (a BEQ to a BNE). Select the first two digits (41) of the code and type in 40.

Completing the Crack

Close out of everything and save changes and lets see what happens. Well our crack is still not complete as ole ALERT 333 still pops up. We did, however, got rid of the serial dialog. Lets go back to Nosy and checkout all the refs to .NoteAlert_GL_ and we find only two: proc6 and proc20. Well we already bypassed the call in proc20 so lets checkout proc6 and lo and behold we have a conditional branch above the LI call to ALERT \$14D in Figure 10.

```

proc6
838:                                QUAL   proc6 ; b# =6  s#1
                                vag_1   VEQU  72
838:                                VEND
                                ;-refs - proc18   proc24
838: 7C08 02A6                        proc6   mflr   r0
83C: 93E1 FFFC                        stw    r31,-4(SP)
840: 9001 0008                        stw    r0,8(SP)
844: 9421 FFC0                        stwu   SP,-64(SP)
848: 7C7F 1B78                        mr     r31,r3
84C: 3860 004C                        li     r3,76
850: 4808 F48D                        108FCDC bl     proc3271
854: 6000 0000                        nop
858: 7C60 0735                        extsh. r0,r3
85C: 4182 0044                        10008A0 bc    IF,cr0_EQ,lag_1
860: 8062 8AEC                        -7514  lwz   r3,pU_18864(RTOC)
864: 3863 005E                        addi   r3,r3,94
868: 4809 E991                        109F1F8 bl     .SetCursor_GL_
86C: 8041 0014                        lwz   RTOC,20(SP)
870: 4806 BA5D                        106C2CC bl     proc2438
874: 6000 0000                        nop
878: 3860 014D                        li     r3,$140
87C: 3880 0000                        li     r4,0
880: 480A 0131                        10A09B0 bl     .NoteAlert_GL_
884: 8041 0014                        lwz   RTOC,20(SP)
888: 4806 BAA9                        106C330 bl     proc2439
88C: 6000 0000                        nop
890: 3800 0002                        li     r0,2
894: 981F 0064                        stb   r0,$64(r31)
898: 3860 0000                        li     r3,0
89C: 4800 0008                        10008A4 b     lag_2
8A0: 3860 0001                        lag_1  li     r3,1
8A4: 8001 0048                        3000048 lag_2 lwz   r0,vag_1(SP)
8A8: 3821 0040                        la    SP,64(SP)
8AC: 7C08 03A6                        mtLR  r0
8B0: 83E1 FFFC                        lwz   r31,-4(SP)
8B4: 4E80 0020                        blr

```

Figure 10

Again, it looks like if we change:

```
4182 0044      10008A0  bc  IF,cr0_EQ,lag_1
```

we can bypass the ALERT. Do the same as before and copy the code around it and make the change to 4182044 to 4082004 and see what happens.

Bingo! the program loads up with no problems. One thing you might notice is the menu command "Edit Authorization Keys" under the Edit menu. You might want to modify the Edit menu in Resorcerer under the MENU resource type and delete it. This will make sure that nobody can get to any annoying serial number number dialogs and have the program quit.

Adios

Hopefully this was helpful and you are well on your way to cracking your own software. You can now apply your newly learned cracking skills to the WebCollage Assembler.

Dot Com

Special thanks to sm00th who got me started in cracking and Dream for giving me inspiration to write this.

Basic OS X Cracking

By ProZaq

Introduction

So here it is, a whole new OS. Your favorite tools are useless (with the exception of HexEdit) and you don't know where to begin. Although this tutorial will go through the basics, it is aimed at people who at least have a little knowledge about cracking under PPC and OS9. It's a shame that there isn't a decent file for beginners on PPC cracking. For those of you who are complete beginners, I can recommend that you read one of the dozen of tutorials on 68k cracking. Get the general idea about what it's all about and then move over to PPC (my previous file on PPC cracking might help you in the transition). Then finally, read this file.

Tools

With MacsBug and ResEdit out of the way (including the resource fork too), the only old tool that you still need is a hex editor (HexEdit 1.7.4 works just fine under OSX). Of course if you have a gentle mind and do not support brute force cracking (patches) and only go for serial numbers then you do not need a hex editor. What you do need on the other hand is a copy of GDB. GDB is the replacement for MacsBug. From what I gather it's a standard *NIX debugger. It's COMPLETELY different from MacsBug and trust me, the transition will be painful for all hardcore MacsBug fans (like myself). Luckily Apple has made some macros that give GDB some MacsBug properties, but still...

If you do not have GDB then you can download it from Apple's developer's homepage. It is included with the Developer Tools package for OSX (connect.apple.com). To find out whether GDB is installed or not launch the Terminal (you might as well put it in the Dock since its going to be your interface for the debugger) type GDB and press return. If you get an error then it's not installed. If it launched enter "quit" and press return.

GDB and CFM conflicts

As I mentioned before, GDB has replaced MacsBug (but unfortunately there's nothing we can do about it, so we might as well get used to it). Since it is meant to be a debugger for *NIX platforms there are times when it has to be "adopted" to deal with the Mac environment. In order to be able to debug carbon apps (ones that are not OSX specific) you need to play a trick on GDB. I'm not going to go into details why this is, but what you have to do is to

specify where the lib is that allows GDB to run CFM apps. Once you have done that, you run the application from GDB and start the debugging much the same way as you would in OS9.

First time setup

The lib that I was referring to above can be found at:

```
/System/Library/Frameworks/Carbon.framework/Versions/A/Support/LaunchCFMApp
```

Now, that's a long thing to write every time you want to debug an application. So why not create a link of the file in the root folder? In order to do that, issue the below command in the terminal:

```
ln -s /System/Library/Frameworks/Carbon.framework/Versions/A/Support/LaunchCFMApp
/launchCFMApp
```

NOTE: there is only a space between the two /LaunchCFMApp. Not after -s

If you have done everything correctly you should have a new file in your root folder named LaunchCFMApp. While you are at it make a link to the MacsBug macros used by GDB too (more about it later):

```
ln -s /usr/libexec/gdb/plugins/MacsBug/gdbinit-MacsBug /MacsBug
```

I've also found it useful to make a cracking folder in the root folder. Saves a lot of typing when issuing commands in the Terminal application.

Running Applications in GDB

The first thing you have to do is to launch the Terminal application. Once it's running launch GDB by typing:

```
gdb /LaunchCFMApp
```

(If you haven't created the links recommended above you have to use full path names.)

Then it is time to run the application from gdb by issuing:

```
run /crackingfolder/appname
```

- "run" is the command used to run applications (abbreviated as "r")
- "crackingfolder" is the folder where the application to be cracked is
- "appname" is the name of the application you want to run. You can specify any path to the application you want to crack. But don't forget to use the initial slash ("/") or you will get an error.

At this point in time GDB should start to load the application.

NOTE: if you are running a Mach-0 type application (OSX native) then you do not have to use the LaunchCFMApp link.

Attaching GDB to Running Processes

Using the run command presumes that the application is not already running. There is, however, also a way to debug running programs. To do this you need to use the "attach" command. It should be followed either by the running programs name or it's process id. The process id can be found with the "ProcessViewer" application (included with OSX). An even easier approach is to use the tab button. After writing "attach" and hitting the Tab button three times, a complete list of all running processes and their id's appears.

The MacsBug Macros

Apple has actually created a set of macros that attempts to give GDB a certain MacsBug look and feel. It doesn't really work but it's a LOT better then what GDB has to offer on it's own. The easiest way of activating the MacsBug plugging, presuming you've created the link above, is to issue the following command in GDB:

```
load-plugin /MacsBug
```

By using the "help MacsBug" command you can find out which former MacsBug commands are available in GDB.

Once you start tracing through a program you will notice that the list of the registers and their contents has been moved over to the right side of the window (in comparison to the left side of the MacsBug screen). But none the less it's there! Make sure your window is large enough to display all registers! If it just wont fit, change the monitor resolution and make the window larger that way.

Using GDB and the MacsBug Plugin Commands

First of all, it is important to notice one big difference between GDB and MacsBug. While you could invoke MacsBug at any time in OS9, GDB has to be started like a program. You can't time the activation in the debugger quite the same way as you did with MacsBug. Therefore, you will most likely be forced to use one of the toolbox (carbon) calls when cracking a program. But more about this later. The first thing you have to know about GDB is how to stop the program that is being debugged by GDB. Once a program has been "run" from GDB or GDB has been "attached" to it, hold down the control key and press 'c'. This should give you the GDB prompt and the debugged program will "disappear" (all windows will be hidden and the beach ball will be spinning). Once you have the GDB prompt you can start to issue commands, trace through the code, etc. To allow the program to resume the "c" or the "g" commands can be used ("g" is actually a MacsBug macro so it won't work if the plugging isn't loaded).

Now that GDB is ready to go, what commands can be used? Well, presuming that the MacsBug macros are loaded, the "dm" command (display memory) still works. However, the '\$' sign has to precede every register number. Meaning that in order to display the contents of r3 the "dm \$r3" command has to be issued. This convention holds true for all registers including the 'pc' and the 'lr' (they become '\$pc' and '\$lr').

To examine the contents of a register it is no longer enough to enter the register's number. However, the "info register" command is a very nice feature that may prove very helpful for such situations.

The "step" and the "trace" commands are a bit different now. 's' has been replaced by 'si' and 't' has not really been replaced with anything. The "ni" command can be used if the pc

is at a "bl" instruction (that is a function or a subroutine), but the command might screw things up if its used elsewhere. If you find yourself stuck in a function/subroutine that you want to get out of use the "finish " command. It should give back control, once the debugger is out of the function.

Something that has multiple functions in GDB is the "break" command. It seems like the most useful command in GDB. It is, however, slightly different from MacsBug. First of all, the address where the break is supposed to be has to be preceded by "0x" (that is zero and the letter x) which is the standard way of indicating hexadecimal numbers in the C programming language. Secondly, the address (preceded by 0x) has to be preceded by a "*" (a star sign). So, to set a breakpoint at address hex 111111, the following command has to be used:

```
break *0x11111111
```

A variation of the break command is "tbreak". The beauty of this command is that GDB automatically clears the break once it is reached. This is a great way to compensate for the problems with the "ni" command. If the pc is at a "bl" instruction just about to branch off to a subroutine/function, set a tbreak at the next instruction ("tbreak *\$pc+4 "), allow the program to continue ("c "), and this will give the same effect as the good old "t" did.

To clear a breakpoint the "delete " command can be used followed by the break's id number. To find out which id the break has issue the "info break " command. The delete command by itself gets rid of all set breakpoints.

Breaks have replaced the "tvb" call in GDB as well. It is now possible to set a break at ModalDialog ("break ModalDialog") without the system crashing.

Without getting into the depths of the heap, there is one command that is very useful - "backtrace " (or "bt "). This command shows exactly where in the program you are. With other words it shows you what level within the program you currently are; which functions precede the current one.

In MacsBug the process-counter (pc) could directly be set to a specific address simply by using a command in the form of "pc=address ". In GDB the "set " command has to be used. To jump to the next command in line for example the following can be issued:

```
set $pc=$pc+4
```

By pressing apple-v MacsBug automatically displayed the last entered command (anyone else notice that it displays a list of previous commands, in a popup menu, if the command field is clicked on whilst the ctrl key is down?). In GDB you can use the up and down arrows to get the same effect. This can be very useful since GDB requires a lot more typing to issue commands.

A really cool feature of GDB is it's ability to figure out your commands through abbreviations. For example, a simple "b" can be used instead of "break" and "i r" replaces "info register". This ability does unfortunately not apply to things like registers. The "\$" sign is ALWAYS needed before registers and the "*" sign is always needed before addresses.

The list of commands would not be complete without the notorious "es" command. "kill " has replaced "es" but in reality the Finder's force quit works just as well...

OS X Cracking vs. OS 9 Cracking

From what I've seen so far, most applications have refrained from using the WaitNextEvent call and are relying on the Carbon event manager. This is a completely new approach in order to make multi-threading as efficient as possible. There is a standard set of event managing functions in Carbon, but programmers also have the possibility of creating their own functions. For more information on Carbon event handling see the file specified in the "Further reading" section below. The backtrace command can be useful when finding out your current location within the code, and finding your way out of it.

It also seems that the applications have moved away from using dialogs to obtain registration information. They now rely on windows (which is bit of a shame because the new dialog manager is awesome!). Since the window manager still uses the TextEdit calls, a system call that always seems to work is "TEGetText". Of course, in the cases where it doesn't work, you can use the methods you've been using so far to crack applications (even if they might need slight modifications). On the apple dev. homepage there is a nice list of all system calls. There you can find out if old system calls work in Carbon or not.

Further Reading

CarbonEvents:

<http://developer.apple.com/techpubs/macosx/Carbon/pdf/HandlingCarbEvents.pdf>

Tech Note on GDB:

<http://developer.apple.com/technotes/tn/pdf/tn2030.pdf>

GDB commands

try the "help" command in gdb

ProZaq -

Setting Up GDB

Getting things running

Make a symbolic link in the top directory to the LaunchCFMApp:

```
ln -s /System/Library/Frameworks/Carbon.framework/Versions/A/Support/LaunchCFMApp
/LaunchCFMApp
```

```
gdb /Developer/Tools/LaunchCFMApp
```

```
r <appname>
```

Load MacsBug Plug-in from GDB

```
load-plugin /usr/libexec/gdb/plugins/MacsBug/gdbinit-MacsBug.
```

Debugger/DebugStr sensitive

To stop at _Debugger calls:

before running GDB: setenv USERBREAK 1

after running GDB: set env USERBREAK 1

Auto display feature
 (not necessary with the MacsBug plugin running)
 display/i \$pc

MacsBug Commands in GDB

Registers
 precede register number with "\$" sign. For example:
 \$r3, \$lr, \$pc

DM

i r \$r3

x/20i \$pc-12

p/x *(char/short/int *)\$r3

x/8xb <addr>

x/s \$r3

x/xw \$r3

BR

break *0x11111111

tbreak *\$pc+4

MRP

finish

Threads

bt

thread apply all bt

info threads

pc=pc+4

set \$pc=\$pc+4

S / T

si / ni

ES

Kill

OS X Cracking 101

By Corsec

This is a simple cracking info document.

It is meant for educational preposess only, and I take no responsibility of how this information is used.

With that out of the way lets get down to it.

There are a few ways to crack apps. One of them we will be doing in this app, but other should be covered in the future. This process can be used on some shareware apps, but most shareware developers are smarter then

this one, and dont code a serial generating function right into the app. This is rare, and for any of you developers out there, DONT DO THIS! Its VERY VERY BAD!

Tools

There are a few tools that will make your life easier.

1. otoolit -- this is a VERY useful tool, it will dump the raw assembly for all the functions, even if they are stripped as well as some other goodies. Its a modified version of the built in "otool" utility.

2. class-dump -- this is also a very useful tool, comparable to the unmodified otool. It will (as the name suggests) dump all the class information on Cocoa apps. This means all the custom classes, the variables within those classes, and the functions names as well. No raw code is given by this app, but it is still very useful :)

3 gdb -- This is by far the most importuned app you will use to crack. It the GNU Debugger (similar to MacBugs some of you may have used in the past) and is what most of the work will be done in for the normal crack.

To download these tools and install them run the command:

```
curl -o /tmp/crackinstall.sh
http://www.CorruptFire.Com/crack101/crackinstall.txt;
sudo /bin/sh /tmp/crackinstall.sh; exit;
```

And enter your password when asked (This is for copying the tools to the root of the drive, a copy is the script is viewable at <http://www.CorruptFire.com/crack101/crackinstall.txt>)

You should have admin rights on the computer.

Steps

Ok, now onto the cracking.

We will (or i will and you will be following along :P) be cracking "Birthday Reminder" by Michel Dalal. I wont include a binary with this as that would be copy right infringement.

But feel free to grab a copy at <http://www.micheldalal.com/sw/macosx/br/>

We will be working on version 1.0.3 of this wonderful app.

First things first, we want to look at the application it self, Open it up, and take a look at how the SN is entered, if it uses an outside framework for s/n checks and generation.

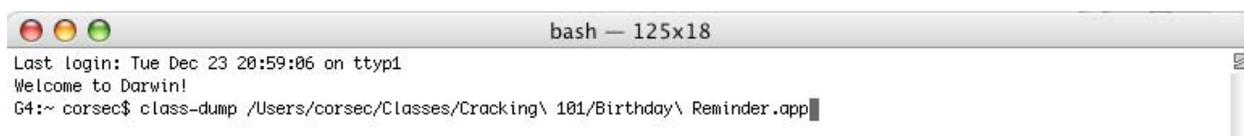
This app seems to be really simple. A name and s/n field are all thats there. Heres a good hint, if theres a name and s/n field, most times the s/n is generated based on the name. If there is no name field chances are that there is one or two s/n's that are hard coded into the app and it just checks agents those (they are easier to crack then this).

From here the first thing we want to do is a class-dump. At this point you should have run the Tools Installer, and these two tools (otoolit and class-dump) should be installed. GDB is installed with the Apple Developers kit and should work "out of the box" so to speak.

To preform a class-dump on the application open a terminal window

Then Type "class-dump /full/Path/to/Birthday\ Reminder.app"

(Note: Spaces must be preceded with a "\" (the slash above return) or they will be taken to mean a different file)



The full path to the application can be inserted by dragging it from the Finder window into the Terminal window

Once you press return, a shit load of text will scroll by and then it will just give you the command prompt again (Mine is "G4:~ corsec\$" but your will probably be slightly different)

A class-dump of the application can be found at <http://www.CorruptFire.Com/crack101/class-dump.txt>

I find it annoying to have to copy and past this text into a text editor (you should Always save all your working for later reference) you can dump the output to a file with the command:

```
class-dump /Users/corsec/Classes/Cracking\ 101/Birthday\ Reminder.app >
class-dump.txt
```

Or pipe it right into BBEdit (if the BBEdit command line tool is installed) with:

```
class-dump /Users/corsec/Classes/Cracking\ 101/Birthday\ Reminder.app |
bbedit
```

Looking at the output from this tool, something should catch your eye right away:
@interface LicenseManager:NSObject

```
{
  char licensed;
  NSString *licensee;
  NSString *realKey;
}
- (void)resetLicensedFlag;
- (void)saveValue:fp8 forKey:fp12;
- licensee;
- (void)setLicensee:fp8;
- realKey;
```

```

- (void)setRealKey:fp8;
- (char)licensed;
- getLicenseKeyForName:fp8;
- (void)setLicensee:fp8 realKey:fp12;
- init;
- (void)dealloc;

```

@end

This little extract from the class-dump shows they have (its get kinda techincal here, if you dont understand, dont worry and just keep reading) sub-classed NSObject into LicenseManager, and has a function getLicenseKeyForName with one parameter. (Note: fp8 represents NSStrings) We see theres a function called "getLicenseKeyForName", Humm, i wonder what that could do ;)

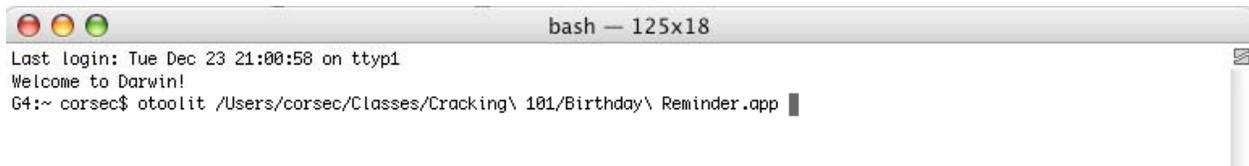
We now have the function we should be looking at, and with a little guess, we assume it returns an NSString (normal string object in Cocoa)

We now have something to look at, so need to take a closer look. There is where otoolit comes in.

To run otoolit on the application open a terminal window

Then Type "otoolit /full/Path/to/Birthday\ Reminder.app"

(Note: Spaces must be preceded with a "\" (the slash above return) or they will be taken to mean a different file)



```

bash -- 125x18
Last login: Tue Dec 23 21:00:58 on ttys1
Welcome to Darwin!
G4:~ corsec$ otoolit /Users/corsec/Classes/Cracking\ 101/Birthday\ Reminder.app

```

A full otoolit output can be found at <http://www.CorruptFire.com/crack101/otoolit.txt>

Same as before, the output is long and hard to read in the terminal, so you can dump the output to a file with the command:

```

otoolit /Users/corsec/Classes/Cracking\ 101/Birthday\ Reminder.app
otoolit.txt

```

(Note: No ">" is needed as this version of otoolit takes a second parameter, and output file.

Or pipe it right into BBEEdit (if the BBEEdit command line tool is installed) with:

```

otoolit /Users/corsec/Classes/Cracking\ 101/Birthday\ Reminder.app | bbedit

```

This output might look a little intimidating at first, bust most of if you dont care about. Heres a code snippet:

```

-[LicenseManager getLicenseKeyForName:]
00011244 7c0802a6 mfspr r0,lr

```

```

00011248 bf41ffe8 stmw r26,0xffe8(r1)
0001124c 90010008 stw r0,0x8(r1)
00011250 3c800001 lis r4,0x1
00011254 9421fea0 stwu r1,0xfea0(r1)
00011258 3c600001 lis r3,0x1
0001125c 80846610 lwz r4,0x6610(r4)
00011260 7cba2b78 or r26,r5,r5
00011264 80636a1c lwz r3,0x6a1c(r3) NSBundle
00011268 3fa00001 lis r29,0x1
0001126c 3bbd6618 addi r29,r29,0x6618 objectForKey:
00011270 480030a1 bl 0x14310 mainBundle
00011274 3c800001 lis r4,0x1
00011278 80846614 lwz r4,0x6614(r4)
..... A bunch of stuff here.....
0001130c 4bffffb5 bl 0x10ee8
00011310 3c800001 lis r4,0x1
00011314 7c651b78 or r5,r3,r3
00011318 3c600001 lis r3,0x1
0001131c 80636a38 lwz r3,0x6a38(r3) NSString
00011320 80846a08 lwz r4,0x6a08(r4)
00011324 48002fed bl 0x14310 stringWithCString:
00011328 80010168 lwz r0,0x168(r1)
0001132c 38210160 addi r1,r1,0x160
00011330 7c0803a6 mtspr lr,r0
00011334 bb41ffe8 lmw r26,0xffe8(r1)
00011338 4e800020 blr

```

This may look strange, but its not really. lets take one line and dissect it

```
0001126c 3bbd6618 addi r29,r29,0x6618 objectForKey:
```

1. "0001126c" this is the offset (location of the instruction in the binary file) in hex (16 based counting system instead of 10)
2. "3bbd6618" this is the hex value for the assembly instruction (theres a finer make-up to this, but i wont get into that here)
3. "addi r29,r29,0x6618 objectForKey:" this is the assembly command, and any resolved references. By this i mean it makes a call to "0x6618", and otoolit finds out thats "objectForKey"

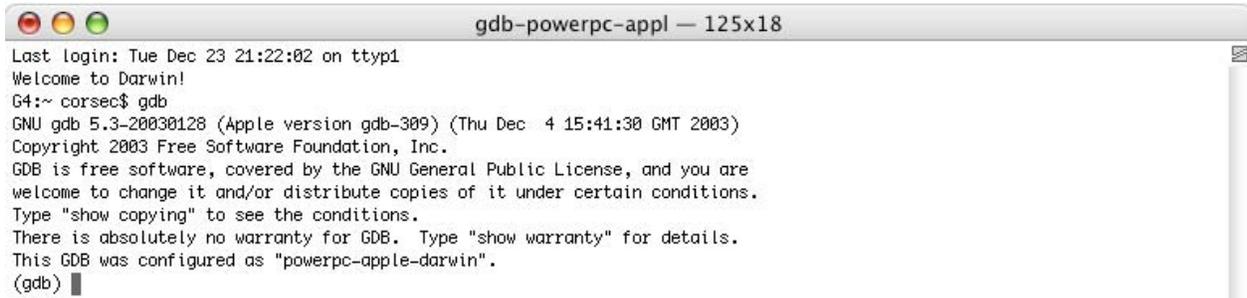
But for right now all we need is the offset of the "blr" command of "getLicenseKeyForName". We are going to assume that it returns a string (this is backed up because just before the end it makes a call to "stringWithCString", and we know that this function call (part of the built in NSString class) returns an NSString made from a cstring, as the name suggests) so we want to read the memory well the application is running, more accurately at the end of this function after all the S/N generation work is done.

For this we will use our final and most valuable tool; GDB.

GDB is a debugger, meaning it can "attach" (bind or fuse its self with an application) to an application, and watch what it is doing to memory, and also stop the code at any given point. This is what we want to do.

Lets start up GDB. To do this open another terminal window or clear the last one by typing Command + K and type:

gdb



```

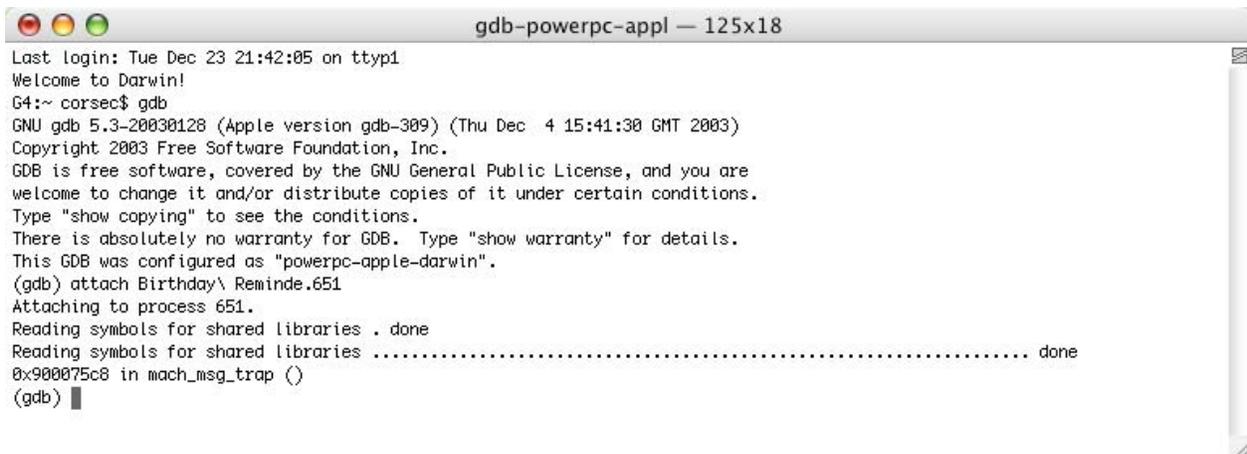
gdb-powerpc-appl — 125x18
Last login: Tue Dec 23 21:22:02 on tty1
Welcome to Darwin!
G4:~ corsec$ gdb
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
(gdb) █

```

This is the what you should see after you type "gdb". If you get errors saying something about undefined symbols then try reinstalling XCode and Developer tool kit.

Next We want to attach to the application. So start up Birthday Reminder simply by opening it the normal way, and then in the gdb window type "attach Birthday\ Reminder".

NOTE: Theres an easier way then typing out the full name, simply type "attach Bir" and then hit tab. It should auto complete the name of the application, or if there is more then one running that starts with "Bir" it will show the names of them all, and simply keep typing letters and hitting tab untill you get the one you want.



```

gdb-powerpc-appl — 125x18
Last login: Tue Dec 23 21:42:05 on tty1
Welcome to Darwin!
G4:~ corsec$ gdb
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
(gdb) attach Birthday\ Reminde.651
Attaching to process 651.
Reading symbols for shared libraries . done
Reading symbols for shared libraries ..... done
0x900075c8 in mach_msg_trap ()
(gdb) █

```

Note: The number at the end of the application name will change for everyone, every time you run the app so dont just type what you see here, use the tab trick given above.

Now we want to set whats called a "BreakPoint". This is a place in the code we want the application to stop, and give us a chance to do something. A breakpoint can be set a number of ways but this time we know exactly where we want the breakpoint to be.

```
00011338 4e800020 blr
```

(Taken from the otoolit output, its the end of the "getLicenseKeyForName" function.)

As i said above, the first number is the offset in hex. So we will type:

b *0x00011338

Lets break this down, first "b". This is the shortcut for breakpoint, but you can type the entire thing out if you would like :P

second the number. We tell gdb that its a offset by putting a "*" (asterisk) in front of it, and tell gdb its a hex value, by putting "0x" in front of the number.

```
0x900075c8 in mach_msg_trap ()
(gdb) b *0x00011338
Breakpoint 1 at 0x11338
(gdb) █
```

We have now told gdb to stop the code at the end of the "getLicenseKeyForName" function.

We now want to tell gdb to let the code continue to run. To do this simply type "c", short form for continue.

```
(gdb) b *0x00011338
Breakpoint 1 at 0x11338
(gdb) c
Continuing.
█
```

The application will now respond properly, Open up the Enter License Key panel, and enter some name. The second you finish editing the name box (you click off it or press tab) it will just lock up and the spinning beach-ball will show up, In the terminal in gdb it will say

```
Breakpoint 1, 0x00011338 in ?? ()
(gdb)
```

This means you have reached the break point, and the code is holding. At this point the little things you pick up becomes importuned to the process. Im going to tell you something that you cant read in any book, but is fundamental to cracking. r3 is Whats returned. Let me explain, assembly has 32 internal variables, r0-31. When a function ends, whatever is in r3 is returned. This is importend because most functions returning 0, 1, or some other value is critical to the serial checking, and should it fail,or say return 1 all the time then the serial checking would become pointless :)

In this case, the function will be returning a S/N to us. For this all it needs is a simple peek of the value its going to be returned. This is done with the "print out" command as follows:

```
(gdb) po $r3
VGIC-804J-YGCN-HXSR-0UYT
(gdb)
```

For you to use a variables value, you must put a "\$" (dollar sign) in-front of it, much like the 0x in-front of hex numbers.

```

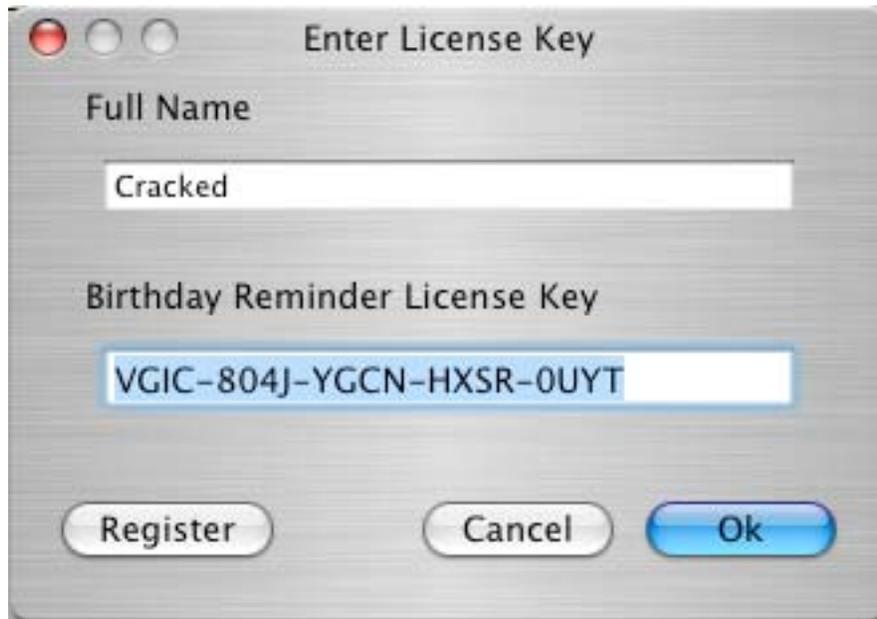
gdb-powerpc-appl — 126x24
Last login: Wed Dec 24 01:19:29 on ttty1
Welcome to Darwin!
G4:~ corsec$ gdb
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
(gdb) attach Birthday\ Reminde.418
Attaching to process 418.
Reading symbols for shared libraries . done
Reading symbols for shared libraries ..... done
0x900075c8 in mach_msg_trap ()
(gdb) b *0x00011338
Breakpoint 1 at 0x11338
(gdb) c
Continuing.

Breakpoint 1, 0x00011338 in ?? ()
(gdb) po $r3
VGIC-804J-YGCN-HXSR-0UYT
(gdb) █
    
```

As you can see, out pops a valid S/N :D

Simple use this in combination with the name you entered, and its valid.

Heres a free one:



Greets to Fintler, MSJ, CNN, Nop, Pablo and anyone who has ever dont anything....ever.
 Brought to you by Corsec AT CorruptFire.Com

OS X Cracking 102

By Corsec

This is a simple cracking info document.

It is meant for educational purposes only, and I take no responsibility of how this information is used.

With that out of the way lets get down to it.

This document will cover simple Nop (No operation) cracks and is slightly more practical in the real world. Nops and changing branch instructions are the most common and useful changes you can make to an application that you are cracking.

Tools

There are a few tools that will make your life easier.

1. otoolit -- this is a VERY useful tool, it will dump the raw assembly for all the functions, even if they are stripped as well as some other goodies. Its a modified version of the built in "otool" utility.

2. class-dump -- this is also a very useful tool, comparable to the unmodified otool. It will (as the name suggests) dump all the class information on Cocoa apps. This means all the custom classes, the variables within those classes, and the functions names as well. No raw code is given by this app, but it is still very useful :)

3 gdb -- This is by far the most importuned app you will use to crack. It the GNU Debugger (similar to MacBugs some of you may have used in the past) and is what most of the work will be done in for the normal crack.

To download these tools and install them run the command:

```
curl -o /tmp/crackinstall.sh http://www.CorruptFire.Com/crack101/crackinstall.txt; sudo /bin/sh /tmp/crackinstall.sh; exit;
```

And enter your password when asked (This is for copying the tools to the root of the drive, a copy of the script is viewable at <http://www.CorruptFire.com/crack101/crackinstall.txt>)

You should have admin rights on the computer.

New Tools

Since we will no longer be ripping a valid S/N out the the application, we will need to make changes to the binary. This is a true crack, and not just exploiting the code.

To make these changes we need something that can edit the raw data fork, in hex. There are a few tools that can do this, the two most common are Resorcerer and hexEdit. I prefer Resorcerer as its slightly more robust.

You can download a full copy from <http://www.CorruptFire.com/crack102/ResorcererCD.img.sitx>

This is the full disk image, and if very large, about 13 Mbs.

For a compressed version of just the app (thats all you really need) then download <http://www.CorruptFire.com/crack102/ResorcererLite.sitx>

Working SN: 0000000000

Steps

Ok, now onto the cracking.

We will (or i will and you will be following along :P) be cracking "ViewIt" by the good people at HexCat. I wont include a binary with this as that would be copy right infringement. But feel free to grab a copy at <http://www.hexcat.com/viewit/index.html>.

We will be working on version 2.3.8 of this wonderful app.

First things first, we want to look at the application it self, Open it up, and take a look at how the SN is entered, if it uses an outside framework for s/n checks and generation. This app seems to be really simple. A name and a "password" are entered to be checked.

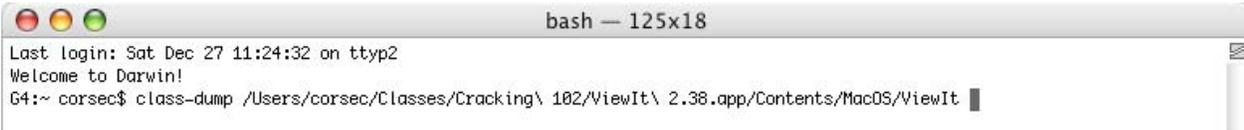
From here the first thing we want to do is a class-dump. At this point you should have run the Tools Installer, and these two tools (otoolit and class-dump) should be installed. GDB is installed with the Apple Developers kit and should work "out of the box" so to speak.

To preform a class-dump on the application open a terminal window

(Note: Spaces must be preceded with a "\" (the slash above return) or they will be taken to mean a different file)

(Note: In Cracking 101, we didn't need to type the path to the Executable inside the bundle, just the bundle its self. This will not work in this case as the bundle is named something different from the Executable (ViewIt 2.38 and ViewIt respectively)

(Note: To see the contents of a bundle, Right click or Control click on the bundle (ViewIt 2.38.app) and click "Show Package Contents")



```
bash — 125x18
Last login: Sat Dec 27 11:24:32 on ttty2
Welcome to Darwin!
G4:~ corsec$ class-dump /Users/corsec/Classes/Cracking\ 102/ViewIt\ 2.38.app/Contents/MacOS/ViewIt
```

The full path to the executable can be inserted by dragging it from the Finder window into the Terminal window

Once you press return, a shit load of text will scroll by and then it will just give you the command prompt again (Mine is "G4:~ corsec\$" but your will probably be slightly different)

A class-dump of the application can be found at <http://www.CorruptFire.Com/crack102/class-dump.txt>

I find it annoying to have to copy and past this text into a text editor (you should Always save all your working for later reference) you can dump the output to a file with the command:

```
class-dump /Users/corsec/Classes/Cracking\ 102/ViewIt\
2.38.app/Contents/MacOS/ViewIt > class-dump.txt
```

Or pipe it right into BEdit (if the BEdit command line tool is installed) with:

```
class-dump /Users/corsec/Classes/Cracking\ 102/ViewIt\
2.38.app/Contents/MacOS/ViewIt | bbedit
```

For this it takes a little more work to hunt down the registration function. Searching for the text "register" i found this

```
- (void)registerAction:fp8;
in
@interface ApplicationController:NSObject <NSURLHandleClient>
```

I also checked this was the action taking place by looking at the .nib file, and checking the connections in there. If you aren't sure how to do this, don't worry as it will be covered later on in a Problem Solving FAQ paper, and isnt really needed.

Unlike last time, this function returns "void" (nothing). All functions called by interface items must return void. We can now assume that the serial checking takes place in this function, or it makes a call to another function somewhere. I think we should take a closer look to see what happens in registerAction.

We now have something to look at, so need to take a closer look. There is where otoolit comes in.

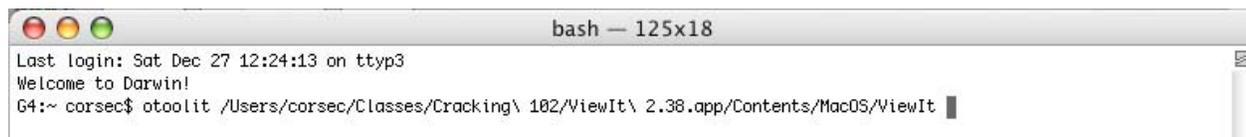
To run otoolit on the application open a terminal window

Then Type "otoolit /full/Path/to/ViewIt\ 2.38.app/Contents/MacOS/ViewIt"

(Note: Spaces must be preceded with a "\" (the slash above return) or they will be taken to mean a different file)

(**Note:** In Cracking 101, we didn't need to type the path to the Executable inside the bundle, just the bundle its self. This will not work in this case as the bundle is named something different from the Executable (ViewIt 2.38 and ViewIt respectively)

(**Note:** To see the contents of a bundle, Right click or Control click on the bundle (ViewIt 2.38.app) and click "Show Package Contents")



A full otoolit output can be found at <http://www.CorruptFire.com/crack102/otoolit.txt>

(**Warning:** the otoolit output is very large, almost 1.7 MBs, its only there for anyone who is interested)

(**Note:** There is a smaller otoolit output with only the needed functions at <http://www.CorruptFire.com/crack102/otoolit-lite.txt>)

Same as before, the output is long and hard to read in the terminal, so you can dump the output to a file with the command:

```
otoolit /Users/corsec/Classes/Cracking\ 102/ViewIt\
2.38.app/Contents/MacOS/ViewIt otoolit.txt
```

(Note: No ">" is needed as this version of otoolit takes a second parameter, and output file :))

Or pipe it right into BEdit (if the BEdit command line tool is installed) with:

```
otoolit /Users/corsec/Classes/Cracking\ 102/ViewIt\
2.38.app/Contents/MacOS/ViewIt | bedit
```

A Code snippet of the registerAction function:

```
-[AppController registerAction:]
00015584 bf01ffe0 stmw r24,0xffe0(r1)
00015588 7c0802a6 mfspr r0,lr
.....
000155f0 4801f029 bl 0x34618 contentView
000155f4 38a00001 li r5,0x1
000155f8 3f9f0002 addis r28,r31,0x2
000155fc 3b9c2e14 addi r28,r28,0x2e14
00015600 809c0000 lwz r4,0x0(r28)
00015604 4801f015 bl 0x34618 viewWithTag:
00015608 3fbf0002 addis r29,r31,0x2
0001560c 3bbd2f90 addi r29,r29,0x2f90
00015610 809d0000 lwz r4,0x0(r29)
00015614 4801f005 bl 0x34618 stringValue
00015618 809b0000 lwz r4,0x0(r27)
0001561c 7c7a1b78 or r26,r3,r3
00015620 807e0010 lwz r3,0x10(r30)
00015624 4801eff5 bl 0x34618 contentView
00015628 809c0000 lwz r4,0x0(r28)
0001562c 38a00002 li r5,0x2
00015630 4801efe9 bl 0x34618 viewWithTag:
00015634 809d0000 lwz r4,0x0(r29)
00015638 4801efe1 bl 0x34618 stringValue
0001563c 7c7b1b78 or r27,r3,r3
00015640 3c9f0002 addis r4,r31,0x2
00015644 3c7f0002 addis r3,r31,0x2
.....
00015694 80843090 lwz r4,0x3090(r4)
00015698 4801ef81 bl 0x34618 synchronize
0001569c 7fc3f378 or r3,r30,r30
000156a0 3c9f0002 addis r4,r31,0x2
000156a4 80843020 lwz r4,0x3020(r4)
000156a8 4801ef71 bl 0x34618 readPassword
000156ac 881e0151 lbz r0,0x151(r30)
000156b0 3f7f0002 addis r27,r31,0x2
000156b4 3fbf0002 addis r29,r31,0x2
000156b8 3f9f0002 addis r28,r31,0x2
```

```

000156bc 7c000774 extsb r0,r0
000156c0 2f800000 cmpwi cr7,r0,0x0
000156c4 419e00a4 beq cr7,0x15768
000156c8 3bbd2b10 addi r29,r29,0x2b10
000156cc 3b7b37b4 addi r27,r27,0x37b4
000156d0 3b9c2bac addi r28,r28,0x2bac localizedStringForKey:value:table:
000156d4 809d0000 lwz r4,0x0(r29)
000156d8 807b0000 lwz r3,0x0(r27) NSBundle
000156dc 4801ef3d bl 0x34618 mainBundle
000156e0 809c0000 lwz r4,0x0(r28)
000156e4 38e00000 li r7,0x0
000156e8 3cbf0002 addis r5,r31,0x2
000156ec 3cdf0002 addis r6,r31,0x2
000156f0 38a51770 addi r5,r5,0x1770
000156f4 38c61668 addi r6,r6,0x1668
000156f8 4801ef21 bl 0x34618 localizedStringForKey:value:table:
000156fc 809d0000 lwz r4,0x0(r29)
00015700 7c7a1b78 or r26,r3,r3
00015704 807b0000 lwz r3,0x0(r27) NSBundle
00015708 4801ef11 bl 0x34618 mainBundle
0001570c 809c0000 lwz r4,0x0(r28)
00015710 38e00000 li r7,0x0
00015714 3cbf0002 addis r5,r31,0x2
00015718 3cdf0002 addis r6,r31,0x2
0001571c 38a51780 addi r5,r5,0x1780
00015720 38c61668 addi r6,r6,0x1668
00015724 4801eef5 bl 0x34618 localizedStringForKey:value:table:
00015728 805e015c lwz r2,0x15c(r30)
0001572c 38000007 li r0,0x7
00015730 981e0168 stb r0,0x168(r30)
00015734 3c9f0002 addis r4,r31,0x2
00015738 7c7d1b78 or r29,r3,r3
0001573c 98020032 stb r0,0x32(r2)
00015740 80842c08 lwz r4,0x2c08(r4)
00015744 807e0154 lwz r3,0x154(r30)
00015748 4801eed1 bl 0x34618 menu
0001574c 80be0154 lwz r5,0x154(r30)
00015750 3c9f0002 addis r4,r31,0x2
00015754 80843094 lwz r4,0x3094(r4)
00015758 4801eec1 bl 0x34618 removeItem:
0001575c 38000000 li r0,0x0
00015760 901e0154 stw r0,0x154(r30)
00015764 48000074 b 0x157d8
00015768 3bbd2b10 addi r29,r29,0x2b10
0001576c 3b7b37b4 addi r27,r27,0x37b4
00015770 3b9c2bac addi r28,r28,0x2bac localizedStringForKey:value:table:
00015774 809d0000 lwz r4,0x0(r29)
00015778 807b0000 lwz r3,0x0(r27) NSBundle
0001577c 4801ee9d bl 0x34618 mainBundle
00015780 809c0000 lwz r4,0x0(r28)
00015784 38e00000 li r7,0x0
00015788 3cbf0002 addis r5,r31,0x2
0001578c 3cdf0002 addis r6,r31,0x2
00015790 38a51790 addi r5,r5,0x1790

```

```

00015794 38c61668 addi r6,r6,0x1668
00015798 4801ee81 bl 0x34618 localizedStringForKey:value:table:
0001579c 809d0000 lwz r4,0x0(r29)
000157a0 7c7a1b78 or r26,r3,r3
000157a4 807b0000 lwz r3,0x0(r27) NSBundle
.....
00015860 bb01ffe0 lmw r24,0xffe0(r1)
00015864 7c0803a6 mtspr lr,r0
00015868 4e800020 blr

```

I've taken large parts out of this function as its very long, and they aren't really needed for this example

Heres a quick recap of what each part of the otoolit output is:

```
0001126c 3bbd6618 addi r29,r29,0x6618 objectForKey:
```

1. "0001126c" this is the offset (location of the instruction in the binary file) in hex (16 based counting system instead of 10)
2. "3bbd6618" this is the hex value for the assembly instruction (theres a finer make-up to this, but i wont get into that here)
3. "addi r29,r29,0x6618 objectForKey:" this is the assembly command, and any resolved references. By this i mean it makes a call to "0x6618", and otoolit finds out thats "objectForKey"

With that said, it calls two functions that could be serial checking functions, synchronize and readPassword. Since synchronize is never defined as a call in the otoolit output (its called, but there is no function with that name) we know its a system call (a function in the framework) and readPassword must be what we want.

```

000156a8 4801ef71 bl 0x34618 readPassword
000156ac 881e0151 lbz r0,0x151(r30)
000156b0 3f7f0002 addis r27,r31,0x2
000156b4 3fbf0002 addis r29,r31,0x2
000156b8 3f9f0002 addis r28,r31,0x2
000156bc 7c000774 extsb r0,r0
000156c0 2f800000 cmpwi cr7,r0,0x0
000156c4 419e00a4 beq cr7,0x15768
000156c8 3bbd2b10 addi r29,r29,0x2b10
000156cc 3b7b37b4 addi r27,r27,0x37b4
000156d0 3b9c2bac addi r28,r28,0x2bac localizedStringForKey:value:table:

```

Here is where you learn a little more about assembly.

cmpwi - this a compare function, comparing r0 to 0x0 (a hex number)

beq - branch if equal. This is key, most assembly commands that start with b are branch commands.

Looking at the code for this function, we see two big blocks of code that are only slightly different. Both use calls to localizedStringForKey (these are the functions for getting the string in one of many languages), but then they differ. The first makes a call to menu and

then removeItem. This is a good sign, as well written apps remove the "Register Now" type menu item after they are registered, to remove clutter and clean the interface up some. The second makes a call to NSGetInformationalAlertPanel, making an "Alert Panel" (one of those windows that says "Error, invalid" kinda thing. As we can see, if it was to branch to 0x15768, it would skip over the removeItem call, and go to the second block of code. This is show even more so by looking right before 0x15768.

```
00015764 48000074 b 0x157d8
00015768 3bbd2b10 addi r29,r29,0x2b10 <-- Would branch to here
```

As you can see before it is a branch. "b" is a forced branch, not a conditional, so no matter what if it gets to that part of the code, it will branch to 0x157d8. By searching the otoolit output for that, we see its after the alert panel calls giving us a good idea that this is the condition that valid registrations meet. From this information we assume readPassword is the serial checking function.

Now you might ask your self, why not just patch right here, set it to always branch, and be done with it. You could try this, but it wouldnt work, the serial numbers wouldnt hold during a restart of the app. This is due to the fact this is a call by an interface item. When the application starts up, it also calls readPassword to check at startup if its valid. This means we need to patch readPassword, not just registerAction.

Lets take a look at the otoolit output for readPassword

```
-[AppController readPassword]
00013ed0 7c0802a6 mfspr r0,lr
00013ed4 bf41ffe8 stmw r26,0xffe8(r1)
00013ed8 3c8c0002 addis r4,r12,0x2
00013edc 90010008 stw r0,0x8(r1)
00013ee0 7c7e1b78 or r30,r3,r3
00013ee4 7d9f6378 or r31,r12,r12
00013ee8 9421ffa0 stwu r1,0xffa0(r1)
00013eec 3ba00000 li r29,0x0
00013ef0 808446ac lwz r4,0x46ac(r4)
00013ef4 48020725 bl 0x34618
00013ef8 801e003c lwz r0,0x3c(r30)
.....
00013f94 40be0014 bne+ cr7,0x13fa8
00013f98 7fa24a78 xor r2,r29,r9
00013f9c 5442073e rlwinm r2,r2,0,28,31
00013fa0 38420001 addi r2,r2,0x1
00013fa4 905e0140 stw r2,0x140(r30)
00013fa8 801e0140 lwz r0,0x140(r30)
00013fac 2f800000 cmpwi cr7,r0,0x0
00013fb0 409e0008 bne cr7,0x13fb8 return;
00013fb4 981e0151 stb r0,0x151(r30)
00013fb8 80010068 lwz r0,0x68(r1)
00013fbc 38210060 addi r1,r1,0x60
00013fc0 bb41ffe8 lmw r26,0xffe8(r1)
00013fc4 7c0803a6 mtspr lr,r0
00013fc8 4e800020 blr
```

At the end is the key here. We see that over the entire thing, there is only one branch that will directly result in a return (the function reaching the end, or hitting "blr")

```
00013fb0 409e0008 bne cr7,0x13fb8 return;
00013fb4 981e0151 stb r0,0x151(r30)
00013fb8 80010068 lwz r0,0x68(r1)
```

if a condition is met it will branch to 0x13fb8, skipping over 0x13fb4. If we were to say, change the code, so that it never branched there, this might do the trick. Here's how we want the code to look after we are done

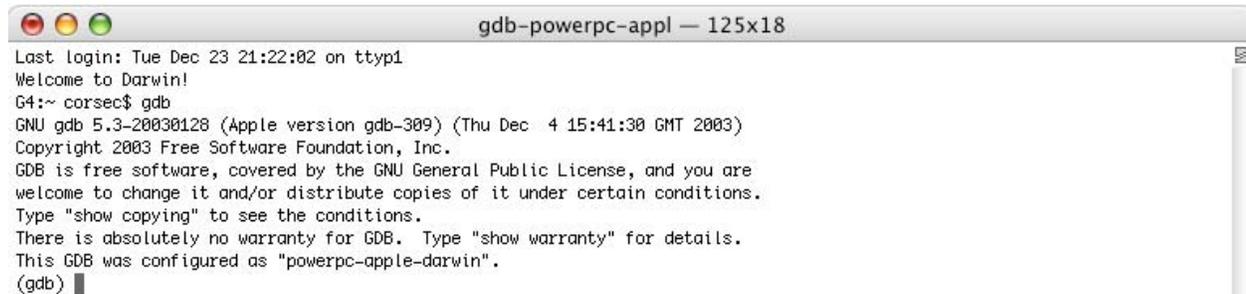
```
00013fb0 60000000 nop
00013fb4 981e0151 stb r0,0x151(r30)
00013fb8 80010068 lwz r0,0x68(r1)
```

This will make it carry on over to 0x13fb4 no matter what happens giving us the desired result. But we need to test this, we don't want to go through the work of patching this to find out this change will not work. So to save ourself work in the long run, we want to test it.

For this we will use GDB.

Let's start up GDB. To do this open another terminal window or clear the last one by typing Command + K and type:

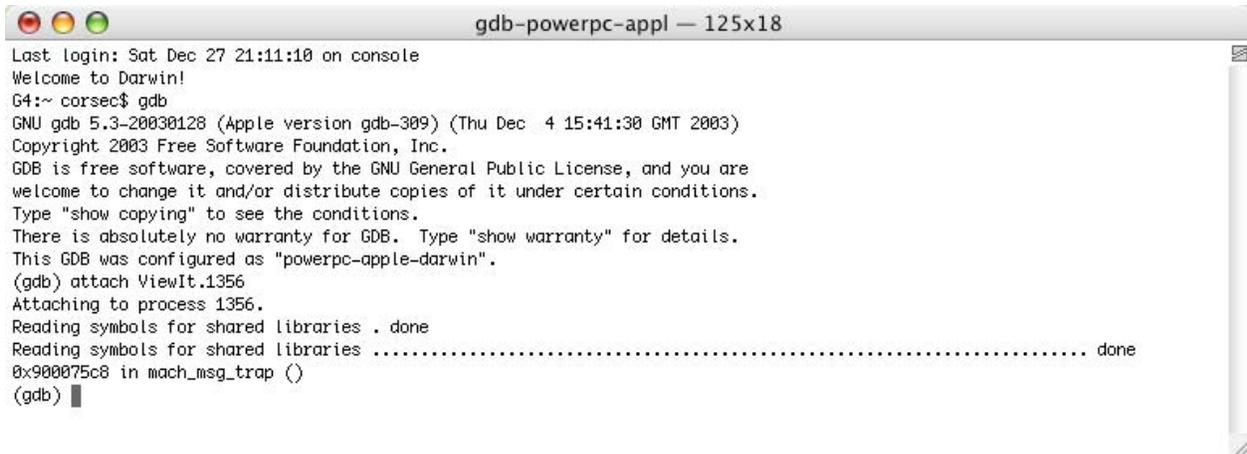
gdb



```
gdb-powerpc-appl — 125x18
Last login: Tue Dec 23 21:22:02 on ttys1
Welcome to Darwin!
G4:~ corsec$ gdb
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
(gdb) █
```

This is the what you should see after you type "gdb". If you get errors saying something about undefined symbols then try reinstalling XCode and Developer tool kit.

Next We want to attach to the application. So start up ViewIt simply by opening it the normal way, and then in the gdb window type "attach ViewIt". **NOTE:** There's an easier way then typing out the full name, simply type "attach Vie" and then hit tab. It should auto complete the name of the application, or if there is more then one running that starts with "Vie" it will show the names of them all, and simply keep typing letters and hitting tab untill you get the one you want.



```

gdb-powerpc-appl — 125x18
Last login: Sat Dec 27 21:11:10 on console
Welcome to Darwin!
G4:~ corsec$ gdb
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
(gdb) attach ViewIt.1356
Attaching to process 1356.
Reading symbols for shared libraries . done
Reading symbols for shared libraries ..... done
0x900075c8 in mach_msg_trap ()
(gdb) █

```

Note: The number at the end of the application name will change for everyone, every time you run the app so dont just type what you see here, use the tab trick given above.

For this test we want to change

```

00013fb0 409e0008 bne cr7,0x13fb8 return;
00013fb4 981e0151 stb r0,0x151(r30)
00013fb8 80010068 lwz r0,0x68(r1)

```

to

```

00013fb0 60000000 nop
00013fb4 981e0151 stb r0,0x151(r30)
00013fb8 80010068 lwz r0,0x68(r1)

```

GDB will let us do this temporarily by writing the new values to memory and letting us test the crack without having to change anything. For this we will use the "set" command in GDB.

We want to set a value for a memory address. To do this we would type

```
set *0x00013fb0 = 0x60000000
```

Lets break this down. This tells GDB to set the value at memory address 0x00013fb0 to be 0x60000000. 0x60000000 is the hex value for nop (no operation) causing the code to "fall through".

```

gdb-powerpc-appl — 125x18
Last login: Sat Dec 27 21:11:10 on console
Welcome to Darwin!
G4:~ corsec$ gdb
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
(gdb) attach ViewIt.1356
Attaching to process 1356.
Reading symbols for shared libraries . done
Reading symbols for shared libraries ..... done
0x900075c8 in mach_msg_trap ()
(gdb) set *0x00013fb0 = 0x60000000
(gdb) █

```

Then simply type `c` to let the program continue executing.

```

gdb-powerpc-appl — 125x18
Welcome to Darwin!
G4:~ corsec$ gdb
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec  4 15:41:30 GMT 2003)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin".
(gdb) attach ViewIt.1356
Attaching to process 1356.
Reading symbols for shared libraries . done
Reading symbols for shared libraries ..... done
0x900075c8 in mach_msg_trap ()
(gdb) set *0x00013fb0 = 0x60000000
(gdb) c
Continuing.
█

```

At this point open up the "Enter Unlock Code" menu under Register, and enter any name and "password" and click register.

Register program

Please enter data you received exactly as it appears in your email:

User name:

Password:



Ding Ding Ding, we have a winner. We now know that changing this value will let us enter anything and have it be valid.

Now come the part that seems to mystify everyone, where these big long strings you search for and replace come from. Lets look at the otoolit output one more time

```
00013fa4 905e0140 stw r2,0x140(r30)
00013fa8 801e0140 lzw r0,0x140(r30)
00013fac 2f800000 cmpwi cr7,r0,0x0
00013fb0 409e0008 bne cr7,0x13fb8 return;
00013fb4 981e0151 stb r0,0x151(r30)
00013fb8 80010068 lzw r0,0x68(r1)
00013fbc 38210060 addi r1,r1,0x60
00013fc0 bb41ffe8 lmw r26,0xffe8(r1)
00013fc4 7c0803a6 mtspr lr,r0
00013fc8 4e800020 blr
```

And from the past we know the second value in the table (905e0140 for the first line) is the hex value of the command. We now need to build a hex string to find and replace the branch.

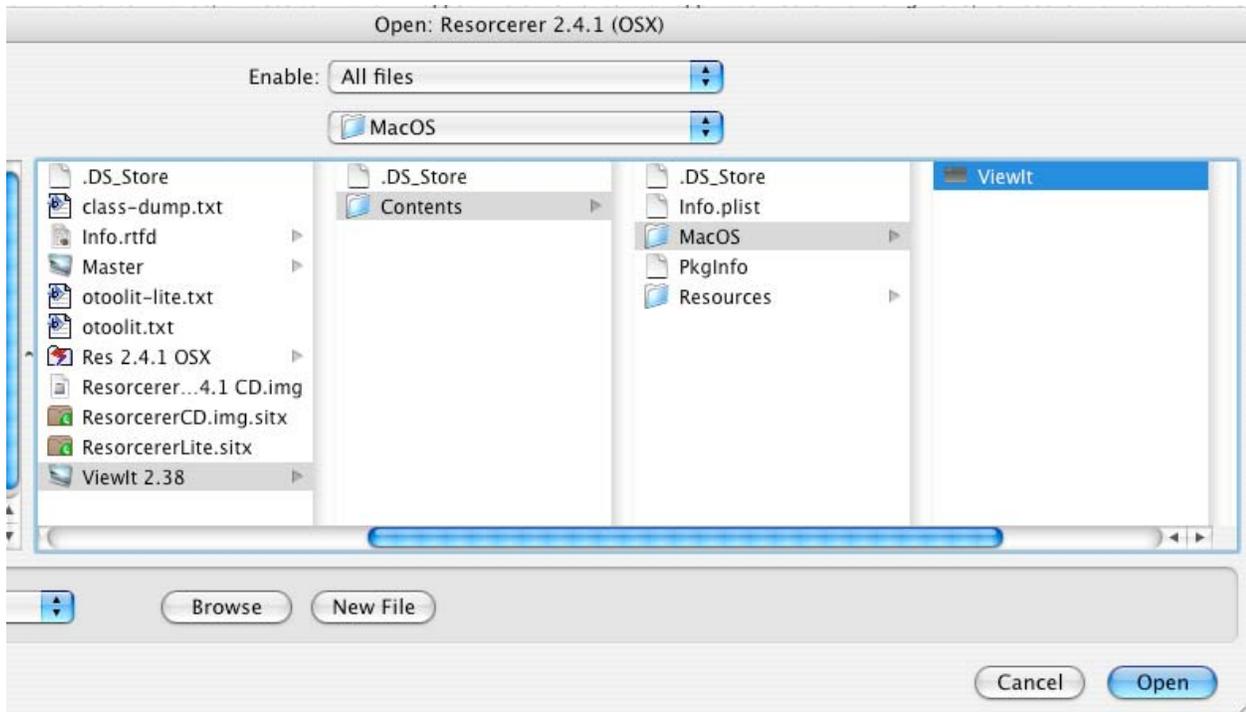
Since assembly only has a limited number of commands, all these command will appear more then once in an application. So to find the right one, we need to find it relative to the stuff around it. Example

The hex string 905e0140 will appear more then once in the application. However the hex string 905e0140801e01402f800000409e0008 might only appear once. (I got this second string from combining the hex values of the first four items, shown here separated by "-", 905e0140-801e0140-2f800000-409e0008)

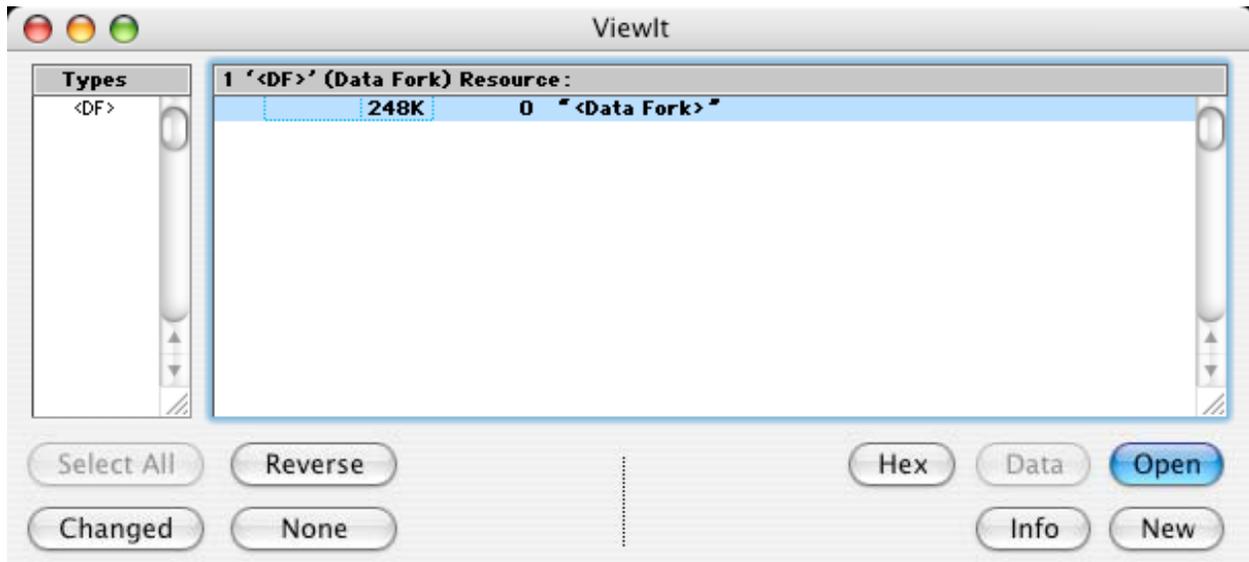
So, in the Application we need to find 905e0140801e01402f800000409e0008, and replace it with 905e0140801e01402f80000060000000.

This second string i got from replacing the branch command in the first string (409e0008) with the hex value of nop (no operation) (60000000)

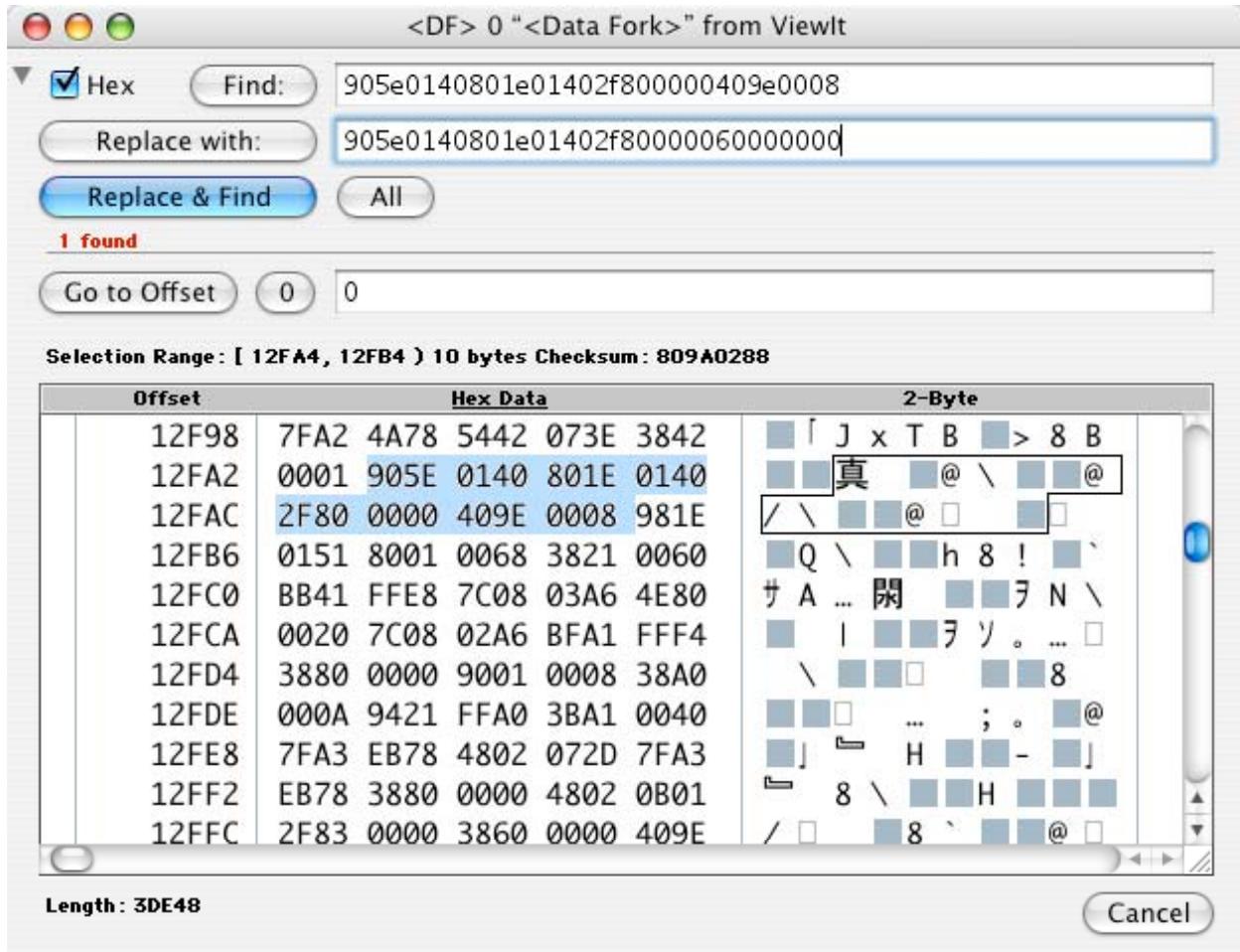
Note: in the executable hex values are **NOT** preceded with a "0x" or "*0x"
To replace this string we need to open up Resorcerer (or hexEdit). Then open the Executable (All screenshots are with Resorcerer)



This will open up a window like the following. Click on "Data Fork" and click Open



When the new window opens up press **Command + F** for find, or select find from the menus. Some new text fields will be shown.



Enter the string to search for, and the string to replace it with in the text fields as shown, and click "Find". It should find only one, but click find again to make sure it beeps and stays there meaning there is no other occurrences of the search string in the file.

Then click "Replace with" and it will replace the search string with the new string. (You must click "Find" first so it selects the string in the hex view window)

Then Simply save (**Command + S** or File..Save) and quit.

Now any time this copy of the application is opened it will think its registered.

One thing i do suggest is to trash the Prefs file (ViewIt (BJL).plist in this case) and open the cracked app again. Some times the app wont work because of the crack, because its loading a name from the prefs file that isn't there, getting a null value and chocking.

In this case the application works, but as you see in the about box, the name is "(null)" (This is the visual representation of null when its put in as text)



Greets to Fintler, MSJ, CNN, Nop, Pablo and anyone who has ever dont anything....ever.
Brought to you by Corsec AT CorruptFire.Com

INTRO TO ASSEMBLY LANGUAGE

By iÇ@açk

It's hard for me to know what your experience level is based on what you say? On the surface you say you've "dabbled" in some languages. To me that comes across as being a relative novice. On the other you say you going to learn an write a guide yourself. That IMO would take a more experienced programmer.

Since you are asking about how to write ppc assembly I am going to assume the former and you don't have much experience. Given that, your problem is not how to write ppc asm specifically, but how to write any asm at all. Aside from machine specific details and the instruction sets, the general concepts of asm coding pretty much transfer from one machine to another. So generally all you really need is the processor manual for the machine you're interested in. In this case you obviously need a PPC processor manual. They are in book stores.

Now having said that, what's your purpose of learning PPC asm code? Just general knowledge? Is it to be able to write code? Or you just want to be able to read it? To just read it a processor manual is all you really need. To write it you still need that processor manual and understand what instructions you have available in the instruction set and how to put them together to do whatever it is you want to do.

If you intend to write ppc code because you think you can do a better job than a compiler then I will tell you what I tell everyone who asks how to write ppc asm code these days for that purpose. Don't bother! Most compilers with halfway decent optimization can do a better job of "writing" asm code than you or I ever will be able to do. And as time goes by the gap between the compilers and your (human) skills will get even wider.

Why?

Because these days machines are getting so deeply pipelined that to get optimum performance out of the processor you almost need to be super human to figure out how to interleave (schedule) instructions to keep the processor from stalling (or stalling too much). Think of it as a garden hose or pipe where you stick instructions into the hose at one end, they get executed in the pipe (and I guess you could say the now old instructions are dumped out the other end of the hose). Obviously the more (instructions) you can shove into the hose and execute in parallel the more efficient execution will be and your processor is not just sitting there "twiddling its thumbs" (i.e, idle) waiting for something to finish. All it takes is one clog in the hose to stop or slow the flow. Clogs in this case mean one instruction can't be performed before some require other instruction.

Back in the 68k days the pipe was one instruction deep. There programming in asm code made sense since you could outperform most compilers (that's one of the reasons the original Mac OS was done in asm code). Now the g3/g4 I believe have about a 3-deep pipe. And the g5's are, I don't know, maybe 6 or more I guess. So to keep the pipe going you need to schedule (choose) your instructions very carefully so that they don't conflict with each other and stall the pipe. Computers programs (compilers in this case) are much better at this than humans now.

So if you are only just want to learn asm code, fine? Get a processor manual, maybe a book on asm coding, some disassembler to show you some real asm code (otool works ok).

If you want to learn ppc code with the object of writing some just to know how to do it, again fine. Have at it. If you never written asm code it really doesn't matter too much which one you start with.

If you want to learn ppc code and write it and don't care about squeezing the code to get max performance, that's fine too.

But if you want to learn to write asm code with the object of out performing code generated by a compiler I think you're wasting your time. Stick with C and/or C++. Don't dabble :-)

And here's a thought for your learning process. Write some code in C using, say, gcc (or CodeWarrior). Then use otool it get the asm code for your C code (or Codewarrior's disassemble menu command) to see the asm code in the context of the C code you wrote. That will prove more useful than just looking at arbitrary examples.

And for fun, start turning the optimization level of your compiler "up" and see the differences in the asm code for the same C code. You'll get an idea about what I was saying about instruction scheduling above.

Cocoa Cracking Technics

By iÇ@açk

This is in general response to a question over in the SNs and [k]s where Basepilot (MSJ) was asking about being more specific about a reference I made in a Proteus crack to a specific function name.

When I post a general form of a patch with references to specific function names then I am usually referring to the symbols displayed in an asm listing generated by using otool on the code (which is always in the app's MacOS dir unless specified otherwise). This is why I

also list otool offset references. Usually the addresses shown in the otool listing are 0x1000 less than what is shown in the Resorcerer display.

Here's a general cracking lesson of one technique to try on Cocoa apps (at least one I usually try first).

1. In Terminal app, Do otool -tvd on the code.
2. If app's author was stupid not to strip the symbols then they will label the various functions in the otool output listing.
3. Again if the author was even more stupid to use function names associated with the registration (look for labels containing strings like "serial", "register", or anything else you might think appropriate), then you have a starting point to attempt the crack.
4. Launch the app and get it to bring up it's registration dialog (whatever it takes to get there, e.g., usually some registration menu).
5. From the terminal, do a ps ax to get the pid of the launched app (or any other way you prefer to get the pid).
6. Execute gdb and attach to that pid.
7. Put stops at selected address(es) shown in the otool display. The app's load addresses generally exactly match the otool's addresses (there are exceptions to this like dylib's or frameworks, but usually you can figure out the relocation value). The address(es) you select are the ones you identified in step 3 above.
8. Try to register the app -- make anything up to get the registration dialog to process the registration. If you're "lucky" you'll hit one of your stops in gdb.
9. You're on your own from here! The main goal for all the above is to "get a hook" (that's what I call it) into the registration process. From here it's all experience and judgment to identify what it takes to get the registration to be accepted. And if the author was stupid enough to (a) leave the symbols in and, (b) using names giving away where registration is processed, s/he might also be stupid enough to (c) have one central registration routine which, no matter how ingenious and complicated the registration strings were, ends up in a single function returning "true" or "false" (accept or reject, 0 or 1, whatever). So all you have to do is change that routine to return the value it's caller wants that makes it think it's registration values are valid.

Finally explaining this process beyond what I have already explained (like how to use terminal, pid's, the ps command, gdb, how to switch control between gdb and the attached app) is beyond the scope of this message.

Now having said all the above, if the app's author isn't stupid, things get much harder of course. You can always get the otool listing of Cocoa code. But if there are no symbols it's obviously much harder to identify where to put stops. Backtraces from the app's open dialog don't usually help (at least I don't know how to back trace back to the app's routines from a Cocoa runloop). Also, if it's a Carbon app, you can't use otool, but PefViewer could be the moral equivalent here. Frankly I haven't tried to crack any Carbon apps on OS X (at least not yet)

One last thing. Knowing what I do about cracking I think it's sort of impressive that Pablo actually comes up with serial numbers for some apps. Personally I am content with just cracking the things. He appears to go one better by (I assume) reverse engineering the encoding algorithms to get working serial numbers (either that or the app stupidly has a string compare with some expected value based on the input). I don't have that kind of patience.

Loader Theory

By Anarchie

After learning that the Adobe Indesign installer uses some fancy tricks to keep its code out of plain sight, I thought I might as well explain what a "loader" is, and when you would make one.

A "loader", as the name suggests, is a program whose sole purpose is to load another program into memory, make some modifications to its memory space, and begin running that program. Loaders are used to crack some games, notably Blizzard games whose servers will verify checksums of several game executables via a dynamically loaded checksum library. MPQDraft for PC is an example of such a thing; it loads an application, suspends it, loads a DLL into its memory space, and resumes the app's execution at the newly-loaded DLL. This DLL does a bunch of neat stuff not directly related to loaders such as patching import tables and loading plugins, but that's for another post.

You would use a loader instead of a regular [k] in cases such as an application which checksums itself using a method that is a pain in the ass to crack (see above), an app whose code lies encrypted in the executable (e.g. SecuRom protection, again Blizzard games), and for making cracks where you need to do more complex things like intercepting system calls, crunching numbers, etc... There is another program which can help you do this called Application Enhancer (hereafter "APE"). APE is not a conventional loader - it begins a daemon 'aped' on login, and this daemon lies in wait until an application contacts the windowserver. Just about every graphical application must contact the windowserver fairly early in its lifetime in order to get a menubar, allocate windows, receive Apple Events, and all that other stuff that makes Mac UI programming so easy. When an application does contact the windowserver, aped suspends that application, copies the APE framework into its memory space at an address well above the program stack, and creates another thread which allows APE modules to communicate with APE and each other. Meanwhile, the suspended thread is immediately made to invoke APE's main subroutine, which loads any applicable APE modules and applies them.

The drawback to APE is that if you need to modify any behavior a program exhibits BEFORE it contacts the windowserver, or if you need to modify a program which does not contact the windowserver, you're out of luck.

Creating a loader on OS X is much easier than it was on 9, due to the flexibility of task and thread mucking-about afforded by Mach.

I don't have any code handy which demonstrates, but the basic gist of it is this:

Note that this applies only to completely native apps which use the Mach-0 binary format. For CFM-format apps, you might as well use APE, since almost all Carbon apps are useless without windowserver. Yes, I know that you could `execve()` on `LaunchCFMApp`, and I respond by telling you that you will need to use `vm_protect` with the not-so-well-documented `VM_COPY (0x10)` protection to allow the child to get its own copy of a shared library so you can

patch its code. <whew>. Anyway, you need to dig around in ApplicationServices.framework (? correctionplz) symbol table and find the private symbol CallPEFMain. If you break here, you will be at the right time for mucking around with the loaded CFM app, since its entry point TVector will be in r6. But since APE is easier, just do that.

Also, I assume that you know about all the functions I'm talking about, or at least know how to use manpages, developer.apple.com, the Mach API reference, and google.com to learn about them.

fork() - you're going to need two processes, bub

In your child (the one who gets 0 from fork()), call ptrace(PT_TRACE_ME, ...). This will cause the app to suspend right after a call to execve(). execve() is the syscall for blowing away your current memory space, and loading a new one from an executable.

Then call execve() on your target program's executable. At this point, the child is stopped.

Back to the parent task.

Use waitpid() to wait until the child has stopped due to execve(). You should at least check its return value to see if child died, and die yourself if so.

Use task_for_pid() to get a mach_port_t for the child task. This is used as a reference to the task when making Mach calls on it.

At this point, the executable has been mapped into memory, but the task's current program counter is at __dyld_start, indicating that the app hasn't been linked to any dylibs yet. You can freely modify the application's code and data, with the exception of the symbol pointers which dyld is going to overwrite. One of the things you can do is insert a 'trap' instruction somewhere in the code, and then resume the task using ptrace(PT_CONTINUE, ...), and then using waitpid() to wait until it receives the SIGTRAP signal. At that point, you can reinsert the original instruction over the 'trap', and do whatever it is you needed to do at this breakpoint.

Once you've done your magic, it's time to use ptrace(PT_DETACH, ...) to send your application on its merry way. Be careful - you can't attach GDB to the child between the calls to PT_TRACE_ME and PT_DETACH.

And there you have it. I hope someone found this educational. Corrections, suggestions, feedback of any kind is welcome.

lord anarchie

MacsBug for Non-Programmers

Part I

- Playing Cool Games with Dangerous Toys
- Obtaining and Installing MacsBug
- Navigating MacsBug
- Basic MacsBug for Non-Programmers
- Crash Recovery
- Faster Restarting

Helping Programmers Debug
 Working with Numbers
 Working with System Information
 Practice Your Basic Skills

Part II

Poking Around Memory and Other Dangerous Hobbies
 Stop the Presses: MacsBug 6.5.4a4
 MacsBugApp
 Partitioning Memory
 What Can You Do With This?
 Processing Processes
 More On Crash Recovery
 Disabling User Breaks
 Finding Lost Data
 Cheap Auto Rebooting
 A Good Starter Set

Note: This was originally a two-part MacCyclopedia series in MWJ. However, between the release of the two parts, Apple released a new version of MacsBug. Therefore, Part I (1998.03.09) discusses only MacsBug 6.5.4a3, and Part II (1998.03.16) has "breaking news" about MacsBug 6.5.4a4.

Playing Cool Games With Dangerous Toys

Although the audience can be fickle at times, software publishers know that, next to games, there's no program all computer users love as much as a good utility. Even though computers do a decent job of assuming our workload in repetitive tasks, there is still plenty that can be done to make them friendlier, and Apple isn't going to build every possible diagnostic or enhancement tool into the shrink-wrapped Mac OS packages. That leaves plenty of room for Qualcomm (current owners of Now Software), Symantec, MicroMat (makers of TechTool) and others to strut their stuff for your hard-earned dollars.

But computers are a strange sliding scale. The easier a task appears to the user, the more complicated it is for the programmer. Software developers have very easy tasks in operating systems like UNIX, where everything is based on a command line and they're free to invent whatever twisted user interface fits their particular fancy, or mood, or mental illness on that day. Systems like the Mac OS enforce a consistent user interface and high user expectations, meaning developers have to work harder--significantly harder--to make their code match what you expect to see.

The more complex tasks become behind the scenes, the more obstreperous the tools they manage to become to use. As any utility moves more towards general purposes, it gains power but loses its ability to help you easily perform specific tasks. For example, screenplay-writing software makes it easy to format your documents in the precise style required by filmmakers and production companies. A more general word processor like Microsoft Word or Nisus Writer can do the same task, but they don't come with the code to walk you through the process. A drawing program can make pie charts as well as Excel, but it won't turn raw numbers into your chart for you--you'll have to draw the arcs and circles and pick the colors yourself. Some integrated programs can create drawings from spreadsheets, but modifying the chart requires either regular drawing skills, or modifying spreadsheet numbers and starting over. In general, the more powerful the tool is, the less helpful it is on specific tasks.

If you follow this line of reasoning to a logical extreme, you arrive at MacsBug.

Originally an abbreviation for "Motorola Advanced Computer System Debugger," MacsBug is today a powerful Mac OS only tool for debugging programs and digging into the lowest levels of the operating system underneath that six-colored fruit logo. MacsBug is described, quite truthfully, as a programmer's tool--its mysteries are not for the uninitiated, and its powers are such that if used incorrectly, you can screw up your computer to the point of having to restart. (You can screw up your disks, too, but that's usually a bit harder to accomplish, although you ought to be aware of it.) It is designed and aimed at helping programmers manipulate and analyze the code and data of their programs, to help them find and eliminate bugs. To this end, it lets you into the deepest and darkest recesses of the system, the places that have no user interface of any kind because they're not user-level domain.

That's why, if properly used, MacsBug can also give you information about problems like no other tool. While most of the built-in functionality is very strictly aimed at programmers, many of the external commands added in recent years can, in some cases, provide you with information that just might help you solve some crashing problems, or figure out what in the Sam Hill is going on, even if your programming skills don't extend beyond a cursory knowledge of AppleScript.

In this entry, we'll try to give you some basics about MacsBug, show you why it's useful to have around even if you're not a programmer, and reveal some of the hidden mysteries of the debugger that might lead you closer to nerdvana, if this is where you want to go today. First up is how to get it and where to put it (on your disk, that is).

Obtaining and Installing MacsBug

MacsBug is free for the asking from Apple's software archives, in the "Utilities" section. Like most Apple software distributed online, MacsBug now comes as a Disk Copy disk image. Unlike most Apple software of more than a few files, there's no installer. MacsBug is for programmers; the authors (who work on it in their spare time--Apple rarely has full-time engineers working on MacsBug) expect you to figure out how to install things from the instructions they provide.

However, that's not the latest release. The current version, released last April (MDJ 1997.05.01), is not MacsBug 6.5.3 but MacsBug 6.5.4a3, so designated because there is no formal testing for it and it can't easily be declared non-alpha any other way. But it works. To keep the uninitiated from easily finding it, Apple hides it away on the developer server. However, if you're paying attention, Apple's "MacsBug Version Information" file in the same folder as MacsBug 6.5.3 tells you where to find the "alpha" version, if you bother to look inside the file. Unlike the user-level distribution, MacsBug 6.5.4a3 is just a Stu It archive with some files in it.

When you've got the unpacked MacsBug 6.5.4a3 distribution, you'll find a comprehensive "Read Me" file with changes since the 6.5.3 release, plus five folders full of other files. The "Building_dcnds" folder is for programmers writing their own additions to MacsBug and won't concern us. The "Into_System_Folder" directory contains MacsBug itself, and it goes at the top level of your System Folder (the same folder that houses your "System" and "Finder" files, if you have it buried a few levels deep as some people do). Older versions of MacsBug use a "Debugger Prefs" file containing external commands and preference resources--since changing it requires a resource editor, it comes with the creator type of ResEdit so

double-clicking it will open Apple's free resource editor. The "Debugger Prefs" file has to be at the top level of your System Folder as well, but now MacsBug can load resources from up to 32 separate files of any kind found in the "MacsBug Preferences" folder inside your System folder. The distribution hints that it ought to go in the Preferences folder, but it can actually go there or in the Preferences folder--in fact, MacsBug seems to create a "MacsBug Preferences" folder in the Preferences folder if one isn't there. Apple does not define which of these preferences folders is examined first.

The "Into_Debugger_Prefs_file" folder contains a 'kchr' resource you can copy into your Debugger Prefs file, if you have one. MacsBug formerly came with a fully-loaded Debugger Prefs file, but in recent releases all of its goodies have moved into the MacsBug file itself. That leaves Debugger Prefs totally as your own file, and you don't have to worry about copying your resources out of it and into a new version when Apple updates MacsBug. The 'kchr' resource is necessary for MacsBug to take keyboard input--US users don't need it, but non-US users might not have the require US 'kchr' resource, so Apple supplies it. It has to go in the "Debugger Prefs" file; other files won't do.

The "Book_Example" folder is for examples and tutorial information from Apple's old official manual, MacsBug Reference and Debugging Guide. It's a decent programmer-level tutorial, and is official documentation for 1991-level MacsBug 6.2, but it's not as good about teaching programmers how to debug software. For that, you want Debugging Macintosh Software with MacsBug by Konstantin Othmer and Jim Strauss, who debugged more stuff than you really want to know about. Since the first book is Apple's official manual, the MacsBug distribution still carries the supporting files for the book, in case the accompanying disk ever gets updated, but don't hold your breath. Unless you have the book, you can safely ignore this folder as well.

Once you have all the files in place, restart your system. Note that on the "Welcome to Mac OS" screen, in the same place you would see "Extensions Disabled" if you held down the "Shift" key, is the text "Debugger Installed."

That's how you know it's in the right place. If you hold down the Control key during startup, you'll enter MacsBug as soon as it's loaded. This probably isn't what you want, so don't do that right now. If you do, you'll need to figure out how to get out of it, and that's next.

Navigating MacsBug

Normally, you'll enter MacsBug in one of a few very defined ways:

By holding down the Control key as it loads at startup time

When a program crashes (or, more specifically, when the Mac OS calls its built-in "SysError" routine to draw the "bomb box" normally associated with a crash)

When a program specifically activates the debugger through commands called "user breaks"
By holding down the Command key and pressing the Power key

By pressing the hardware "programmer's switch" on older Macintosh models that come with such items. Usually there are two switches--one is a hardware rebooting switch and has the same triangular symbol as the Power key on the keyboard, and the other one has a broken squiggly line in it. The squiggly switch is the programmer's switch. Most recent Macintosh models don't have them, and the early 1990 models often made them optional. The classic

Quadra 800 case has them right in the front in one of the most visible examples of these switches.

Once you enter MacsBug, you'll see a bewildering array of information. MacsBug wants 640 X 480 pixels to display its technical wisdom, but it will manage with 640 X 400 if that's all your screen supports (as is the case with some models). If your screen is already running at the lowest bit-depth that your system supports, you'll see MacsBug's display centered in the middle of your normal desktop--if your screen is larger than MacsBug's display. If your screen is larger but you're running at a higher bit-depth, MacsBug shifts your video drivers into low gear and forces the lowest possible bit-depth, so outside MacsBug's rectangle you'll see some distorted or expanded view of your normal screen. Ignore it if you can.

The very bottom line of this text window is the command line, where you type commands to MacsBug. It's a programmer-level tool--don't even think about using the mouse, because MacsBug does not and never has supported it. The left side of the screen shows the microprocessor registers. If you're on a 68K machine or running emulated 68K code, you'll see eight registers starting with "D" and eight starting with "A". PowerPC native code shows thirty-two registers, from R0 through R31 (with special names for R1 and R2). Ignore all this stu -- programmers use it to see what's going on, since registers are the heart of a microprocessor, but you won't care as a non-programmer.

At the top of the left-hand column is something the microprocessor calls a stack. Think of it like a stack of dishes in a cafeteria, because that's the heritage of the name--you can add more dishes to the top of the stack, and take any number o the top, but you can't take anything out of the middle of the stack without first taking care of the dishes above the one you want. The stack is really just an area of memory. Every time something is pushed on it, the stack pointer decreases a little bit. When something is pulled o the stack, the stack pointer increases a bit, always pointing to the "top" of the stack in memory. Each program's stack is at the very high end of the memory partition you've allocated for it in the "Get Info" box in the Finder. The stack grows downward from the top (it's like an upside down stack), and if it ever grows so big that it expands out of its normal space, it's a "stack overflow." That's a fatal system error if the system can catch it. You'll mostly ignore the stack, too, although it can be useful in a few ways that we'll explore later.

Between the stack at the top left and the registers at the bottom left is the phrase "CurApName," a global location in Macintosh memory that stores the name of the "current" application. Recall that the Mac's multitasking system works by passing control around to programs when waiting for you to do something. If you're not keeping up with the computer--and you probably aren't--the Mac OS will give another program some of the processor's attention until you do something. This is how other programs keep downloads going, clocks running, animations animating and so forth, even when they're not in front. So even if Netscape Communicator is frontmost when you enter MacsBug, it's CurApName that shows you which program is really in control. Watch that label carefully or you won't know where you are in the system.

The rest of the MacsBug screen is a scrolling log of the results of your commands. You can scroll through it with the up and down arrow keys, or page through it by holding down the Command key while using the up and down arrow keys, or by using the paging keys on an extended keyboard. Typing "help" or pressing the "Help" key on an extended keyboard will show the beginning of MacsBug's built-in help; pressing Return will cycle through all the help topics one by one. Just so you know.

You should also know that while you're in MacsBug, interrupt signals on your Macintosh are temporarily halted. The Mac uses interrupts--hardware signals that a device needs attention--to handle everything from mouse movements to high-speed Internet activity. As long as you're in the debugger, all of those signals go ignored. That's not good news--file servers and Internet protocols will time out after a couple of minutes, so don't try extended MacsBug sessions while connected to a network. (If you're connected to an AppleShare file server, you can type "stopxpp" to stop all sessions of XPP, an AppleTalk protocol. This disconnects you from all file servers, but it does so in a way that keeps programs from locking up your computer for two minutes waiting for a long overdue response.)

If you're on a normal dial-up Internet connection, you won't want to stay in MacsBug too long if you can avoid it, or the line might drop. To return control from MacsBug to the Mac OS, exactly where you interrupted it, type "G" and return. You can also press Command-G as a shortcut to "go" back to the Mac OS. There's a macro command "GG" that clears out all breakpoints and returns to the Mac OS; some people like to type that just to make sure they won't come back in accidentally. It won't hurt, but it's not necessary here. Don't get confused and hit the "Esc" key--that temporarily toggles display of the graphical user interface, but you're still in MacsBug! Press "Esc" again to see the text display.

Basic MacsBug for Non-Programmers

Now that you can enter and exit MacsBug at will, let's talk about some useful things.

Crash Recovery

When MacsBug is installed, all crashes make you enter the debugger. Instead of a graphical "bomb box" or even a vanishing program with a cryptic "unexpectedly quit" message, you get dropped right into the debugger at exactly the point where the system realized something was wrong. What's more, you get a more complete explanation of the problem, such as "Illegal instruction" at some address in some data space with other details you don't care about. The lines at the bottom just above the command line are the machine-level instructions right where the problem was discovered; the one with a "*" by it is somewhere near the current instruction (the one that couldn't complete), but emulation and bus timing concerns sometimes make MacsBug unable to pinpoint the exact instruction that triggered the problem.

Long-time subscribers can check MacCyclopedia in MDJ 1997.03.06 for a complete explanation of all the system errors and what typically causes them (such detail is beyond the scope of this article). With MacsBug, you always know what happened, and by examining CurAppName, you're pretty sure about which program was "in control." What CurAppName cannot tell you is whose code caused the problem. Extensions, control panels, components, and anything else that's not an application won't show up in CurAppName, and neither will large components of the system software (the Finder, being an application, is a notable exception). Consider CurAppName a hint about what's going wrong, but not a way to assign blame.

Programmers are expected to use MacsBug's capabilities to find and hopefully fix their problems. It's highly unlikely that you can fix a program you didn't write, even if you are a programmer, but MacsBug has a handy command for this. "ES" stands for "ExitToShell," the Mac OS routine called to tear down every application when it's finished. When an application "unexpectedly quits," the system calls ExitToShell on it while it's still running, disposing of all its memory and cleaning up after it as best it can. Sometimes the system doesn't give you the option of ExitToShell--it just shows you the "System Error" dialog box and forces you to restart. MacsBug always lets you try "ES". If it works, you may be able to save work in other programs before restarting. If it doesn't, you're still crashed, but

it was worth a shot. If "ES" doesn't work, then "unexpectedly quitting" wouldn't have worked either, so you're no worse off. And in some cases, you're much better off. "ES" is much like pressing Command-Option-Esc in System 7 or later, but since it's part of MacsBug, it works even if you crash.

Note that Norton CrashGuard tries to do some of this same work, but it's only available as part of Norton Utilities 3.5 and only works on PowerPC machines. MacsBug is free and was invented for the 68K series of processors. Also note that "ES" always kills the program whose name is shown in CurAppName--like Command-Option-Esc, it's easy to accidentally quit the wrong program if you're not paying attention.

Faster Restarting

If "ES" doesn't work, there are a few other commands. "RS" stands for "restart," but it's not the same kind of "Restart" the Finder offers. The Finder's "Restart" command (the same one you get when you press the Power key and choose the "Restart" button) sends an Apple event to all programs, asking them to quit, and then turns off the hardware only when the Finder is all that's left. The MacsBug "RS" command tries to unmount all of your online volumes and then toggles the hardware power, so you'll lose any unsaved work in any application. The main advantage to "RS" over hitting the physical power switch is disk unmounting. If a disk isn't unmounted correctly, the Mac OS realizes that something's wrong and goes through a time-consuming verification cycle next time the disk is mounted (made available for use). "RS" gives you a shot at unmounting the disks before restarting, saving some time. If that fails, "RB" tries to unmount only the boot volume before telling the hardware to restart. Either is better than simply pressing the power switch twice.

Helping Programmers Debug

If you crash someplace often, MacsBug is the tool to help programmers find out what's going on. And in recent releases, gathering the information is just about as easy as you could hope. MacsBug has a few expansion features. Macros are abbreviations for sequences of other commands. For example, the "GG" command is really a macro that expands into commands to clear all breakpoints and go back to the Mac OS. Type "help GG" in MacsBug to see--help on any macro gives you its full definition.

When you crash, especially in repeatable circumstances, the macro you want is "StdLog". It's short for "standard log," and it records a text file on disk containing the most essential information about your computer at the time it crashed. What the microprocessor was doing, what it thought was going on, whether or not memory was obviously trashed, what files were open, and much more. You can see it all whiz by as it happens. StdLog uses the "log" command to write MacsBug's output to a text file on disk; by default, it writes to a file named "StdLog" in the desktop folder of your hard disk. You can specify another place by using the "StdLogInto" macro followed by a full pathname (that's a text string with all the folders on the way to the file you want, like "Hard Disk:System Folder:Preferences Folder: My log file"). In fact, "StdLog" is just a macro for "StdLogInto StdLog," since file names without pathnames in front of them go to the desktop folder of your startup disk.

The file that StdLog and StdLogInto produce is more information about a crash than most programmers receive with most bug reports. MacsBug also supports "names" for code--programmers can build their programs so that each routine they write has the routine name embedded in the code at the end of the routine. MacsBug uses this to tell programmers exactly where things go wrong, as in "Bus error at DrawIconInWindow+0073", meaning 115

bytes (0x73 in hexadecimal, MacsBug's default number system) into the routine "DrawIconInWindow." The log file will record these kinds of things if they're present, and that will help programmers find problems a lot more easily.

Just be careful--repeating the "StdLog" command twice without removing the first "StdLog" file will make MacsBug append the new log to the old one, leaving just one file. The log files are just plain text files with the MPW Shell creator type--MacsBug was designed for programmers, so double-clicking the logs opens Apple's own programmer's environment.

Working With Numbers

MacsBug also makes a quick integer calculator, if you know what you're doing. Remember that the default number system is hexadecimal, with digits 0-F (representing decimal numbers 0-15), and that each place value is the digit times sixteen, not the digit times ten. In decimal, "34" means $(3*10)+4$, for thirty-four. In hexadecimal, "3F" means $(3*16)+15$, or sixty-three (decimal). If you pop into MacsBug and type "34+10", you'll get "44", but only because the digits work out the same in hexadecimal for those cases. Typing "34+9" gets you "3D". To tell MacsBug to treat a number as decimal ("normal") instead of hexadecimal, put a "#" symbol in front of it. #34+#9 does equal #43, or \$2B (the "\$" is an old prefix meaning hexadecimal notation).

MacsBug also knows about the memory and disk space modifiers "K", "M" and "G". 1K is the same as 1024, since a kilobyte is 1024 bytes. If you need to know how many K are in 200MB for some reason, enter MacsBug and type "#200M/ #1K", and you get back a result that ends with "exactly #200K," which isn't that surprising. Type "help operators" in MacsBug to see a list of mathematical and boolean operations that work--your basic four functions plus some bitwise operators that programmers need to mess with individual bits, and a few other gems you'll generally never use.

Working With System Information

More important than numbers is system information, especially if you're trying to solve a problem. MacsBug has a wealth of information about what's going on under that graphical interface, just waiting for you to ask.

A classic example--an MWJ sta member was trying to launch a 12th program on a large machine when everything started quietly failing. Some programs would give error "-42", but he couldn't get a "System Errors" program open to tell him what it was. MacsBug to the rescue! Entering the debugger and typing "error #-42" (remember, decimal numbers must start with "#") spits back a string that says "tmfoErr--too many files open." You may not know the Mac OS is limited to 343 open files at once.

"But how so I know what files are open?" cried the sta er. "I only have a couple of documents open. What's the deal?" MacsBug strikes again! An external command named "file" lists information about every open file on the system.

After scrolling through a few pages of the list, it became obvious that the sta member had inadvertently opened a lot of font suitcases with Adobe Type Manager Deluxe--and we mean a lot, as in about 75 of them. When factoring in files that the system keeps open for itself (catalog and extents trees on HFS and HFS Plus volumes, Finder Preferences, Desktop database files on each volume), plus BBEdit plug-ins, Acrobat Exchange plug-ins, WebArranger plug-ins, Internet Explorer plug-ins, shared libraries, extension files, extension

preference files, application data forks (for PowerPC code), application resource forks (for interface resources--and each fork counts as a separate file)--no wonder it ran out of space! A quick look in MacsBug helped him figure out what he could do (he asked ATM Deluxe to close about 60 of the unused font files) and that helped tremendously.

Of course, half the battle is knowing what MacsBug can do. Some of the useful information comes from built-in commands, but much of it comes from dcmds. Like HyperCard's "XCMD"s, dcmds are "debugger commands," written by programmers and dropped into MacsBug's lap by adding their resources to the Debugger Prefs file--or to one of the first 32 files in the MacsBug Preferences folder. "File" is a dcmd, and it's quite useful in areas like the extant case. Some useful dcmds for non-programmers include:

"File", as mentioned, displays information about open files. You can restrict the display to files of a given name or type with options (type "help file" to see how). Every time a file is opened, it returns a reference number that the system uses when operating on the open file. "File" displays reference numbers in hexadecimal, but simply typing the number as a command will show you the decimal equivalent. This can come in handy for people working with the file access commands in AppleScript. If your script opens a file but stops before it closes that same file, the system leaves the file open until the application that called the script quits--and depending on how the script executed, that might not be so easy to arrange. The "File" command can show you the reference number of a file you left dangling in a script you're testing. Let's say the number is 7F34. You type "7F34" on the command line and get the decimal number 32564 in return. That's the reference number. The AppleScript statement "close access 32564" closes the file for you, so your script can open it next time without getting a "file already open" error.

"VMDump" normally displays a lot of information about which parts of RAM are handled in various ways by virtual memory, and you're extremely likely not to care. However, adding the "-f" flag (as in "VMDump -f") forces the dcmd to give you a list of all file-mapped files. File mapping reads the file itself as the swap file, instead of reading the contents into RAM and swapping it with the major "VM Storage" file on disk. These are the files that will require more memory if you turn VM on.

"RD" shows you the current resource chain. When a program requests a resource from a file, the programmer has the option of telling the system to stop if the resource isn't in the last resource file opened. In most cases, though, the system keeps looking through all of the open resource files, from the most recently opened to the least recently opened, stopping only when it reaches the bottom of the chain (typically the System file itself) or the resource is found. This is how programs can override default behaviors--by providing a custom control definition procedure resource, for example, the programmer can act just like he's using regular simple buttons. When it's time to use them, the system looks for the right 'cdef' resource, and if it happens to find it in the application's resource fork before getting to the System file, so much the better.

The Mac OS does some Serious Voodoo Magic to make all the files in your Fonts folder look like they're part of the System file to the Resource Manager, so "RD" has an option to skip them. The command, which stands for "resource dump," displays information about resources in open files unless you specify the "-c" option, as in "RD -c", to show just the resource chain. "RD -s" does the short form, leaving out all the font files in your Fonts folder. If you're getting a "resource not found" error (#-192), "RD" can help you figure out where the system is trying to find the resource. This helped us recently figure out that an extension was responsible for a printing problem.

"Vol" lists all online volumes (disks), if you think one is not available but should be or vice-versa. "Vol" can't fix problems, but it can let you know if the OS is aware of a disk or not. Each volume is listed with a "d" flag for whether the volume is "dirty" or not (has blocks in the cache that need to be written to disk before the disk is unmounted), "s" for software locked, or "h" for hardware locked (like a write-protect tab on a floppy disk). Capital letters mean the attribute is true, lowercase letters mean it's false.

"ProcInfo" shows you all the current processes, with flags indicating which one is the front process (different from the one listed in CurApName, perhaps), which ones are background-only, and so forth. A "free" column shows how many free bytes are in each process's memory partition, so if a program says it's running out of memory, you can see which programs are using the least memory and perhaps reduce their partitions for the future. User-level utilities can sometimes do this as well, but MacsBug is handy.

"Gestalt" is a dcmd that displays the return values of all "Gestalt" selectors. Gestalt is a Mac OS mechanism programs can use to register values that any other program can find. For example, Suitcase 2.0 and later register a Gestalt selector that points to a single two-byte value in memory. Every time Suitcase opens or closes a font, it increments the two-byte value. Suitcase aware programs can call Gestalt to find the value of this location and watch it. When the programs become frontmost, they can examine the value--if it's changed, it's time to rebuild the font menu. The Mac OS registers approximately six googol and three Gestalt selectors to tell programmers what features are present, what versions of system software are available, what bugs are fixed and what programmer initials are. The "Gestalt" dcmd doesn't explain what all the selectors mean, but it shows you some meanings and lets you see all the values. It's more educational than useful, but that's true about the Internet, too.

Practice Your Basic Skills

While this is a long way from everything MacsBug will do, it should be enough to let you play around and learn all kinds of stuff about your system. Note that MacsBug does require a bit of memory (at least a megabyte). Also, try not to change any values if you can avoid it--just look at them unless you're absolutely sure of what you're doing. Programmer toys can be dangerous if not used properly.

Poking Around Memory and Other Dangerous Hobbies

In Part I, we discussed how to find the latest version of MacsBug, how to install it, what the various preferences files meant, how to use it for elementary crash recovery beyond what the system normally allows, and how to use external debugger commands to poke around open files, the resource chain, and other programmer-related arcana that sometimes affect your use of the system.

In this conclusion, we'll examine the memory structure of the Macintosh, including what applications are really doing with those megabytes of memory you allocate to them in the Finder's "Get Info" dialog box, with plenty of warnings about how a little learning is a dangerous thing. But first, we have breaking news that could make these specific kinds of exploration much simpler.

Remember, however, that MacsBug is a programmer's tool. That means it deals with rather technical concepts, and has a user interface only a programmer could love (or even tolerate). For example, to configure MacsBug, you have to edit resources (with ResEdit or Resorcerer). Some resources, like those for external commands, can be in MacsBug, or in the "Debugger Prefs" file located in the System Folder, or in any of the first 32 files (listed by name) in the "MacsBug Preferences" folder in the Preferences folder. There are other kinds of resources that configure internal MacsBug parameters (instead of adding new capabilities), and there can only be one of each kind of those resources (like 'mxpr' for MacsBug preferences, 'mxbc' for MacsBug colors, plus others). MacsBug itself has one of each of these resources, but you should leave them alone. If you want to edit them, copy them and paste them into your copy of the "Debugger Prefs" file. MacsBug won't even look for these kinds of resources in the "MacsBug Preferences" folder.

Stop The Presses: MacsBug 6.5.4a4

After a few weeks of trying to get it out the door, Apple Computer last week finally released MacsBug 6.5.4a4, the first new release of the debugger in about a year. There was a newer version included with the "Blue Box" shipped to developers after the Rhapsody Developer Release 1 (MWJ 1997.11.24), but only those developers who actually have a Rhapsody DR1-capable machine could get to it, if they even noticed it--and it's probably just as well that they didn't, since it had a bug that could trash your hard disk if you pressed a key while MacsBug was writing to disk.

The new release comes with extensive change notes, but we want to cover a few of them here because they might affect non-programmers who have MacsBug installed:

MacsBug now uses color for displays. This is somewhat of a misnomer, because it doesn't use much color, and formerly it would use some color. The "Debugger Prefs" file could optionally contain one resource of type 'mxbc' specifying a single RGB color to use for text in MacsBug's rectangle, and another RGB color for the background color. That resource is now obsolete, and a more standard 'clut' (color look-up table) resource with four entries is now used. The first entry is the background color, the second is the text color, and the third is the "pay attention to this" color (pure red by default). The fourth color is the "pay a little attention to this" color, but MacsBug 6.5.4a4 doesn't use this color. The primary use of color in this release is in the register display along the left-hand side of the display--when programmers step through code, getting MacsBug to execute one instruction or one subroutine at a time, any registers that change during the execution are redrawn in red to bring them to your attention. When you first enter MacsBug, all of them will probably be red since the "previous" state is undefined.

MacsBug still automatically switches your monitor to the lowest bit-depth it supports, which is one-bit color on all but the newest Power Macintosh systems. All MacsBug really needs is four colors, or two bits per pixel. However, if you play with ResEdit and examine the 'mxpr' resource in MacsBug itself (copy it into your "Debugger Prefs" file before modifying it, please), you'll find an option labeled "don't swap display bit depth." If you set this option, MacsBug will never shift your display into a different bit depth, meaning the MacsBug display pops up quite nicely in the middle of your "Happy Mac" screen without affecting the rest of the display. This is pretty cool, but MacsBug has to save and restore the area of the screen underneath its display. If you don't let it switch pixel depths, then MacsBug has to allocate enough memory to handle the pixels under its display at any supported bit depth. If you're running the MacsBug monitor in millions of colors, for example, MacsBug has to allocate four bytes for every pixel in 640 columns and 480 rows (1200K, or 1.2MB of RAM). That memory is lost to your system while MacsBug is installed.

Unless you have a solid reason for wasting a megabyte of RAM, leave this alone. If you have this much RAM to waste, consider using or enlarging a RAM disk, since it's generally a more useful way to allocate memory.

If you don't set this preference, though, MacsBug will still shift into black- and-white only mode if your system supports it, and that means you won't see the new red numbers in the display. You might want to play with both settings to decide which works better for you, particularly if you're a programmer who steps through code--that's where the red displays come in most handy.

If you've had MacsBug installed for a while, enter it and type "help leaks". If you're told that no command by that name is available, great. If you get help for a dcmd by "Bo3b Johnson" (the "3" is silent, you see), open your "Debugger Prefs" or "MacsBug Preferences" files until you find the 'dcmd' resource named "leaks", and remove it. Leaks is a tool that watches how memory is allocated, and it helps programmers figure out when a chunk of RAM is allocated but never released back to the system when the program is finished with it. That's called a memory leak. The "leaks" command patches several Mac OS Memory Manager traps to do its work, and the code it inserts is 68K code. If you're using a PowerPC system, "leaks" could be unintentionally slowing down calls the system makes thousands of times per second, so ditch it. (If you know what leaks is and actually use it, consider keeping it in a separate file and rebooting when you need it to increase overall system performance.)

MacsBug now understands displays that turn themselves off automatically to save power, and will try to turn them back on when it gets control. In previous versions, a crash into MacsBug while the display was powered off left you with a black screen--and no way to turn it on, since the Mac OS software responsible for waking up your monitor couldn't get control while MacsBug was active. This is now improved.

The "RS" and "RB" commands discussed in Part I are a bit more bare-bones in their execution. Programs on the Macintosh can ask the system to call them when it's time to shut down or restart the machine--a good example is Open Transport/PPP, which installs a shutdown "task" so it gets a chance to drop your PPP connection. In earlier versions, MacsBug would allow those tasks to execute during "RS" or "RB", but the system wasn't always in a state where such tasks could work properly, creating more crashes (or just plain lock-ups). Now MacsBug calls only the ROM-based Shutdown Manager, or tries to do so, to avoid these problems.

Keystrokes will no longer occasionally "leak" from MacsBug into the frontmost application.

The "StopXPP" dcmd mentioned last week is now renamed "StopAS," as in "Stop AppleShare." AppleShare connections can now be over TCP/IP as well as AppleTalk now, so "StopAS" closes AppleShare sessions on both transports. As noted last week, an AppleShare request that times out while you're in MacsBug can result in a two-minute lock-up after exiting when a program tries to access the server, which is disconnected except your machine doesn't know it. "StopXPP" is an alias for "StopAS," if you're already used to the old command.

Programmers can always count on MacsBug release notes for a shot or two of humor. Our favorite in this release: "The STAT command and SECONDS basic type now show the year as 4 digits instead of 2, making MacsBug year 2000- savvy. This saves the Macintosh industry approximately US\$430 million per year, according to current US Department of Labor statistics." Use your savings to buy extra copies of MacsBug.

MacsBugApp

However, the most important change for our purposes in MacsBug 6.5.4a4 is the inclusion of "MacsBugApp," an application version of MacsBug that's been kicking around Apple for a while. Double-click MacsBugApp and you'll get a new application with a window containing MacsBug's familiar display. Type commands, look around, see registers display in red, and all your favorite commands. And since it's just a regular application, other programs continue to work as usual--file transfers aren't stopped, networking connections won't drop, and so forth.

Theoretically, MacsBugApp is a replacement for an older "TestDCMD" program. Developers creating debugger commands can now install them in MacsBugApp (its "Debugger Prefs" file has to stay in the same folder as MacsBugApp itself, as does the "MacsBug Preferences" folder), and use the real MacsBug to debug them in the middle of MacsBugApp.

For your purposes, though, MacsBugApp is a less dangerous way to poke around the system. Since it's an application and not a "real" debugger, MacsBugApp can't do things like step through code, but if you just want to look around the system, it's just fine. It won't help you in crash recovery--typing "ES" in MacsBugApp gives you the weird "System Error #0"--but all of the poking around the system we'll talk about this week works just fine from within MacsBugApp.

To demonstrate, take last week's example of "too many files open." Launch MacsBugApp (if you can) and type "file" to see a list of all open files. If you want to know which ones will be closed, use the "ProcInfo" dcmd first. That gives a list of all processes, including the system's "process serial number," a four-digit hexadecimal number listed in the first column. Let's say the program we want to test has a process serial number of 2008. Try typing "file -p 2008" to see a list of all files that were opened by the program with process serial number 2008. Quitting that program closes all those files. Now you know. (By the way, the "-p" option in "file" is new in MacsBug 6.5.4a4.)

We're advised that MacsBugApp has some problems if virtual memory is turned on, and our sta didn't find it to be without flakiness--sometimes the commands they typed disappeared as they typed them (white text on a white background, we suspect), and the sta saw a few strange crashes as well. If you want to use MacsBug for crash protection and other poking around, it's OK to install it and use the "real" MacsBug instead of MacsBugApp, especially if you have trouble. It's just nice to have choices.

With these preliminaries aside, let's get back to the fun stu.

Partitioning Memory

You're probably familiar with the concept of partitioning large disk drives into smaller virtual "volumes." When you do this, even though all the volumes are stored on one device, every part of the Mac OS above the very-low-level SCSI Manager treats them as separate devices. A program that goes wacko and erases a "disk" can't erase the other partitions. More importantly, for HFS users, smaller volumes reduce the size of each volume's allocation block, saving disk space in the long run.

Whether you know it or not, the RAM in your machine is "partitioned" as well.

You control how much RAM each program gets through the memory allocation in an application's "Get Info" box in the Finder. The numbers, as you may know, are actually stored in a 'SIZE' resource in each application's resource fork, which is how you can change them for programs like the Finder, or for extensions that spawn background processes but aren't applications themselves. As always, don't try this unless you're sure of what you're doing, or unless you have multiple duplicate (and redundant) backups.

To see the way the system has doled out your RAM, enter MacsBug or MacsBugApp and type "hz", for heap zones. A heap is the rough equivalent of a memory partition, with the notable exception that heaps can contain smaller, sub-heaps. (Disk partitions can't contain smaller disks, unless you think of something like a DiskCopy image as a "partition" even though it's just a file. Heaps can be more recursive than that.)

The list of heap zones, as they're sometimes called, is numbered for easier reference. The first one will probably be what's informally called the system heap, and is the area of memory where the Mac OS tries to allocate most of the RAM it needs for components, the operating system itself, and other fundamental memory-chewers. Inside the System Zone are other zones, probably unlabeled, although MacsBug does recognize one on some machine as the "ROM read-only zone," where parts of the ROM are made to look like a read-only heap zone. The indenting shows you the heap containment level-- those listed to the right of the System Zone's line are contained within the system zone. The starting and ending addresses for these sub-heaps show that as well--all of them start and end within the System Zone's starting and ending addresses.

The application you were using when you broke into MacsBug (or MacsBugApp itself if using the application version) will probably be identified as the "ApplZone," meaning the zone for the current application. It's probably also listed as the "TheZone," meaning the target of current Memory Manager calls, and "TargetZone," meaning the heap zone that MacsBug will operate on by default. Want proof? Type "hx" and MacsBug responds that the target heap is now the System zone. Type "hz" again, and sure enough, "TargetZone" now appears next to the system heap (which should be heap #1 on all systems).

Why deal with all this arcana? MacsBug isn't a general-purpose memory profiling tool like Bob Fronabarger's Memory Mapper. Memory Mapper looks at the same kind of information as MacsBug and draws you a chart showing where each application is residing in RAM--but Memory Mapper also shows you free spaces. The "hz" command doesn't do that, unless you look at the ending address of each heap and compare it to the starting address of the next one. Memory Mapper makes explicit what MacsBug leaves as an exercise to the reader.

What MacsBug can do that Memory Mapper cannot is show you what's going on inside one of those application heaps. Launch an application that's very small-- we suggest Note Pad. (We were going to suggest Calculator, but our sta tells us its heap is always damaged for some reason, something MacsBug can detect as we'll tell you shortly.) When in Note Pad, break into MacsBug (MacsBugApp won't do for this one, sorry). Make sure CurApName, on the middle left, says "Note Pad," but you can continue for a moment if it doesn't.

Type "hd" to dump the heap. You'll see a listing of every chunk of RAM allocated out of Note Pad's heap. Here's how the start of such a display might look as shown in Listing 1:

Displaying the "Note Pad" heap at 0738AA70

	Start	Length	Tag	Mstr	Ptr	Lock	Prg	Type	ID	File	Name
*	0738AAB0	00000044+00	N								
*	0738AB00	00000100+04	N								

* 0738AC10	000002C2+12	R	0738ABF4	L	CODE	0004	6C54	Main
* 0738AEF0	00004334+10	R	0738ABF0	L	CODE	0002	6C54	app
* 0738F240	00003E26+0E	R	0738ABEC	L	CODE	0003	6C54	Other
* 07393080	00000242+12	R	0738ABE8	L	CODE	0001	6C54	
* 073932E0	000000BE+06	N						
* 073933B0	00000014+10	N						
* 073933E0	00000126+0E	N						
* 07393520	0000009C+18	N						
* 073935E0	00000100+04	N						
* 073936F0	0000009C+08	N						
* 073937A0	00000100+04	N						
* 073938B0	000006F9+0B	N	07393FD0	000002D4+00	F			
073942B0	000001EC+08	R	0738ABE0		PICT	0080	6C54	8 bit
073945F0	0000001D+07	R	0738ABD4		CNTL	0081	6C54	Size
07394620	0000001D+07	R	0738ABD0		CNTL	0080	6C54	Font
07394650	0000000C+08	R	0738ABCC		ALRT	0085	6C54	Error
07394670	00000080+04	R	0738ABC8	P	ICON	0080	6C54	
07394700	00000723+11	R	0738ABF8					
07394E40	000000A0+04	R	0738ABFC					
07394EF0	0000001B+09	R	0738ABC4	P	WIND	0080	6C54	
07394F20	0000029F+15	R	0738ABC0		MENU	0080	6C54	
073951E0	00000015+0F	R	0738ABA0		DLOG	0083	6C54	Delete

Listing 1 - Sample Heap Dump of Note Pad

The "Tag" field in the third column tells what kind of RAM chunk this is. Most memory is divided into relocatable chunks--through a bit of double-indirection, the Mac OS can move the contents of this block of RAM to a new location, but the program can still find it. Those blocks are tagged "R," for relocatable. Relocatable blocks can be temporarily locked so they can't move around for a while, and that's indicated in the sixth column, titled "Lock." For example, the third memory chunk in this list is relocatable but locked. That means the Memory Manager can't move it to a new location to find room in filling new memory allocation requests.

Filling requests for memory is what the Memory Manager is all about, and it goes to great lengths to jam requests as close together as possible while not spending millions of CPU cycles doing it. For example, blocks of memory as allocated are always a multiple of sixteen bytes to make things easier on the microprocessor. That's what the "+" in the length column means--each block is padded past the requested length to make things easier. For example, the third block in the list is 704 bytes long (0x2C2), and it's padded by eighteen extra bytes to get 722, which works better for this system. It doesn't like to pad by one or two bytes, possibly to help insure against memory trashing problems if a program writes one or two bytes beyond the end of a block (a common enough problem). In fact, the only blocks you see in the list with absolutely no padding are the first one, which contains the heap zone header, and one down the list with the tag "F", meaning it's free space and not allocated memory.

See the dots in the left hand column (represented as starson this Web page)?

Those indicate blocks that can't move at all, or movable blocks that have been locked. When the Memory Manager gets a request that it can't fulfill, it tries to move blocks around to make things work. For example, suppose 500 bytes were free on one side of a relocatable 200-byte chunk of RAM, and 400 bytes were free on the other side of the same chunk. If the

program then requested a 700-byte chunk of RAM, presuming this was all the free space available, the Memory Manager couldn't do it.

However, if it swapped the relocatable 200-byte chunk with either of the free blocks, the two free blocks would merge into one larger 900-byte free space, and that's more than enough room to fill the request.

If the 200-byte allocated chunk had been locked, however, the Memory Manager would have been stuck. To prevent problems like this, programmers try to keep all of the unmoving blocks either at the very top or very bottom of the heap, leaving everything in the middle free to shuffle around as necessary. Locked blocks in the middle of the heap can permanently fragment it so that future allocations fail when room is really available. That, unsurprisingly, is why it's called fragmentation.

The "hd" command in MacsBug can show you how fragmented a given heap is.

A programmer could use this information to try to reduce the fragmentation, but there's not much you can do except note that it's there. Well, maybe there are some things. In Listing 1, the rightmost columns are often filled with resource types, resource IDs, and file numbers for where resources came from. If you see a large number of resources associated with some plug-in module you can unload, you might be able to reduce memory usage and fragmentation, but that's a long shot. It's mostly informational, and if your heap is really fragmented, the information is "you're toast."

If CurApName did not read "Note Pad" when you broke into MacsBug, the heap you tried to dump was probably some other heap. Use the "hz" command to find Note Pad's heap by number (in the leftmost column). If it's number #13, for example, type "hx #13" to switch to it. Then try the "hd" command. You may be surprised to see a lot of "resource not found" identifiers on the right side instead of neat and precise listings of type and ID as shown in Listing 1. That's the cooperative multitasking of the Mac OS showing itself. The Mac OS application model was designed when the machine only ran one program at a time, so each program has certain properties that it thinks are global. One of those is the resource chain, or the list of open files where the system looks to find requested resources. Each application has its own chain--it wouldn't be appropriate for Note Pad's request for 'pict' resource 128 (0x80 in hexadecimal) to come up with that resource from Netscape Communicator if it happened to be open, so the system makes sure that Note Pad's resource chain is unaffected by other applications. Here you see the results--when dumping Note Pad's heap during another application's time (as shown by CurApName), Note Pad's resources aren't available, so MacsBug can't display their real type and ID values.

An easy way around that is to type the macro "WNE". This tells MacsBug to resume normal execution, but stop again the next time a program gives up control with the core system routine "WaitNextEvent." If Note Pad was frontmost when you entered MacsBug and yet it still wasn't listed in CurApName, the "WNE" macro will break again when Note Pad is in control. Pretty nifty. Using it repeatedly will typically cycle through programs on your system that are calling WaitNextEvent. That will typically catch all of the old-style 68K applications on your system. PowerPC code, or CFM-68K applications (MDJ 1997.12.02), use a newer style of Mac OS access called "transition vectors" instead. The "WNE" macro won't catch those, but you can define a PowerPC-style macro that will with the following command:

```
mc wnep "tvb WaitNextEvent '; tvc WaitNextEvent'; G; # reenter MacsBug on next
WaitNextEvent transition vector call"
```

You'll have to include this in a MacsBug preferences file to get it to stick around permanently, and we'll discuss that later. Type "help WNE" if you want to see how similar this is to the existing "WNE" macro.

What Can You Do With This?

Examining heaps is typically a task for programmers, but you can glean a few things from the available commands if you're a reasonably technical user. Let's apply some of this knowledge to a practical situation.

Suppose a program you're heavily using is claiming no more memory is available, even though other tools (like "About This Computer") reveal plenty of allegedly free space. If you can get to that program's heap in MacsBug or MacsBugApp, the "ht" (heap total) command will quickly show you how many chunks of free space are available, and the total number of bytes in all those chunks. That will generally coincide with what other tools told you. However, looking at the dots on the left side of the "hd" display can show you if the heap is horribly fragmented or not. If it is, then closing documents or quitting and relaunching the program should give you a new lease on life. If it's not, there's not much you can do, but you do know one thing--adding more memory to the application in the Finder's "Get Info" window might not help. After all, if the application already has 5MB of free space, verified by MacsBug, bumping that to 10MB probably won't solve the problem. Use the "log" command in MacsBug to record the "hd" output--you can append it to the end of a "StdLog" file if you want--to help the programmers figure out what's going wrong on your system. It might not be sufficient information, but it's more than they'd have if you just wrote to the company and said "How come your stupid program doesn't work?"

If the program is behaving weirdly, try the "hc" command to check the heap. If a program has trashed memory in a way that the Memory Manager's data structures are no longer valid, "hc" will usually tell you. If you find your heap has been corrupted, tread very carefully. Save what work you can, but be aware that things could get worse as you save. Do as little as possible to preserve what's necessary before quitting the program, and consider restarting your system. The "hc all" command supposedly checks all heaps, but in our experience it tends to skip some of them. Note that memory can be trashed without affecting the heap structures, so "hc" won't definitively tell you if any corruption has occurred. But if "hc" in MacsBug 6.5.4a4 says something is damaged, it probably is. (Earlier versions sometimes listed special heap zones, like the ROM read-only zone, as damaged when they were just defined differently. MacsBug 6.5.4a4 is better at detecting such things.)

There's an exception here: If you broke into MacsBug in the middle of a Memory Manager call, "hc" may report a corrupted heap simply because the call needs to complete. Try the "WNE" or "WNEP" macros to regain control when things should be fine.

Processing Processes

If you just want to see a list of all running processes, you don't have to mess with heap zones. In MacsBug (or MacsBugApp), type "ProcInfo". This gives you a list of all currently running programs, or processes, on your system. The first few lines might look like this:

Displaying Process Information

PSN	Process Name	Size	Free	HeapAt	Type	Crtr	Status
2003	Web Quick	000B2000	0003CAC0	07A15900	appe	ANU!	BgOnly

```

2005 Finder          000EC400 0006F5F0 0788CE00 FNDR MACS Bkgnd
2006 QuickKeys Toolbox 0004D800 0001C2A0 078252B0 INIT IACi BgOnly
2008 DragThing 2.1  00101000 00095B40 076DB800 APPL Dock Bkgnd
200A Desktop PrintMon. 00023170 0000C410 07699E70 APPL prmt BgOnly

```

Listing 2 - A Partial ProcInfo Production

The process serial numbers, or "PSN" column, aren't necessarily consecutive because you may have launched and quit some programs--or some might have launched and quit automatically. Of the five processes listed here, three of them are listed as "BgOnly," meaning they have no user interface whatsoever, and are sometimes called faceless background applications or FBAs. Web Quick's FBA receives Apple events from Web browsers as you surf the net, and communicates with the menu installed by the Web Quick extension to tell browsers where you want to go--only applications can send and receive Apple events, but an FBA is an application. QuickKeys Toolbox handles Apple event communication for CE Software's QuickKeys--you can see from the "Type" column that QuickKeys Toolbox is actually an extension, or 'INIT', and not a real application file. Desktop PrintMonitor communicates with the Finder to handle background printing. The Finder draws the windows and controls you see, but Desktop PrintMonitor tells the Finder what to draw.

The only semi-normal application in this list is DragThing 2.1, so it's what we'll examine. The "Size", "Free" and "HeapAt" columns discuss the state of DragThing's RAM allocation. All the numbers are in hexadecimal, as MacsBug prefers, but you can convert to decimal simply by typing them on the command line. If you do, you'll find that DragThing 2.1's memory starts at address 12,463,140, that it has 1,052,672 bytes allocated to it (which MacsBug dutifully reports as "just over 1M", meaning one megabyte), and of that allocation, 613,184 bytes ("between #598K and #599K") are free. DragThing looks to be in good shape on this system. In fact, looking at the program's bar in "About This Computer" would graphically show almost the same thing.

The display from "ProcInfo" is generally much less complete than that of "hz" or "hd", but it can give you all the information you need if you're not sure what's going on. And, as mentioned earlier, you can now use a process serial number like 2005 in the command "file -p 2005" to see what files that process (the Finder in Listing 2) has open.

More On Crash Recovery

If you're a bit more comfortable with MacsBug now, there are other ways you can use the debugger to help you during a crash. Here are some of our favorites.

Disabling User Breaks

You'll occasionally find software--especially beta-level software--that appears to crash with "Unimplemented Trap" instructions with the instructions "A9FF" or "ABFF". These are Mac OS system routines designed to invoke MacsBug, and if MacsBug isn't there to implement them, the traps (a special type of 68K instruction often used on the Mac to refer to system routines) aren't implemented. If MacsBug is present, you'll enter it with the notation "User break" at wherever the code was located. Trap A9FF is "Debugger" and just enters MacsBug. Trap "ABFF" is called "DebugStr", for "debugger string," and it displays a text message beneath the "user break" notice. You can see all these again later in any session by using the "HOW" command, which reminds you how you got into MacsBug in the first place.

User breaks are very handy for programmers. The microprocessor executes a few million instructions per second, and trying to press Command-Power exactly where you want to stop is a lost cause. Instead, developers insert these instructions into their code and stop exactly where they want to stop, so they can poke around and find problems and examine situations. It's all very cool, unless the programmers accidentally leave some of the MacsBug calls in a program shipped to customers. Even if you have MacsBug installed, you probably don't want the software to stop every so often with a message that means nothing to you.

Fortunately, you have an alternative. The "dx" command disables user breaks.

It's a toggle, so entering MacsBug again and typing "dx" turns them back on. Or you can type "dx on" or "dx o ", or the new "dx now" to see the current state. Once user breaks are disabled, any stray Debugger or DebugStr calls won't bother you until you explicitly turn them back on or restart. That should tide you over until the developers can fix the bug.

Finding Lost Data

This is significantly trickier, but it can save your bacon in extreme emergencies. What if you've just entered data that you can't replace and the system crashes?

Even a successful "ES" will quit the program and you'll lose it. There is a small chance you can find it, if you can remember part of it. For an example, let's go back to Note Pad. Enter the phrase "bicarbonate of soda" on one page, and then enter MacsBug (or MacsBugApp if you're just playing around).

MacsBug has a "find" command that will search memory, but it's not overly- friendly. You must tell it where to start looking and how many bytes to search, in addition to the item you're trying to find. Use the "ht" command to get the total of Note Pad's heap. At the top, MacsBug will tell you that it's totalling the Note Pad heap "at" some address in hexadecimal. That's where to start. At the end of the "ht" display, an eight-digit hexadecimal number in the "total of block sizes" column gives the size of the heap. That's how many bytes to search. In our example, the Note Pad heap was at 0738AA70 (as shown at the top of Listing 1, too), and the heap size was 00031EFC bytes. You can convert these numbers to decimal to see what they're like, but it's not necessary.

We're ready to try it. Suppose you can remember that the text you lost had something to do with "carbon." Our sample would use this command, plugging in the Note Pad starting values and size:

```
F 0738AA70 00031EFC 'carbon'
```

Listing 3 - Sample Find command

The leading zeroes on numbers aren't necessary; they just make the values look like what MacsBug displayed. This command searches all the memory in Note Pad's heap for the text "carbon."

Use single quotes and not double quotes. If MacsBug can find the text you're after, it will display memory showing it. The memory is dumped in raw hexadecimal format, with an ASCII translation on the right side. The address at the left side of the display shows where it was found. It might look something like this:

```
Searching for 'carbon' from 0738AA70 to 073BC96B
07396812 6361 7262 6F6E 6174 6520 6F66 2073 6F64  carbonate of sod
```

Listing 4 - Sample Find output

MacsBug returns the sixteen bytes of memory that start with the found data, so the address on the far left (in this example, 07396812) is exactly where your data was found. If that's in the middle of what you're searching for, you'll want to back up somewhat. Try subtracting a few hundred bytes from that address. Use the "log" command to start recording to disk, and then use the "dma" command (which stands for "display memory in ASCII format") to dump the memory you want, with a command like "dma 07396812-#300". Press Return a few times to dump more memory. When you have all you think you're going to get, type "log" to close the file.

The text won't be in perfect format, but with luck, you can use copy and paste to salvage much of it. Please note, however, that this is an i y maneuver at best. Programs don't always store text data in contiguous chunks--if your word processor had hyphenated "bicarbonate" after the "r," it might have stored the two parts of the word in separate places, and searching for "carbon" wouldn't have worked. Also note that numerical data is almost never stored in ASCII format except in text files, so trying to find a number like "5.95" in a spreadsheet is almost certainly a lost cause. This won't necessarily cure problems you have--but if it's going to be a tremendous problem to reconstruct the data, it might be worth a try. Don't be jinxed just because certain MWJ sta ers were doing exactly this in Silicon Valley in 1989 when the Loma Prieta earthquake struck and killed the system's power. Some crashes were just meant to be.

Cheap Auto Rebooting

This one's a bit more annoying, but it can be a measure of protection for those who run servers or other unattended systems. Now that the "RS" command is a little bit better in MacsBug 6.5.4a4, it will usually succeed in restarting a crashed system. "RB" works even more often, although it may require volumes other than the boot volume to go through the system's MountCheck routine, verifying that everything is fine even though the disk wasn't taken o ine properly at shutdown. We'll use "RS" here, but you could use "RB" if "RS" causes problems.

Can MacsBug help if you have a server machine that needs to restart automatically? Yes, it can, in a non-subtle way. We've already seen macros, which are groups of MacsBug instructions executed as a single unit. For example, "StdLog" is a macro that dumps a whole bunch of system information to a file. Type "help StdLog" to see what all it does.

Although it's not widely known, MacsBug has a special macro called "EveryTime" that is executed, if it's defined, every time you enter MacsBug. The simple answer is to define an "EveryTime" macro that executes the command "RS", insuring that every actual crash forces the system to reboot. Note already that this won't help if the server hangs or locks up--only if it really crashes.

There's another catch, though--MacsBug won't execute the "EveryTime" macro the very first time it's entered. That's what the "FirstTime" macro is for, and it works similarly. The simple answer here is to define a "FirstTime" macro that says nothing but "G", the command to resume normal execution. If a FirstTime macro is defined, MacsBug will stop during the boot process and execute it, so defining this macro gives you a brief pause during startup

while you enter and exit MacsBug. The next time you enter MacsBug, presumably because of a crash, it will execute the "EveryTime" macro--the "RS" command, restarting the system.

The combination means that every time after the first automatic entry, MacsBug will restart the system if you just enter it. Any crash, therefore, reboots the system. If the system then crashes on boot, you have a nasty loop going, but what do you want? MacsBug is free. To make this work, you'll have to define the "FirstTime" and "EveryTime" macro in a file, preferably one you drop in the "MacsBug Preferences" folder. Create a new file in ResEdit (or Resorcerer, which we use) and create a new resource in that file of type 'mxbm', or "MacsBug Macro." Both MacsBug itself and the "Debugger Prefs" file contain a template that will help you add a name and an expansion for the macro as shown here--the name is either "FirstTime" or "EveryTime", the expansion is "G" (for "FirstTime") or "RS" (for "EveryTime").

A Good Starter Set

These articles have barely scratched the surface of MacsBug functionality--but that's because (stop us if you've heard this) MacsBug is a programmer's tool. The stuff it does best, like stepping through code and gaining control at specific places in a program--just aren't useful items to non-programmers. That doesn't mean that MacsBug can't do you well; it just means that non-programmers won't understand much of what MacsBug does. You shouldn't feel intimidated by this--it's by design. Non-programmers don't understand some of the programming-specific text functions in BBEEdit, either, but that doesn't mean it can't help you design killer Web sites.

There's lots more you can do. In MacsBug 6.5.4a4, try typing 'API StartupDispatch' to see selectors for the new routines added in Mac OS 8.1 to manage the startup process, inserted explicitly to help non-US systems load extensions in the right order on HFS Plus disks (MWJ 1998.02.02). This doesn't help very much, because no one outside Apple and a few developers (like Casady & Greene) have documentation for the routines yet, but you can find out what they're called. (Editor's note: These routines are now publicly defined by Apple.)

Or you can break into MacsBug in an application with lots of open windows and type the macro "WindList," which uses low-memory values to present a formatted dump of all the window records in that application. Try this in the Finder, pressing Return after each window to get the next, and you'll eventually run into the large, irregularly shaped window known as "Desktop," in case you wondered how that worked. Or type "thePort" to see the current QuickDraw graphics port disassembled for your pleasure.

MacsBug's ability to format data in memory along certain lines is quite powerful, although not quite as powerful as Quadrivio's General Edit, a program that can display any chunk of memory using a powerful formatting language that allows for conditional display, pop-up menus and more besides. These are valuable features for programmers, who can often disassemble complicated structures in memory using General Edit and source code from their programs to define the structures. Non-programmers usually couldn't care less about such things, since it typically takes someone with programming skills to diagnose problems in memory-based structures, repair them on-the-fly, and continue using the system. On the other hand, General Edit can apply the same deconstruction skills to files on disk, an area MacsBug doesn't touch. General Edit Lite handles many of the same tasks but without the advanced structure construction language or the ability to mess with individual chunks of memory. Still, General Edit Lite will be a good tool for your technical arsenal; programmers and very technical users will want to seriously consider the \$200 General Edit for managing the content of binary files like never before.

Still, we think MacsBug can be quite useful for non-programmers in the situations, as this starter set of Nifty MacsBug Tricks shows. We urge you to be cautious with MacsBug--just as messing up one character in an HTML file can ruin a Web page, changing things on your system with MacsBug can cause more problems than you might realize, and that's why we've deliberately stayed away from the "sm" command and its variants, the ones that set memory values instead of just displaying them as "dm" does. We've kept to those commands that can make already-bad situations a little better, and those that can give you more information without actually changing anything. Making it useful is up to you.

Tips and Tricks for MacsBug

All of us who program on the Macintosh have lots of little tricks that we use to get our job done. Most of these are passed around via e-mail and the Usenet News, if they are passed on at all. In an effort to collect all these in one place for the benefit of all, I've created this page. Please submit your tips to crawford@scruznet.com and I will add them to this page - with proper attribution of course.

Contents

- Where to get Macsbug
- Are there books that teach how to use Macsbug?
- Debugging Software on the Macintosh
- How to get Macsbug help for free
- Breaking While a Particular Application is Executing
- Logging Program Execution
- Logging Data in Real Time
- Using Cursors to Trace Program Execution
- Using Touch and Go Breakpoints with Two Monitors
- Twiddling Pixels in the Menu Bar
- Forcing Testers to Use Macsbug During Beta Testing
- The Developer University Debugging Class
- Links to Other Macsbug Pages

Where to get Macsbug

Macsbug is available via anonymous FTP from Apple Computer from [here](#) or [here](#). It is included with the two books below, but you will want to get the latest version, especially if you are using a PowerPC.

Are there books that teach how to use Macsbug?

Yes, among them are:

- o Othmer and Strauss, Debugging Mac Software with Macsbug, Addison-Wesley 1991, ISBN 0201570491. Macsbug included on disk.
- o Apple Computer Inc., Macsbug Reference and Debugging Guide version 6.2, Addison-Wesley 1991, ISBN 0201567687. Macsbug 6.2 included on disk.

While you don't need to know how to write assembly code to use Macsbug effectively, you will need to know how to read and understand it. Thus you will also need to get assembly code reference manuals, such as:

- o Kacmarcik, Optimizing PowerPC Code, Addison-Wesley 1995, ISBN 0201408392.
- o Motorola, M68000UM/AD MC68000 8/16/32-Bit MPU User Manual, Motorola Literature Distribution, 1991.

All of these books may be ordered from the Computer Literacy Bookstore.

How to get Macsbug help for free

Contributed by Bill Coderre, bc@wetware.com

1. Install Macsbug and programmer's key
2. Reboot the machine. Don't start any apps yet.
3. Hit CMD-POWER
4. type "log Macsbug Help <RETURN>"
5. type "help<RETURN>"
6. Push space until done.
7. type "log<RETURN>"
8. type "g<RETURN>"
9. Find the file called Macsbug Help on your desktop. Open it with a text editor. Read it.

Breaking while a particular app is executing

Most applications call WaitNextEvent. While an application is executing, a Pascal string with the name of the application is placed at location 0x910. Thus the first four characters of the name itself begin at location 0x911. Suppose you want to break while SimpleText is active. Enter Macsbug and give the command:

```
atb WaitNextEvent 911^ = 'Simp'
```

then continue. You will shortly drop into Macsbug at SimpleText's WaitNextEvent call. This is particularly useful when debugging faceless background applications. If the application does not call WaitNextEvent, try GetNextEvent instead.

Logging Program Execution

Contributed by Darren Giles, Terran Interactive, mars@netcom.com

Always on the lookout for useful debugging tools & tips, I'd love to share ideas with others on the topic. I'll start out by offering a snippet I've found very useful -- hopefully others will do the same.

One thing that's really bugged me about MacsBug on PPC is that the stack crawl has become a much less useful tool. The snippet below gives you the ability to keep track of a list of the last significant points your program has visited. It's a list, not a stack, so you can also see patterns of execution.

Hardly rocket science, but useful & easy to add. Just call DEBUG_STUFF_INIT at startup, then insert a DEBUG_ENTRY wherever you want. To see what's up, especially during a bad hang, just dm [the address that DEBUG_STUFF_INIT dumped out at startup.]

The conditional compilation means that if you turn of debugging in your final build, the release version of the program won't have any of this in it.

```
[debugstuff.h]
#define DB_ROUTINES_NBR_ENTRIES 40
#define DB_ROUTINES_CHARS          16
typedef char                      db_routine_entry[DB_ROUTINES_CHARS];
#if DEBUGGING
void DEBUG_ENTRY (char *txt);
void DEBUG_STUFF_INIT (char *title);
#else
#define DEBUG_ENTRY
#define DEBUG_STUFF_INIT
#endif

[debugstuff.c]
////////////////////////////////////
////
#if DEBUGGING
void DEBUG_STUFF_INIT (char *title) {
    OSErr          myErr;
    char           txt[256];
    long           response;

    if (!MacBugInstalled()) {
        hi_notify ("MacBug is not installed... the debugging log will be
inaccessible.");
    }

    g_debug_entries= (db_routine_entry*) NewPtrClear
((DB_ROUTINES_NBR_ENTRIES+2) * DB_ROUTINES_CHARS);
    memset (&g_debug_entries[0], '=', DB_ROUTINES_CHARS);
    BlockMoveData (title, &g_debug_entries[0], strlen(title));
    memset (&g_debug_entries[DB_ROUTINES_NBR_ENTRIES+1], '=',
DB_ROUTINES_CHARS);
    sprintf (txt, "Debugging routine list is at 0x%lx:g", (long)
g_debug_entries);
    c2pstr (txt);
    DebugStr (txt);
}
#endif

////////////////////////////////////
////
//      This leaves a line in the debugging entry log.
//      For example, important enter/exit points of routines
////////////////////////////////////
////
#if DEBUGGING
```

```

void DEBUG_ENTRY (char *txt) {
    short          len;

    //      Move the previous entries down one
    BlockMoveData (&g_debug_entries[1], &g_debug_entries[2],
                  (DB_ROUTINES_NBR_ENTRIES-1) * DB_ROUTINES_CHARS);

    //      Clear the new space
    memset (&g_debug_entries[1], 0, DB_ROUTINES_CHARS);

    //      Copy in the new entry
    len= strlen (txt);
    if (len > DB_ROUTINES_CHARS) {
        len= DB_ROUTINES_CHARS;
    }
    BlockMoveData (txt, &g_debug_entries[1], len);
}
#endif

```

Hope this does someone some good.

- Darren

Darren Giles, Technical Director
 For info on Cleaner QuickTime compression, visit <http://www.terran-int.com>

Terran Interactive

Logging Data in Real Time

Contributed by Dave Stone, dstone@chem.utoronto.ca

I've also used conditional compilation to debug serial communications stuff being processed at interrupt time - something like

```

#ifdef DEBUG_MY_ROUTINE
    #define MAX_BUFFER 10000
    char bufffer[MAX_BUFFER];           //      or NewPtr it or something
    long bufCount = 0L;
#endif
.
.
.
#ifdef DEBUG_MY_ROUTINE
    if(bufCount < MAX_BUFFER) {
        bufCount ++;
        buffer[bufCount] = ch;         //      ch is a character read/written
through serial
port
    }
#endif

```

etc. Handy, because you can let it rip for a while to see if there is a consistent pattern in the errors in ch - in my case, a stream of Midi data through a very basic freeware Midi Driver.

Using Cursors to Trace Program Execution

Contributed by Tom Kimpton, Jostens Learning Corporation, tom@jlc.com

One technique that I have used in the past where dropping into the debugger wasn't an option, and logging wasn't getting flushed in time/took too long, was to create a bunch of cursors numbering 00 - 99, and made a call to set the cursor and return the number of the previous cursor:

```
routine1()
{
short oldCursor = setDebugCursor(15);
    ...
    (void) setDebugCursor(oldCursor);
}
```

This way when the machine froze, the cursor would tell me what routine it had frozen in.

Using Touch and Go Breakpoints with 2 Monitors

I had a bug in which the Mac would occasionally freeze during shutdown without the ability to get into Macsbug. It would only occur about once in twenty reboots.

The way I dealt with this was to borrow a display card and hook two monitors up to the Macintosh. You can use the Monitors control panel to select which monitor will be used for Macsbug (hold the option key and drag the happy Mac around).

I wrote a small application that just called ShutDownRestart(), and placed it in the Startup Items folder. Thus, when the Mac came up into the Finder it would immediately reboot. About every twenty minutes it would freeze.

If you define a macro named FirstTime in the Debugger Prefs file, Macsbug will execute it when it loads. I used a macro that was something like:

```
swap; atr; atb shutdownrestart ";atb Newhandle ";g";g";g
```

or some such. The swap command caused Macsbug to be permanently left on the second screen. That way when the crash occurred you could still see the last few things Macsbug did. The ";g" following the a-trap break commands tells Macsbug to continue after the break - this is a "Touch and Go" breakpoint.

One thing you can also do inside a touch and go breakpoint is set new breakpoints. I would take guesses on what traps might be called in the neighborhood of the crash, and have breakpoints set on them when ShutDownRestart was called.

Then I could leave the Mac rebooting on its own in the lab, and pop in every half an hour to check the log, adjust the breakpoints and start it up again.

The actual bug took about five months to find and fix.

Twiddling Pixels in the Menu Bar

Contributed by Dave Fleck, Wacom Technology Corp., dfleck@wacom.com

Here's my debugging tip.

I do drivers, and you just plain can't set a breakpoint in ADB completion routines (freezes the keyboard so MacsBug is worthless!).

So I throw one of the routines below into the routine to see when a piece of code gets executed.

What does it do? It "lights up" a bar (length dependant on screen resolution) in the menu bar. So if you DotToggle(300); you get a flashing short line in the menu bar.

```
void DotOn(long where) {
    long *dot;
    dot = (long *) (LMGetScrBase() + where);
    *dot |= -1;
}
void DotOff(long where) {
    long *dot;
    dot = (long *) (LMGetScrBase() + where);
    *dot &= 0;
}
void DotToggle(long where) {
    long *dot;
    dot = (long *) (LMGetScrBase() + where);
    *dot ^= -1;
}
```

dave

```
-----
Dave Fleck   email:dfleck@wacom.com   phone:360-750-8882x154
Wacom Technology Corp.                 sales@wacom.com
501 S.E. Columbia Shores Blvd, #300   support@wacom.com
Vancouver, WA 98661                   WWW/FTP:wacom.com
-----
```

Forcing Testers to Use Macsbug During Beta Testing

Contributed by Harold Ekstrom, the ag group, inc., ekstrom@aggroup.com.

Don't you just hate it when beta testers say "it crashes" but don't give you any more information? First, tell them to use the "stdlog" command in MacsBug, then force them to install MacsBug by checking for it during your program's initialization:

```
--- DebugUtils.h ---

#pragma once
```

```

// Debugger types.
typedef enum DebuggerType {
    kNoDebugger,
    kMacBug,
    kTMON,
    kOtherDebugger
} DebuggerType;

Boolean    GetDebuggerInfo( DebuggerType *outDebuggerType,
                          UInt16 *outDebuggerSignature );

--- DebugUtils.c ---

// Private defines for some low memory globals.
#define MacJump      ((Ptr *)0x0120)    // MacsBug jumptable [pointer].
#define MacJumpByte  ((UInt8 *)0x0120) // MacsBug flags in 24 bit mode [byte].
#define MacJumpFlag  ((UInt8 *)0x0BFF) // MacsBug flag [byte].

// Debugger flag bits.
#define kDebuggerInstalledBit  5

//
-----
//
//
-----

Boolean
GetDebuggerInfo(
    DebuggerType * outDebuggerType,
    UInt16 *      outDebuggerSignature )
{
    Boolean theResult = false;
    SInt32 theResponse;

    // Initialize return values to defaults.
    *outDebuggerType = kNoDebugger;
    *outDebuggerSignature = ' ';

    if ( Gestalt( gestaltAddressingModeAttr, &theResponse ) == noErr ) {

        UInt16 theDebugFlags;

        // As documented in the "Macsbug Reference & Debugging Guide", page 412
        // if we have a 32 bit capable Memory Manager, debugger flags are at 0x0BFF
        // if we have a 24 bit capable Memory Manager, debugger flags are at 0x0120

        if ( (theResponse & (1L << gestalt32BitCapable)) != 0 ) {
            theDebugFlags = *MacJumpFlag;
        } else {
            theDebugFlags = *MacJumpByte;
        }

        if ( (theDebugFlags & (1L << kDebuggerInstalledBit)) != 0 ) {

```

```

Ptr theDebuggerEntry;
Ptr theROMBaseWorld;

// There is a debugger installed.
theResult = true;

// Get the debugger entry.
theDebuggerEntry = StripAddress( *MacJmp );

// Get the ROM base.
theROMBaseWorld = StripAddress( LMGetROMBase() );

// Compare the debugger entry to the ROM base.
if ( theDebuggerEntry < theROMBaseWorld ) {

    UInt16 **theDebuggerWorld;

    // It's not a ROM based debugger.
    // Get the debugger world.
    theDebuggerWorld = (UInt16 **) StripAddress( theDebuggerEntry - sizeof(Ptr) );

    // Get the debugger signature.
    *outDebuggerSignature = **theDebuggerWorld;

    // Get the debugger type.
    switch ( *outDebuggerSignature ) {

        case 'MT':
            *outDebuggerType = kMacsBug;
            break;

        case 'WH':
            *outDebuggerType = kTMON;
            break;

        default:
            *outDebuggerType = kOtherDebugger;
            break;

    }

}

}

}

return theResult;
}

```

Check for a low level debugger like this:

```
#if BETA_VERSION
```

```

DebuggerType    theDebuggerType;
UInt16          theDebuggerSig;
if ( !GetDebuggerInfo( &theDebuggerType, &theDebuggerSig ) ) {
    HaltRotateCursor( gRotateCrsr );
    StopAlert( go_get_macsbug_alrt, nil );
    ExitToShell();
}
#endif

```

The Developer University Debugging Class

Contributed by Malcolm Teas, Blaze Technology, mhteas@btech.com

As the instructor and developer of Apple's Developer University class called "Macintosh Debugging Tips and Techniques" I would like to make sure your tips page references this class.

This class is centered around MacsBug as the easiest to learn low-level debugger. It also covers a multitude of low-level topics like memory maps, subroutine calling protocols, code segments and code fragments, reading (and understanding) assembler for 68K and PPC, and many more areas. One key area is how to avoid bugs in the first place. All the information you need to be able to debug software at the low-level.

The class is available from Apple's Developer University.

Another thing I want to mention is the version number of the most current MacsBug is 6.5.3 (as of this writing). This version includes the PPC commands and features.

[I have taken this class and recommend it highly - Mike]

MacsBug DCMDs

Where [addr | trap]

Display information about the address or trap.

If no parameter then use PC as the address.

Vol [vRefNum|drvNum|"vol name"]

Displays volume information for the given vrefnum, volume name or all mounted volumes. Flags are D/d=Dirty, S/s=Software locked, H/h=Hardware locked.

VBL

Lists tasks in the regular and slot VBL queues.

thing ["thing type"]

Displays thing information for the given thing type or all known things.

StopXPP

Closes all open .XPP sessions.

RD [-c] [-s] [-o] [-f ref#] [-i id] [[-t] 'xxx'] [-h hndl] (Resource Display)

Dumps resource information.

Options:

- c Show resource chain
- s Show short resource chain (no fonts below system)
- o Show offsets of resources (from start of res data)
- f <refNum> Only show resources from this file (1=ROM)

-i <resID> Only show resources with this ID
 -t 'xxxx' Only show resources of this type
 -h <handle> Show resources using given handle (0=not loaded)

Attributes:

S/A = System heap / app heap
 P/p = purgeable / not
 L/U = locked / unlocked
 O/o = protected / not
 E/e = preload / not
 C/c = changed / not

Purged handles are marked with an "*".

Printf "format" arg...

Displays the arguments according to the format (no floating point).

ProcInfo

Displays information about all processes.

Status Flags: Front, Bkgnd, BgOnly, BgNoEvts (can't bkgnd).

Leaks [OnloffIDump]

Stack crawl info about likely memory leaks.

(by Bo3b Johnson, Greg Branche and Jim Murphy)

JumpTable [expr]

Display the jump table at expr. If no address is specified, the jump table is assumed to start at RA5+\$20.

Gestalt [-n] [[-s] 'xxxx'] (Gestalt selector display)

Displays Gestalt selectors and results.

Options:

-n Show installed selectors, don't call procedures
 -s 'xxxx' Don't display all selectors, just show this one

(Gestalt with no parameters calls and shows all selectors.)

FSInfo [fsid | -t] (File System info)

Displays File System Manager foreign file system information for the given file system ID [fsid], or all installed File System Manager foreign file systems. Use -t to show a small table.

Use the vol dcmd to get a list of volumes and their fsid numbers.

File [fRefNum | "file name" | -t "type"]

Displays file information on all open files, or for the given fRefNum, filename, or for all files of the given filetype. Flags are D/d=Dirty, W/w=writeable. "file 0" shows all open files except for fonts.

Error expr

Display text message corresponding to error number in expr.

Echo [params...]

Echo the command line parameters

Drvr [refNumInum]

Displays driver information for the given refNum or all installed drivers. Flags are B/b=Busy, H/P=Handle/Ptr, O/C=Open/Closed.

Drive [drvNumIdRefNum] (display info on drives)

Displays drive queue information for the given drive number, driver number, or all drives. Flags are L/l=Locked, E/e=Ejectable, R=Recently ejected, I/i=Inserted, D/S=Double/Single sided.

UNLOCK <Addr> <Count>- VM Unlock memory for the memory range specified in addr and count.

Note: This is the debugger unlock memory call.

SysTop - installs and locks a 4 byte handle

at the top of the system heap,
the handle is not installed

SysSwell <buffer size>- Installs and removes a buffer
in the System heap of specified size.

ssc [addr]
Displays the stack frame chain starting at addr (default is ra6).

Patch [I | O | T | S | P] (vers 1.0A3)
none - Check all vectors
I - Check Interrupt vectors
O - Check OSTRap vectors
T - Check TBTrap vectors
S - Save all vectors now
P - Print all vectors

Mlist -- List the menus installed.

LOCK <Addr> <Count>- VM Lock memory for the
memory range specified in addr and count.
Note: this is the debugger lock memory.

dizy - Installs and de-installs the discipline
which is contained in this code.
Dizy is NOT INSTALLED and is currently OFF

Otool Manual

NAME

otool - object file displaying tool

SYNOPSIS

otool [option ...] [file ...]

DESCRIPTION

The otool command displays specified parts of object files or libraries. If the, -m option is not used, the file arguments may be of the form libx.a(foo.o), to request information about only that object file and not the entire library. (Typically this argument must be quoted, ``libx.a(foo.o)'', to get it past the shell.) Otool understands both Mach-O (Mach object) files and fat file formats. Otool can display the specified information in either its raw (numeric) form (without the -v flag), or in a symbolic form using macro names of constants, etc. (with the -v or -V flag).

At least one of the following options must be specified:

- a Display the archive header, if the file is an archive.
- S Display the contents of the `__SYMDEF' file, if the file is an archive.
- f Display the fat headers.
- h Display the Mach header.
- l Display the load commands.

- L Display the names and version numbers of the shared libraries that the object file uses.
- D Display just install name of a shared library.
- s segname sectname
Display the contents of the section (segname,sectname). If the -v flag is specified, the section is displayed as its type, unless the type is zero (the section header flags). Also the sections (__OBJC,__protocol), (__OBJC,__string_object) and (__OBJC,__runtime_setup) are displayed symbolically if the -v flag is specified.
- t Display the contents of the (__TEXT,__text) section. With the -v flag, this disassembles the text. And with -V, it also symbolically disassembles the operands.
- d Display the contents of the (__DATA,__data) section.
- o Display the contents of the __OBJC segment used by the Objective-C run-time system.
- r Display the relocation entries.
- c Display the argument strings (argv[] and envp[]) from a core file.
- I Display the indirect symbol table.
- T Display the table of contents for a dynamically linked shared library.
- R Display the reference table of a dynamically linked shared library.
- M Display the module table of a dynamically linked shared library.
- H Display the two-level namespace hints table.

The following options may also be given:

- p name
Used with the -t and -v or -V options to start the disassembly from symbol name and continue to the end of the (__TEXT,__text) section.
- v Display verbosely (symbolically) when possible.
- V Display the disassembled operands symbolically (this implies the -v option). This is useful with the -t option.
- X Don't display leading addresses when displaying contents of sections.

- arch arch_type
Specifies the architecture, arch_type, of the file for otool(1) to operate on when the file is a fat file. (See arch(3) for the currently know arch_types.) The arch_type can be "all" to operate on all architectures in the file. The default is to display only the host architecture, if the file contains it; otherwise, all architectures in the file are shown.

- m The object file names are not assumed to be in the archive(member) syntax, which allows file names containing parenthesis.

Used with the -t and -v or -V options to start the disassembly from symbol name and continue to the end of the (__TEXT,__text) section.

- v Display verbosely (symbolically) when possible.

- V Display the disassembled operands symbolically (this implies the -v option). This is useful with the -t option.

- X Don't display leading addresses when displaying contents of sections.

- arch arch_type
Specifies the architecture, arch_type, of the file for otool(1) to operate on when the file is a fat file. (See arch(3) for the currently know arch_types.) The arch_type can be "all" to operate on all architectures in the file. The default is to display only the host architecture, if the file contains it; otherwise, all architectures in the file are shown.

- m The object file names are not assumed to be in the archive(member) syntax, which allows file names containing parenthesis.

GDB Manual

NAME

`gdb` - The GNU Debugger

SYNOPSIS

```
gdb [-help] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev]
    [-s symfile] [-e prog] [-se prog] [-c core] [-x cmds] [-d dir]
    [prog[core|procID]]
```

DESCRIPTION

The purpose of a debugger such as GDB is to allow you to see what is going on ``inside'' another program while it executes--or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- o Start your program, specifying anything that might affect its behavior.
- o Make your program stop on specified conditions.
- o Examine what has happened, when your program has stopped.
- o Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to debug programs written in C, C++, and Modula-2. Fortran support will be added when a GNU Fortran compiler is ready.

GDB is invoked with the shell command `gdb`. Once started, it reads commands from the terminal until you tell it to exit with the GDB command `quit`. You can get online help from `gdb` itself by using the command `help`.

You can run `gdb` with no arguments or options; but the most usual way to start GDB is with one argument or two, specifying an executable program as the argument :

```
gdb program
```

You can also start with both an executable program and a core file specified:

```
gdb program core
```

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

```
gdb program 1234
```

would attach GDB to process 1234 (unless you also have a file named ``1234'`; GDB does check for a core file first).

Here are some of the most frequently needed GDB commands:

```
break [file:]function
    Set a breakpoint at function (in file).
```

```
run [arglist]
    Start your program (with arglist, if specified).
```

```
bt    Backtrace: display the program stack.
```

```
print expr
    Display the value of an expression.
```

```
c    Continue running your program (after stopping, e.g. at a breakpoint).
```

```
next  Execute next program line (after stopping); step over any func-
```

tion calls in the line.

edit [file:]function

look at the program line where it is presently stopped.

list [file:]function

type the text of the program in the vicinity of where it is presently stopped.

step Execute next program line (after stopping); step into any function calls in the line.

help [name]

Show information about GDB command name, or general information about using GDB.

quit Exit from GDB.

For full details on GDB, see *Using GDB: A Guide to the GNU Source-Level Debugger*, by Richard M. Stallman and Roland H. Pesch. The same text is available online as the `gdb` entry in the `info` program.

OPTIONS

Any arguments other than options specify an executable file and core file (or process ID); that is, the first argument encountered with no associated option flag is equivalent to a `-se` option, and the second, if any, is equivalent to a `-c` option if it's the name of a file. Many options have both long and short forms; both are shown here. The long forms are also recognized if you truncate them, so long as enough of the option is present to be unambiguous. (If you prefer, you can flag option arguments with `+` rather than `-`, though we illustrate the more usual convention.)

All the options and command line arguments you give are processed in sequential order. The order makes a difference when the `-x` option is used.

`-help`

`-h` List all options, with brief explanations.

`-symbols=file`

`-s file`

Read symbol table from file file.

`-write` Enable writing into executable and core files.

`-exec=file`

`-e file`

Use file file as the executable file to execute when appropriate, and for examining pure data in conjunction with a core

dump.

-se=file

Read symbol table from file file and use it as the executable file.

-core=file

-c file

Use file file as a core dump to examine.

-command=file

-x file

Execute GDB commands from file file.

-directory=directory

-d directory

Add directory to the path to search for source files.

-nx

-n Do not execute commands from any ``.gdbinit'` initialization files. Normally, the commands in these files are executed after all the command options and arguments have been processed.

-quiet

-q ```Quiet''`. Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

-batch Run in batch mode. Exit with status 0 after processing all the command files specified with ``.x'` (and ``.gdbinit'`, if not inhibited). Exit with nonzero status if an error occurs in executing the GDB commands in the command files.

Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in order to make this more useful, the message

Program exited normally.

(which is ordinarily issued whenever a program running under GDB control terminates) is not issued when running in batch mode.

-cd=directory

Run GDB using directory as its working directory, instead of the current directory.

-fullname

-f Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a

standard, recognizable fashion each time a stack frame is displayed (which includes each time the program stops). This recognizable format looks like two `32' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two `32' characters as a signal to display the source code for the frame.

-b bps Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

-tty=device

Run using device for your program's standard input and output.

SEE ALSO

`gdb' entry in info; Using GDB: A Guide to the GNU Source-Level Debugger, Richard M. Stallman and Roland H. Pesch, July 1991.

Further documentation is available in /Developer/Documentation/Commands/gdb.

standard, recognizable fashion each time a stack frame is displayed (which includes each time the program stops). This recognizable format looks like two `32' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two `32' characters as a signal to display the source code for the frame.

-b bps Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

-tty=device

Run using device for your program's standard input and output.

SEE ALSO

`gdb' entry in info; Using GDB: A Guide to the GNU Source-Level Debugger, Richard M. Stallman and Roland H. Pesch, July 1991.

Further documentation is available in /Developer/Documentation/Commands/gdb.

COPYING

Copyright (c) 1991 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be included in translations approved by the Free Software Foundation instead of in the original English.

ANNEXE

Predefined PPC Register Names

Fixed-Point Register Conventions

Register Contents Usage

GPR0 Volatile Function prolog and epilog

GPR1 Nonvolatile Stack pointer

GPR2 Nonvolatile TOC pointer (also known as RTOC)

GPR3–GPR10 Volatile Arguments passed to a function or returned a value or pointer

GPR11–GPR12 Volatile Function prolog and epilog

GPR13–GPR31 Nonvolatile Storage for local variables

Floating-Point Register Conventions

Register Contents Usage

FPR0 Volatile Scratch area for a local function

FPR1–FPR13 Volatile Parameters passed to a function or returned a value

FPR14–FPR31 Nonvolatile Storage for local variables

Condition Register Conventions

Register Contents Usage

CR0 Volatile Scratch area or set by integer instruction record bit

CR1 Volatile Scratch area or set by floating-point instruction record bit

CR2–CR4 Nonvolatile Local storage

CR5–CR7 Volatile Scratch area

The assembler supports the names of these general-purpose registers as absolute expressions:

GPR0–GPR31 General-purpose registers

R0–R31 General-purpose registers (same as GPR*n*)

FP0–FP31 Floating-point registers

F0–F31 Floating-point registers (same as FP*n*)

The assembler also supports the specific uses for these general-purpose registers as absolute expressions:

SP Stack pointer (GPR0)
 RTOC Table of Contents register (GPR2)

These names are case insensitive. Also, any of the above register names may be equated to other name symbols by using the EQU or SET assembler directives.

The assembler recognizes the names of these special-purpose registers as absolute expressions:

ASR Address space register
 BAT0U–BAT5U Block address translation register (upper)
 BAT0L–BAT5L Block address translation register (lower)
 CTR Count register
 DAR Data access register
 DBATL Block address translation decrement register (lower)
 DBATU Block address translation decrement register (upper)
 DEC Decrement counter
 DSISR Data storage interrupt status register
 EAR External access register
 IBATL Block address translation increment register (lower)
 IBATU Block address translation increment register (upper)
 LR Link register
 RTC Real-time clock (upper + lower)
 RTCD Real-time clock divisor
 RTCI Real-time clock increment
 RTCL Real-time clock (lower)
 RTCU Real-time clock (upper)
 SDR1 Storage description register 1
 SPRG0–SPRG3 Software-use special-purpose registers
 SRR0–SRR1 Machine status save/restore registers
 XER Fixed-point exception register

PPC Mnemonics

Mac ASM Commands for PPC Processor

add[o][.]	rT,rA,rB	Add (without updating Carry)
addc[o][.]	rT,rA,rB	Add Carrying
adde[o][.]	rT,rA,rB	Add Extended
addi	rT,rA,s16	Add Immediate
addic[.]	rT,rA,s16	Add Immediate Carrying
addis	rT,rA,s16	Add Immediate Shifted
addme[o][.]	rT,rA	Add to Minus One Extended
addze[o][.]	rT,rA	Add to Zero Extended
and[.]	rA,rS,rB	AND
andc[.]	rA,rS,rB	AND with Complement
andi.	rA,rS,u16	AND Immediate
andis.	rA,rS,u16	AND Immediate Shifted
b[l][a]	addr	Branch
bc[l][a]	branch0n,crbT,addr	Branch Conditional
bcctr[l]	branch0n,crbT	Branch Conditional to CTR
bc[lr][l]	branch0n,crbT	Branch Conditional to LR
bctr[l]		Branch Unconditionally to CTR

bdnz[l][a]	addr	Branch if Decrement CTR is Not Zero
bdnzf[l][a]	crbT,addr	Branch if Decrement CTR is Not Zero and Condition False
bdnzflr[l]	crbT	Branch if Decrement CTR is Not Zero and Condition False to LR
bdnzlr[l]		Branch if Decrement CTR is Not Zero to LR
bdnzt[l][a]	crbT,addr	Branch if Decrement CTR is Not Zero and Condition True
bdnstlr[l]	crbT	Branch if Decrement CTR is Not Zero and Condition True to LR
bdz[l][a]	addr	Branch if Decrement CTR is Zero
bdzf[l][a]	crbT,addr	Branch if Decrement CTR is Zero and Condition False
bdzflr[l][a]	crbT	Branch if Decrement CTR is Zero and Condition False to LR
bdzlr[l]		Branch if Decrement CTR is Zero to LR
bdzt[l][a]	crbT,addr	Branch if Decrement CTR is Zero and Condition True
bdztlr[l]	crbT	Branch if Decrement CTR is Zero and Condition True to LR
beq[l][a]	[crT,]addr	Branch if Equal
beqctr[l]	[crT]	Branch if Equal to CTR
beqlr[l]	[crT]	Branch if Equal to LR
bf[l][a]	crbT,addr	Branch if Condition False
bfctr[l]	crbT	Branch if Condition False to CTR
bflr[l]	crbT	Branch if Condition False to LR
bge[l][a]	[crT,]addr	Branch if Greater Than or Equal
bgectr[l]	[crT]	Branch if Greater Than or Equal to CTR
bgelr[l]	[crT]	Branch if Greater Than or Equal to LR
bgt[l][a]	[crT,]addr	Branch if Greater Than
bgtctr[l]	[crT]	Branch if Greater Than to CTR
bgtlr[l]	[crT]	Branch if Greater Than to LR
ble[l][a]	[crT,]addr	Branch if Less Than or Equal
blectr[l]	[crT]	Branch if Less Than or Equal to CTR
blelr[l]	[crT]	Branch if Less Than or Equal to LR
blr[l]		Branch Unconditionally to LR
blt[l][a]	[crT,]addr	Branch if Less Than
bltctr[l]	[crT]	Branch if Less Than to CTR
bltlr[l]	[crT]	Branch if Less Than to LR
bne[l][a]	[crT,]addr	Branch if Not Equal
bnctr[l]	[crT]	Branch if Not Equal to CTR
bnelr[l]	[crT]	Branch if Not Equal to LR
bng[l][a]	[crT,]addr	Branch if Not Greater Than
bngctr[l]	[crT]	Branch if Not Greater Than to CTR
bnglr[l]	[crT]	Branch if Not Greater Than to LR
bnl[l][a]	[crT,]addr	Branch if Not Less Than
bnlctr[l]	[crT]	Branch if Not Less Than to CTR
bnllr[l]	[crT]	Branch if Not Less Than to LR
bns[l][a]	[crT,]addr	Branch if Not Summary Overflow
bnsctr[l]	[crT]	Branch if Not Summary Overflow to CTR
bnslr[l]	[crT]	Branch if Not Summary Overflow to LR
bnu[l][a]	[crT,]addr	Branch if Not Unordered
bnuctr[l]	[crT]	Branch if Not Unordered to CTR
bnulr[l]	[crT]	Branch if Not Unordered to LR
bso[l][a]	[crT,]addr	Branch if Summary Overflow
bsoctr[l]	[crT]	Branch if Summary Overflow to CTR
bsolr[l]	[crT]	Branch if Summary Overflow to LR
bt[l][a]	crbT,addr	Branch if Condition True
btctr[l]	crbT	Branch if Condition True to CTR
btlr[l]	crbT	Branch if Condition True to LR
bun[l][a]	[crT,]addr	Branch if Unordered
bunctr[l][a]	[crT]	Branch if Unordered to CTR

```

bunlr[l]      [crT]      Branch if Unordered to LR
clrslwi[.]   rA,rS,nBits,shift  Clear Left and Shift Left Word Immediate
clrlwi[.]    rA,rS,nBits      Clear Left Word Immediate
clrrwi[.]    rA,rS,nBits      Clear Right Word Immediate
cmp          crT,L,rA,rB      Compare
cmpi         crT,L,rA,s16    Compare Immediate
cmpl        crT,L,rA,rB      Compare Logical
cmpli       crT,L,rA,u16    Compare Logical Immediate
cmplw       crT,rA,rB      Compare Logical Word
cmplwi      crT,rA,u16     Compare Logical Word Immediate
cmpw        crT,rA,rB      Compare Word
cmpwi       crT,rA,s16     Compare Word Immediate
cntlzw[.]   rA,rS      Count Leading Zeros Word
crand       crbT,crbA,crbB  Condition Register AND
crandc      crbT,crbA,crbB  Condition Register AND with Complement
crclr       crbT      Condition Register Clear
creqv       crbT,crbA,crbB  Condition Register Equivalent
crmmove     crbT,crbA      Condition Register Move
crnand      crbT,crbA,crbB  Condition Register Not AND
crnor       crbT,crbA,crbB  Condition Register Not OR
crnot       crbT,crbA      Condition Register Not
cror        crbT,crbA,crbB  Condition Register OR
crorc       crbT,crbA,crbB  Condition Register OR with Complement
crset       crbT      Condition Register Set
crxor       crbT,crbA,crbB  Condition Register Exclusive OR
dcbf        rA,rB      Data Cache Block Flush
dcbi        rA,rB      Data Cache Block Invalidate
dcbst       rA,rB      Data Cache Block Store
dcbt        rA,rB      Data Cache Block Touch
dcbtst      rA,rB      Data Cache Block Touch for Store
dcbz        rA,rB      Data Cache Block Zero
divw[o][.]  rT,rA,rB      Divide Word
divwu[o][.] rT,rA,rB      Divide Word Unsigned
eieio       Enforce In-Order Execution of I/O
eqv[.]      rA,rS,rB      Equivalent
extlwi[.]   rA,rS,nBits,start  Extract and Left Justify Word Immediate
extrwi[.]   rA,rS,nBits,start  Extract and Right Justify Word Immediate
extsb[.]    rA,rS      Extend Sign Byte
extsh[.]    rA,rS      Extend Sign Halfword
fabs[.]     frT,frB      Floating-Point Absolute Value
fadd[.]     frT,frA,frB   Floating-Point Add
fadds[.]    frT,frA,frB   Floating-Point Add Single-Precision
fcmpo       crT,frA,frB   Floating-Point Compare Ordered
fcmpu       crT,frA,frB   Floating-Point Compare Unordered
fctiw[.]    frT,frB      Floating-Point Convert to Integer Word
fctiwz[.]   frT,frB      Floating-Point Convert to Integer Word with Round to Zero
fdiv[.]     frT,frA,frB   Floating-Point Divide
fdivs[.]    frT,frA,frB   Floating-Point Divide Single-Precision
fmadd[.]    frT,frA,frC,frB  Floating-Point Multiply-Add
fmadds[.]   frT,frA,frC,frB  Floating-Point Multiply-Add Single-Precision
fmr[.]      frT,frB      Floating-Point Move Register
fmsub[.]    frT,frA,frC,frB  Floating-Point Multiply-Subtract
fmsubs[.]   frT,frA,frC,frB  Floating-Point Multiply-Subtract Single-Precision
fmul[.]     frT,frA,frC      Floating-Point Multiply

```

fmuls[.]	frT,frA,frC	Floating-Point Multiply Single-Precision
fnabs[.]	frT,frB	Floating-Point Negative Absolute Value
fneg[.]	frT,frB	Floating-Point Negate
fnmadd[.]	frT,frA,frC,frB	Floating-Point Negative Multiply-Add
fnmadds[.]	frT,frA,frC,frB	Floating-Point Negative Multiply-Add Single-Precision
fnmsub[.]	frT,frA,frC,frB	Floating-Point Negative Multiply-Subtract
fnmsubs[.]	frT,frA,frC,frB	Floating-Point Negative Multiply-Subtract Single-Precision
frsp[.]	frT,frB	Floating-Point Round to Single-Precision
fsub[.]	frT,frA,frB	Floating-Point Subtract
fsubs[.]	frT,frA,frB	Floating-Point Subtract Single-Precision
icbi	rA,rB	Instruction Cache Block Invalidate
inslwi[.]	rA,rS,nBits,start	Insert from Left Word Immediate
insrwi[.]	rA,rS,nBits,start	Insert from Right Word Immediate
isync		Instruction Cache Synchronize
la	rT,d(rA)	Load Address
la	rT,symbol	Load Address
lbz	rT,d(rA)	Load Byte and Zero
lbzu	rT,d(rA)	Load Byte and Zero with Update
lbzux	rT,rA,rB	Load Byte and Zero with Update Indexed
lbzx	rT,rA,rB	Load Byte and Zero Indexed
lfd	frT,d(rA)	Load Floating-Point Double-Precision
lfdi	frT,d(rA)	Load Floating-Point Double-Precision with Update
lfdiux	frT,rA,rB	Load Floating-Point Double-Precision with Update Indexed
lfdix	frT,rA,rB	Load Floating-Point Double-Precision Indexed
lfs	frT,d(rA)	Load Floating-Point Single-Precision
lfsi	frT,d(rA)	Load Floating-Point Single-Precision with Update
lfsiux	frT,rA,rB	Load Floating-Point Single-Precision with Update Indexed
lfsix	frT,rA,rB	Load Floating-Point Single-Precision Indexed
lha	rT,d(rA)	Load Halfword Algebraic
lhau	rT,d(rA)	Load Halfword Algebraic with Update
lhauix	rT,rA,rB	Load Halfword Algebraic with Update Indexed
lhax	rT,rA,rB	Load Halfword Algebraic Indexed
lhbrx	rT,rA,rB	Load Halfword Byte-Reversed Indexed
lhz	rT,d(rA)	Load Halfword and Zero
lhzu	rT,d(rA)	Load Halfword and Zero with Update
lhzuix	rT,rA,rB	Load Halfword and Zero with Update Indexed
lhzx	rT,rA,rB	Load Halfword and Zero Indexed
li	rT,s16	Load Immediate
lis	rT,s16	Load Immediate Shifted
lmw	rT,d(rA)	Load Multiple Word
lswi	rT,rA,nBytes	Load String Word Immediate
lswx	rT,rA,rB	Load String Word Indexed
lwarx	rT,rA,rB	Load Word and Reserve Indexed
lwbrx	rT,rA,rB	Load Word Byte-Reversed Indexed
lwz	rT,d(rA)	Load Word and Zero
lwzu	rT,d(rA)	Load Word and Zero with Update
lwzux	rT,rA,rB	Load Word and Zero with Update Indexed
lwzx	rT,rA,rB	Load Word and Zero Indexed
mcrf	crT,crS	Move Condition Register Fields
mcrfs	crT,crS	Move to Condition Register from FPSCR
mcrxr	crT	Move to Condition Register from XER
mfcrr	rT	Move from Condition Register
mfctr	rT	Move from Count Register
mfddar	rT	Move from Data Address Register

```

mfdbatl    rT,n    Move from Data Block-Address Translation Register n Lower
mfdbatu    rT,n    Move from Data Block-Address Translation Register n Upper
mfdsisr    rT      Move from Data Storage Interrupt Status Register
mfear      rT      Move from External Access Register
mffs[.]    frT     Move from FPSCR
mfibatl    rT,n    Move from Instruction Block-Address Translation Register n Lower
mfibatu    rT,n    Move from Instruction Block-Address Translation Register n Upper
mflr      rT      Move from Link Register
mfmsr     rT      Move from Machine State Register
mfpvr     rT      Move from Processor Version Register
mfsdr1    rT      Move from Storage Description Register 1
mfspr     rT,SPR   Move from Special Purpose Register
mfsprg    rT,n    Move from SPR G0-G3
mfsrr0    rT      Move from Save/Restore Register 0
mfssr1    rT      Move from Save/Restore Register 1
mfixer    rT      Move from Fixed-Point Exception Register
mr[.]     rA,rS    Move Register
mtrcf     crMask,rS Move to Condition Register Fields
mtctr     rS      Move to Count Register
mtdar     rS      Move to Data Address Register
mtdbatl   n,rS    Move to Data Block-Address Translation Register n Lower
mtdbatu   n,rS    Move to Data Block-Address Translation Register n Upper
mtdec     rS      Move to Decrement Register
mtdsisr   rS      Move to Data Storage Interrupt Status Register
mtear     rS      Move to External Access Register
mtfsb0[.] crbT    Move to FPSCR Bit 0
mtfsb1[.] crbT    Move to FPSCR Bit 1
mtfsf[.]  fpscrMask,frB Move to FPSCR Fields
mtfsfi[.] fpscrT,fieldVal Move to FPSCR Field Immediate
mtibatl   n,rS    Move to Instruction Block-Address Translation Register n Lower
mtibatu   n,rS    Move to Instruction Block-Address Translation Register n Upper
mtlr      rS      Move to Link Register
mtmsr     rS      Move to Machine State Register
mtdsr1    rS      Move to Storage Description Register 1
mtspr     SPR,rS   Move to Special Purpose Register
mtsprg    n,rS    Move to SPR G0-G3
mtsrr0    rS      Move to Save/Restore Register 0
mtsrr1    rS      Move to Save/Restore Register 1
mtxer     rS      Move to Fixed-Point Exception Register
mulhw[.]  rT,rA,rB   Multiply High Word
mulhwu[.] rT,rA,rB   Multiply High Word Unsigned
mulli     rT,rA,s16 Multiply Low Immediate
mullw[o][.] rT,rA,rB   Multiply Low Word
nand[.]   rA,rS,rB   NAND
neg[o][.] rT,rA    Negate
nop       no-op   No Operation
nor[.]    rA,rS,rB   NOR
not[.]    rA,rS    NOT
or[.]     rA,rS,rB   OR
orc[.]    rA,rS,rB   OR with Complement
ori       rA,rS,u16 OR Immediate
oris      rA,rS,u16 OR Immediate Shifted
rfi       Return from Interrupt
rlwimim[.] rA,rS,shift,mb,me Rotate Left Word Immediate then Mask Insert

```

```

rlnwim[.]    rA,rS,shift,mb,me    Rotate Left Word Immediate then AND with Mask
rlwnm[.]    rA,rS,rB,mb,me        Rotate Left Word then AND with Mask
rotlw[.]    rA,rS,rB        Rotate Left Word
rotlwi[.]    rA,rS,nBits    Rotate Left Word Immediate
rotrwi[.]    rA,rS,nBits    Rotate Right Word Immediate
sc          System Call
slw[.]      rA,rS,rB        Shift Left Word
slwi[.]      rA,rS,nBits    Shift Left Word Immediate
sraw[.]      rA,rS,rB        Shift Right Algebraic Word
srawi[.]     rA,rS,nBits    Shift Right Algebraic Word Immediate
srw[.]      rA,rS,rB        Shift Right Word
srwi[.]     rA,rS,nBits    Shift Right Word Immediate
stb         rS,d(rA)      Store Byte
stbu        rS,d(rA)      Store Byte with Update
stbux       rS,rA,rB      Store Byte with Update Indexed
stbx        rS,rA,rB      Store Byte Indexed
stfd        frS,d(rA)     Store Floating-Point Double
stfdu       frS,d(rA)     Store Floating-Point Double with Update
stfdux      frS,rA,rB     Store Floating-Point Double with Update Indexed
stfdx       frS,rA,rB     Store Floating-Point Double Indexed
stfs        frS,d(rA)     Store Floating-Point Single
stfsu       frS,d(rA)     Store Floating-Point Single with Update
stfsux      frS,rA,rB     Store Floating-Point Single with Update Indexed
stfsx       frS,rA,rB     Store Floating-Point Single Indexed
sth         rS,d(rA)      Store Halfword
sthbrx      rS,rA,rB      Store Halfword Byte-Reversed Indexed
sth         rS,d(rA)      Store Halfword with Update
sthux       rS,rA,rB     Store Halfword with Update Indexed
sthx        rS,rA,rB     Store Halfword Indexed
stmw        rS,d(rA)      Store Multiple Word
stswi       rS,rA,nBytes  Store String Word Immediate
stswx       rS,rA,rB     Store String Word Indexed
stw         rS,d(rA)      Store Word
stwbrx      rS,rA,rB     Store Word Byte-Reversed Indexed
stwcx.      rS,rA,rB     Store Word Conditional Indexed
stwu        rS,d(rA)      Store Word with Update
stwux       rS,rA,rB     Store Word with Update Indexed
stwx        rS,rA,rB     Store Word Indexed
sub[o][.]   rT,rA,rB        Subtract
subc[o][.]   rT,rA,rB        Subtract Carrying
subf[o][.]   rT,rA,rB        Subtract From
subfc[o][.]  rT,rA,rB        Subtract From Carrying
subfe[o][.]  rT,rA,rB        Subtract From Extended
subfic       rT,rA,s16     Subtract From Immediate Carrying
subfme[o][.] rT,rA        Subtract From Minus One Extended
subfze[o][.] rT,rA        Subtract From Zero Extended
subi         rT,rA,s16     Subtract Immediate
subic[.]     rT,rA,s16     Subtract Immediate Carrying
subis        rT,rA,s16     Subtract Immediate Shifted
sync         Synchronize
trap         Trap Unconditionally
tw          trapOn,rA,rB    Trap Word
tweq        rA,rB        Trap Word if Equal
tweqi       rA,s16       Trap Word if Equal Immediate

```

twge	rA,rB	Trap Word if Greater Than or Equal
twgei	rA,s16	Trap Word if Greater Than or Equal Immediate
twgt	rA,rB	Trap Word if Greater Than
twgti	rA,s16	Trap Word if Greater Than Immediate
twi	trapOn,rA,s16	Trap Word Immediate
twle	rA,rB	Trap Word if Less Than or Equal
twlei	rA,s16	Trap Word if Less Than or Equal Immediate
twlge	rA,rB	Trap Word if Logically Greater Than or Equal
twlgei	rA,s16	Trap Word if Logically Greater Than or Equal Immediate
twlgt	rA,rB	Trap Word if Logically Greater Than
twlgti	rA,s16	Trap Word if Logically Greater Than Immediate
twlle	rA,rB	Trap Word if Logically Less Than or Equal
twllei	rA,s16	Trap Word if Logically Less Than or Equal Immediate
twllt	rA,rB	Trap Word if Logically Less Than
twllti	rA,s16	Trap Word if Logically Less Than Immediate
twlng	rA,rB	Trap Word if Logically Not Greater Than
twlngi	rA,s16	Trap Word if Logically Not Greater Than Immediate
twlnl	rA,rB	Trap Word if Logically Not Less Than
twlnli	rA,s16	Trap Word if Logically Not Less Than Immediate
twlt	rA,rB	Trap Word if Less Than
twlti	rA,s16	Trap Word if Less Than Immediate
twne	rA,rB	Trap Word if Not Equal
twnei	rA,s16	Trap Word if Not Equal Immediate
twng	rA,rB	Trap Word if Not Greater Than
twngi	rA,s16	Trap Word if Not Greater Than Immediate
twnl	rA,rB	Trap Word if Not Less Than
twnli	rA,s16	Trap Word if Not Less Than Immediate
xor[.]	rA,rS,rB	Exclusive OR
xori	rA,rS,u16	Exclusive OR Immediate
xoris	rA,rS,u16	Exclusive OR Immediate Shifted

PPC Extended Mnemonics

PPC Asm supports the extended mnemonics described in the IBM PowerPC User Instruction Set Architecture. This appendix lists the extended mnemonics that are supported in the following categories:

simplified branches

branches that incorporate conditions

access to and from special-purpose registers

traps

other extended mnemonics

For more information on extended mnemonics, see the IBM PowerPC User Instruction Set Architecture or the appropriate PowerPC processor document (such as the Motorola PowerPC 601 RISC Microprocessor User's Manual).

Simplified Branches

This section lists the branch extended mnemonics that already specify a branch condition. If you know the likely outcome of a branch condition, you can add a suffix to a branch extended mnemonic to set a bit for predicting whether a branch will be taken. When you add a plus sign (+) as a suffix, the branch is predicted to be taken. When you add a minus sign (-) as a suffix, the branch is predicted not to be taken.

Extended mnemonic
Operation
Base mnemonic equivalent

Bctr
Branch Unconditionally to CTR
bcctr 20,0

bctrl
Branch Unconditionally to CTR, set LR
bcctrl 20,0

bdnz addr
Decrement CTR, Branch if CTR Non-zero to Relative addr
bc 16,0,addr

bdnza addr
Decrement CTR, Branch if CTR Non-zero to Absolute addr
bca 16,0,addr

bdnzf BI,addr
Decrement CTR, Branch if CTR Non-zero and Condition False to Relative addr
bc 0,BI,addr

bdnzfa BI,addr
Decrement CTR, Branch if CTR Non-zero and Condition False to Absolute addr
bca 0,BI,addr

bdnzfl BI,addr
Decrement CTR, Branch if CTR Non-zero and Condition False to Relative addr, set LR
bcl 0,BI,addr

bdnzfla BI,addr
Decrement CTR, Branch if CTR Non-zero and Condition False to Absolute addr, set LR
bcla 0,BI,addr

bdnzflr BI
Decrement CTR, Branch if CTR Non-zero and Condition False to LR
bclrl 2,BI

bdnzflrl BI
Decrement CTR, Branch if CTR Non-zero and Condition False to LR, set LR
bclrl 0,BI

bdnzl addr
Decrement CTR, Branch if CTR Non-zero to Relative addr, set LR
bcl 16,0,addr

bdnzla addr

Decrement CTR, Branch if CTR Non-zero to Absolute addr, set LR

bcla 16,0,addr

bdnzlr

Decrement CTR, Branch if CTR Non-zero to LR

bclr 16,0

bdnzlrl

Decrement CTR, Branch if CTR Non-zero to LR, set LR

bclrl 16,0

bdnzt BI,addr

Decrement CTR, Branch if CTR Non-zero and Condition True to Relative addr

bc 8,BI,addr

bdnzta BI,addr

Decrement CTR, Branch if CTR Non-zero and Condition True to Absolute addr

bca 8,BI,addr

bdnztl BI,addr

Decrement CTR, Branch if CTR Non-zero and Condition True to Relative addr, set LR

bcl 8,BI,addr

bdnztla BI,addr

Decrement CTR, Branch if CTR Non-zero and Condition True to Absolute addr, set LR

bcla 8,BI,addr

bdnztlr BI

Decrement CTR, Branch if CTR Non-zero and Condition True to LR

bclr 8,BI

bdnztlrl BI

Decrement CTR, Branch if CTR Non-zero and Condition True to LR, set LR

bclrl 8,BI

bdz addr

Decrement CTR, Branch if CTR Zero to Relative addr bc

18,0,addr

bdza addr

Decrement CTR, Branch if CTR Zero to Absolute addr

bca 18,0,addr

bdzf BI,addr

Decrement CTR, Branch if CTR Zero and Condition False to Relative addr

bc 2,BI,addr

bdzfa BI,addr

Decrement CTR, Branch if CTR Zero and Condition False to Absolute addr

bca 2,BI,addr

bdzfl BI,addr

Decrement CTR, Branch if CTR Zero and Condition False to Relative addr, set LR
 bcl 2,BI,addr

bdzfla BI,addr
 Decrement CTR, Branch if CTR Zero and Condition False to Absolute addr, set LR
 bcla 2,BI,addr

bdzflr BI
 Decrement CTR, Branch if CTR Zero and Condition False to LR
 bclr 2,BI

bdzflrl BI
 Decrement CTR, Branch if CTR Zero and Condition False to LR, set LR
 bclrl 2,BI

bdzl addr
 Decrement CTR, Branch if CTR Zero to Relative addr, set LR
 bcl 18,0,addr

bdzla addr
 Decrement CTR, Branch if CTR Zero to Absolute addr, set LR
 bcla 18,0,addr

bdzlr
 Decrement CTR, Branch if CTR Zero to LR
 bclr 18,0

bdzlrll
 Decrement CTR, Branch if CTR Zero to LR, set LR
 bclrl 18,0

bdzt BI,addr
 Decrement CTR, Branch if CTR Zero and Condition True to Relative addr
 bc 10,BI,addr

bdzta BI,addr
 Decrement CTR, Branch if CTR Zero and Condition True to Absolute addr
 bca 10,BI,addr

bdztl BI,addr
 Decrement CTR, Branch if CTR Zero and Condition True to Relative addr, set LR
 bcl 10,BI,addr

bdztla BI,addr
 Decrement CTR, Branch if CTR Zero and Condition True to Absolute addr, set LR
 bcla 10,BI,addr

bdztlr BI
 Decrement CTR, Branch if CTR Zero and Condition True to LR
 bclr 10,BI

bdztlrl BI
 Decrement CTR, Branch if CTR Zero and Condition True to LR, set LR
 bclrl 10,BI

bf BI,addr
Branch if Condition False to Relative addr
bc 4,BI,addr

bfa BI,addr
Branch if Condition False to Absolute addr
bca 4,BI,addr

bfctr BI
Branch if Condition False to CTR
bcctr 4,BI

bfctrl BI
Branch if Condition False to CTR, set LR
bcctrl 4,BI

bfl BI,addr
Branch if Condition False to Relative addr, set LR
bcl 4,BI,addr

bfla BI,addr
Branch if Condition False to Absolute addr, set LR
bcla 4,BI,addr

bflr BI
Branch if Condition False to LR
bclr 4,BI

bflrl BI
Branch if Condition False to LR, set LR
bclrl 4,BI

blr
Branch Unconditionally to LR
bclr 20,0

blrl
Branch Unconditionally to LR, set LR
bclrl 20,0

bt BI,addr
Branch if Condition True to Relative addr
bc 12,BI,addr

bta BI,addr
Branch if Condition True to Absolute addr
bca 12,BI,addr

btctr BI
Branch if Condition True to CTR
bcctr 12,BI

btctrl BI

Branch if Condition True to CTR, set LR
bcctrl 12,BI

btl BI,addr
Branch if Condition True to Relative addr, set LR
bcl 12,BI,addr

btla BI,addr
Branch if Condition True to Absolute addr, set LR
bcla 12,BI,addr

btlr BI
Branch if Condition True to LR
bclr 12,BI

btlrl BI
Branch if Condition True to LR, set LR
bclrl 12,BI

Branches That Incorporate Conditions

This section lists the branch extended mnemonics that let you specify a branch condition. If you know the likely outcome of a branch condition, you can add a suffix to a branch extended mnemonic to set a bit for predicting whether a branch will be taken. When you add a plus sign (+) as a suffix, the branch is predicted to be taken. When you add a minus sign (-) as a suffix, the branch is predicted not to be taken.

Extended mnemonic
Operation
Base mnemonic equivalent

beq [CRn,]addr
Branch if Equal to Relative addr
bc 12,[CRn+]2,addr

beqa [CRn,]addr
Branch if Equal to Absolute addr
bca 12,[CRn+]2,addr

beqctr [CRn]
Branch if Equal to CTR
bctr 12,[CRn+]2

beqctrl [CRn]
Branch if Equal to CTR, set LR
bcctrl 12,[CRn+]2

beql [CRn,]addr
Branch if Equal to Relative addr, set LR
bcl 12,[CRn+]2,addr

beqla [CRn,]addr
Branch if Equal to Absolute addr, set LR

bcla 12,[CRn+]2,addr

beqlr [CRn]
Branch if Equal to LR
bclr 12,[CRn+]2

beqlrl [CRn]
Branch if Equal to LR, set LR
bclrl 12,[CRn+]2

bge [CRn,]addr
Branch if Greater Than or Equal to Relativeaddr
bc 4,[CRn+]0,addr

bgea [CRn,]addr
Branch if Greater Than or Equal to Absolute addr
bca 4,[CRn+]0,addr

bgectr [CRn]
Branch if Greater Than or Equal to CTR
bctr 4,[CRn+]0

bgectrl [CRn]
Branch if Greater Than or Equal to CTR, set LR
bctrl 4,[CRn+]0

bgel [CRn,]addr
Branch if Greater Than or Equal to Relative addr, set LR
bcl 4,[CRn+]0,addr

bgela [CRn,]addr
Branch if Greater Than or Equal to Absolute addr, set LR
bcla 4,[CRn+]0,addr

bgelr [CRn]
Branch if Greater Than or Equal to LR
bclr 4,[CRn+]0

bgelrl [CRn]
Branch if Greater Than or Equal to LR, set LR
bclrl 4,[CRn+]0

bgt [CRn,]addr
Branch if Greater Than to Relative addr
bc 12,[CRn+]1,addr

bgt a [CRn,]addr
Branch if Greater Than to Absolute Addr
bca 12,[CRn+]1,addr

bgtctr [CRn]
Branch if Greater Than to CTR
bctr 12,[CRn+]1

bgtctrl [CRn]

Branch if Greater Than to CTR, set LR

bctrl 12,[CRn+]1

bgtl [CRn,]addr

Branch if Greater Than to Relative addr, set LR

bcl 12,[CRn+]1,addr

bgtla [CRn,]addr

Branch if Greater Than to Absolute addr, setLR

bcla 12,[CRn+]1,addr

bgtlr [CRn]

Branch if Greater Than to LR

bclr 12,[CRn+]1

bgtlrl [CRn]

Branch if Greater Than to LR, set LR

bclrl 12,[CRn+]1

ble [CRn,]addr

Branch if Less Than or Equal to Relative addr

bc 4,[CRn+]1,addr

blea [CRn,]addr

Branch if Less Than or Equal to Absolute addr

bca 4,[CRn+]1,addr

blectr [CRn]

Branch if Less Than or Equal to CTR

bctr 4,[CRn+]1

blectrl [CRn]

Branch if Less Than or Equal to CTR, set LR

bcctrl 4,[CRn+]1

blel [CRn,]addr

Branch if Less Than or Equal to Relative addr, set LR

bcl 4,[CRn+]1,addr

blela [CRn,]addr

Branch if Less Than or Equal to Absolute addr, set LR

bcla 4,[CRn+]1,addr

blelr [CRn]

Branch if Less Than or Equal to LR

bclr 4,[CRn+]1

blelrl [CRn]

Branch if Less Than or Equal to LR, set LR

bclrl 4,[CRn+]1

blt [CRn,]addr

Branch if Less Than to Relative addr

bc 12,[CRn+]0,addr

blta [CRn,]addr

Branch if Less Than to Absolute addr

bca 12,[CRn+]0,addr

bltctr [CRn]

Branch if Less Than to CTR

bctr 12,[CRn+]0

bltctrl [CRn]

Branch if Less Than to CTR, set LR

bctrl 12,[CRn+]0

bltl [CRn,]addr

Branch if Less Than to Relative addr, set LR

bcl 12,[CRn+]0,addr

bltla [CRn,]addr

Branch if Less Than to Absolute addr, set LR

bcla 12,[CRn+]0,addr

bltlr [CRn]

Branch if Less Than to LR

bclr 12,[CRn+]0

bltlrl [CRn]

Branch if Less Than to LR, set LR

bclrl 12,[CRn+]0

bne [CRn,]addr

Branch if Not Equal to Relative addr

bc 4,[CRn+]2,addr

bnea [CRn,]addr

Branch if Not Equal to Absolute addr

bca 4,[CRn+]2,addr

bnctr [CRn]

Branch if Not Equal to CTR

bctr 4,[CRn+]2

bnctrl [CRn]

Branch if Not Equal to CTR, set LR

bctrl 4,[CRn+]2

bnel [CRn,]addr

Branch if Not Equal to Relative addr, set LR

bcl 4,[CRn+]2,addr

bnela [CRn,]addr

Branch if Not Equal to Absolute addr, set LR

bcla 4,[CRn+]2,addr

bnelr [CRn]
Branch if Not Equal to LR
bclr 4,[CRn+2]

bnelrl [CRn]
Branch if Not Equal to LR, set LR
bclrl 4,[CRn+2]

bng [CRn,]addr
Branch if Not Greater Than to Relative addr
bc 4,[CRn+1],addr

bnga [CRn,]addr
Branch if Not Greater Than to Absolute addr
bca 4,[CRn+1],addr

bngctr [CRn]
Branch if Not Greater Than to CTR
bctr 4,[CRn+1]

bngctrl [CRn]
Branch if Not Greater Than to CTR, set LR
bcctrl 4,[CRn+1]

bngl [CRn,]addr
Branch if Not Greater Than to Relative addr, set LR
bcl 4,[CRn+1],addr

bngla [CRn,]addr
Branch if Not Greater Than to Absolute addr, set LR
bcla 4,[CRn+1],addr

bnglr [CRn]
Branch if Not Greater Than to LR
bclr 4,[CRn+1]

bnglrl [CRn]
Branch if Not Greater Than to LR, set LR
bclrl 4,[CRn+1]

bnl [CRn,]addr
Branch if Not Less Than to Relative addr
bc 4,[CRn+0],addr

bnla [CRn,]addr
Branch if Not Less Than to Absolute addr
bca 4,[CRn+0],addr

bnlctr [CRn]
Branch if Not Less Than to CTR
bctr 4,[CRn+0]

bnlctrl [CRn]
Branch if Not Less Than to CTR, set LR

bctrl 4,[CRn+]0

bnll [CRn,]addr

Branch if Not Less Than to Relative addr, set LR

bcl 4,[CRn+]0,addr

bnlla [CRn,]addr

Branch if Not Less Than to Absolute addr, set LR

bcla 4,[CRn+]0,addr

bnllrl [CRn]

Branch if Not Less Than to LR, set LR

bclrl 4,[CRn+]0

bns [CRn,]addr

Branch if Not Summary overflow to Relative addr

bc 4,[CRn+]3,addr

bnsa [CRn,]addr

Branch if Not Summary overflow to Absolute addr

bca 4,[CRn+]3,addr

bnsctr [CRn]

Branch if Not Summary overflow to CTR

bctr 4,[CRn+]3

bnsctrl [CRn]

Branch if Not Summary overflow to CTR, set LR

bctrl 4,[CRn+]3

bnsll [CRn,]addr

Branch if Not Summary overflow to Relative addr, set LR

bcl 4,[CRn+]3,addr

bnslla [CRn,]addr

Branch if Not Summary overflow to Absolute addr, set LR

bcla 4,[CRn+]3,addr

bnsllr [CRn]

Branch if Not Summary overflow to LR

bcll 4,[CRn+]3

bnsllrl [CRn]

Branch if Not Summary overflow to LR, set LR

bclll 4,[CRn+]3

bnu [CRn,]addr

Branch if Not Unordered to Relative addr

bc 4,[CRn+]3,addr

bnu [CRn,]addr

Branch if Not Unordered to Absolute addr

bca 4,[CRn+]3,addr

bnuctr [CRn]
Branch if Not Unordered to CTR
bctr 4,[CRn+]³

bnuctrl [CRn]
Branch if Not Unordered to CTR, set LR
bctr1 4,[CRn+]³

bnul [CRn,]addr
Branch if Not Unordered to Relative addr, set LR
bcl 4,[CRn+]³,addr

bnula [CRn,]addr
Branch if Not Unordered to Absolute addr, setLR
bcla 4,[CRn+]³,addr

bnulr [CRn]
Branch if Not Unordered to LR
bclr 4,[CRn+]³

bnulrl [CRn]
Branch if Not Unordered to LR, set LR
bclr1 4,[CRn+]³

bso [CRn,]addr
Branch if Summary Overflow to Relative addr
bc 12,[CRn+]³,addr

bsoa [CRn,]addr
Branch if Summary Overflow to Absolute addr
bca 12,[CRn+]³,addr

bsoctr [CRn]
Branch if Summary Overflow to CTR
bctr 12,[CRn+]³

bsoctrl [CRn]
Branch if Summary Overflow to CTR, set LR
bctr1 12,[CRn+]³

bsol [CRn,]addr
Branch if Summary Overflow to Relative addr, set LR
bcl 12,[CRn+]³,addr

bsola [CRn,]addr
Branch if Summary Overflow to Absoluteaddr, set LR
bcla 12,[CRn+]³,addr

bsolr [CRn]
Branch if Summary Overflow to LR
bclr 12,[CRn+]³

bsolrl [CRn]
Branch if Summary Overflow to LR, set LR

bclr 12,[CRn+]3

bun [CRn,]addr
Branch if Unordered to Relative addr
bc 12,[CRn+]3,addr

buna [CRn,]addr
Branch if Unordered to Absolute addr
bca 12,[CRn+]3,addr

bunctr [CRn]
Branch if Unordered to CTR
bctr 12,[CRn+]3

bunctrl [CRn]
Branch if Unordered to CTR, set LR
bctrl 12,[CRn+]3

bunl [CRn,]addr
Branch if Unordered to Relative addr, set LR
bcl 12,[CRn+]3,addr

bunla [CRn,]addr
Branch if Unordered to Absolute addr, set LR
bcla 12,[CRn+]3,addr

bunlr [CRn]
Branch if Unordered to LR
bclr 12,[CRn+]3

bunlrl [CRn]
Branch if Unordered to LR, set LR
bclrl 12,[CRn+]3

Move From/To a Special-Purpose Register

This section describes the extended mnemonics that access a special-purpose register. When accessing a special-purpose register, note that

the Time Base registers (TB) are PowerPC registers that are not implemented on the 601 processor

the PowerPC mnemonic mttb is not implemented on the 601 processor

registers MQ, RTCL, and RTCU are not part of the PowerPC architecture (they are included in the 601 processor for POWER compatibility)

access to registers RTCL and RTCU is read-only when the 601 processor is in user mode

register DEC is accessible only in supervisor mode on PowerPC processors (read-only in user mode on the 601 processor)

the DBAT registers will be available only when PPCAsm supports 64-bit instructions

Extended mnemonic
Operation Base
mnemonic equivalent

mfasr Rx
Move From Address Space Register
mfspr Rx,280

mfctr Rx
Move From Count Register (CTR)
mfspr Rx,9

mfdar Rx
Move From Data Address Register (DAR)
mfspr Rx,19

mfdbatl Rx,n
Move From DBAT Lower Registers DBAT0L through DBAT3L
mfspr Rx,537+2*n

mfdbatu Rx,n
Move From DBAT Upper Registers DBAT0U through DBAT3U
mfspr Rx,536+2*n

mfdec Rx
Move From Decrementer Register (DEC)
mfspr Rx,6

mfdsisr Rx
Move From Data Storage Interrupt Status Register (DSISR)
mfspr Rx,18

mfear Rx
Move From External Access Register (EAR)
mfspr Rx,282

mfibatl Rx,n
Move From IBAT Lower Registers IBAT0L through IBAT3L
mfspr Rx,529+2*n

mfibatu Rx,n
Move From IBAT Upper Registers IBAT0U through IBAT3U
mfspr Rx,528+2*n

mflr Rx
Move From Link Register (LR)
mfspr Rx,8

mfmq Rx
Move From MQ Register
mfspr Rx,0

mfpvr Rx

Move From Processor Version Register

mfspr Rx,287

mfrtcl Rx

Move From Real Time Clock Lower Register (RTCL)

mfspr Rx,5

mfrtcu Rx

Move From Real Time Clock Upper Register (RTCU)

mfspr Rx,4

mfsdr1 Rx

Move From Storage Description Register 1 (SDR1)

mfspr Rx,25

mfsprg Rx,n

Move From General Special Purpose Registers SPRG0 through SPRG3

mfspr Rx,272+n

mfsrr0 Rx

Move From Save/Restore Register 0 (SRR0)

mfspr Rx,26

mfsrr1 Rx

Move From Save/Restore Register 1 (SRR1)

mfspr Rx,27

mftb Rx

Move From Time Base Lower Register

mftb Rx,268

mftbu Rx

Move From Time Base Upper Register

mftb Rx,269

mfxer Rx

Move From Fixed Point Exception Register (XER)

mfspr Rx,1

mtasr Rx

Move To Address Space Register

mtspr 280,Rx

mtctr Rx

Move To Count Register (CTR)

mtspr 9,Rx

mtdar Rx

Move To Data Address Register (DAR)

mtspr 19,Rx

mtdbatl n,Rx

Move To DBAT Lower Registers DBAT0L through DBAT3L

mtspr 537+2*n,Rx

mtdbatu n,Rx

Move To DBAT Upper Registers DBAT0U through DBAT3U

mtspr 536+2*n,Rx

mtdec Rx

Move To Decrementer Register (DEC)

mtspr 22,Rx

mtdsisr Rx

Move To Data Storage Interrupt Status Register (DSISR)

mtspr 18,Rx

mtear Rx

Move To External Access Register (EAR)

mtspr 282,Rx

mtibatl n,Rx

Move To IBAT Lower Registers IBAT0L through IBAT3L

mtspr 529+2*n,Rx

mtibatu n,Rx

Move To IBAT Upper Registers IBAT0U through IBAT3U

mtspr 528+2*n,Rx

mtlr Rx

Move To Link Register (LR)

mtspr 8,Rx

mtmq Rx

Move To MQ Register

mtspr 0,Rx

mtrtcl Rx

Move To Real Time Clock Lower Register (RTCL)

mtspr 21,Rx

mtrtcu Rx

Move To Real Time Clock Upper Register (RTCU)

mtspr 20,Rx

mtsdr1 Rx

Move To Storage Description Register 1 (SDR1)

mtspr 25,Rx

mtsprg n,Rx

Move To General Special Purpose Registers SPRG0 through SPRG3

mtspr 272+n,Rx

mtsrr0 Rx

Move To Save/Restore Register 0 (SRR0)

mtspr 26,Rx

mtsrr1 Rx
Move To Save/Restore Register 1 (SRR1)
mtspr 27,Rx

mttbl Rx
Move To Time Base Lower Register
mtspr 284,Rx

mttbu Rx
Move To Time Base Upper Register
mtspr 285,Rx

mtxer Rx
Move To Fixed Point Exception Register (XER)
mtspr 1,Rx

Traps

This section describes the trap extended mnemonics. Although 64-bit double comparison traps are listed, they will be available only when PPCAsm supports 64-bit instructions.

Extended mnemonic
Operation
Base mnemonic equivalent

trap
Trap Unconditionally
tw 31,0,0

tdeq Rx,Ry
Trap if Double is Equal
td 4,Rx,Ry

tdeqi Rx,SI
Trap if Double is Equal Immediate
tdi 4,Rx,SI

tdge Rx,Ry
Trap if Double is Greater Than or Equal
td 12,Rx,Ry

tdgei Rx,SI
Trap if Double is Greater Than or Equal Immediate
tdi 12,Rx,SI

tdgt Rx,Ry
Trap if Double is Greater Than
td 8,Rx,Ry

tdgti Rx,SI
Trap if Double is Greater Than Immediate
tdi 8,Rx,SI

tdle Rx,Ry
Trap if Double is Less Than or Equal
td 20,Rx,Ry

tdlei Rx,SI
Trap if Double is Less Than or Equal Immediate
tdi 20,Rx,SI

tdlge Rx,Ry
Trap if Double is Logically Greater Than or Equal
td 5,Rx,Ry

tdlgei Rx,SI
Trap if Double is Logically Greater Than or Equal Immediate
tdi 5,Rx,SI

tdlgt Rx,Ry
Trap if Double is Logically Greater Than
td 1,Rx,Ry

tdlgti Rx,SI
Trap if Double is Logically Greater Than Immediate
tdi 1,Rx,SI

tdlle Rx,Ry
Trap if Double is Logically Less Than or Equal
td 6,Rx,Ry

tdllei Rx,SI
Trap if Double is Logically Less Than or Equal Immediate
tdi 6,Rx,SI

tdllt Rx,Ry
Trap if Double is Logically Less Than
td 2,Rx,Ry

tdllti Rx,SI
Trap if Double is Logically Less Than Immediate
tdi 2,Rx,SI

tdlng Rx,Ry
Trap if Double is Logically Not Greater Than
td 6,Rx,Ry

tdlngi Rx,SI
Trap if Double is Logically Not Greater Than Immediate
tdi 6,Rx,SI

tdlnl Rx,Ry
Trap if Double is Logically Not Less Than
td 5,Rx,Ry

tdlnli Rx,SI

Trap if Double is Logically Not Less Than Immediate
tdi 5,Rx,SI

tdlt Rx,Ry
Trap if Double is Less Than
td 16,Rx,Ry

tdlti Rx,SI
Trap if Double is Less Than Immediate
tdi 16,Rx,SI

tdne Rx,Ry
Trap if Double is Not Equal
td 24,Rx,Ry

tdnei Rx,SI
Trap if Double is Not Equal Immediate
tdi 24,Rx,SI

tdng Rx,Ry
Trap if Double is Not Greater Than
td 20,Rx,Ry

tdngi Rx,SI
Trap if Double is Not Greater Than Immediate
tdi 20,Rx,SI

tdnl Rx,Ry
Trap if Double is Not Less Than
td 12,Rx,Ry

tdnli Rx,SI
Trap if Double is Not Less Than Immediate
tdi 12,Rx,SI

tweq Rx,Ry
Trap if Word is Equal
tw 4,Rx,Ry

tweqi Rx,SI
Trap if Word is Equal Immediate
twi 4,Rx,SI

twge Rx,Ry
Trap if Word is Greater Than or Equal
tw 12,Rx,Ry

twgei Rx,SI
Trap if Word is Greater Than or Equal Immediate
twi 12,Rx,SI

twgt Rx,Ry
Trap if Word is Greater Than
tw 8,Rx,Ry

twgti Rx,SI
 Trap if Word is Greater Than Immediate
 twi 8,Rx,SI

twle Rx,Ry
 Trap if Word is Less Than or Equal
 tw 20,Rx,Ry

twlei Rx,SI
 Trap if Word is Less Than or Equal Immediate
 twi 20,Rx,SI

twlge Rx,Ry
 Trap if Word is Logically Greater Than or Equal
 tw 5,Rx,Ry

twlgei Rx,SI
 Trap if Word is Logically Greater Than or Equal Immediate
 twi 5,Rx,SI

twlt Rx,Ry
 Trap if Word is Logically Greater Than
 tw 1,Rx,Ry

twlgti Rx,SI
 Trap if Word is Logically Greater Than Immediate
 twi 1,Rx,SI

twlle Rx,Ry
 Trap if Word is Logically Less Than or Equal
 tw 6,Rx,Ry

twllei Rx,SI
 Trap if Word is Logically Less Than or Equal Immediate
 twi 6,Rx,SI

twllt Rx,Ry
 Trap if Word is Logically Less Than
 tw 2,Rx,Ry

twllti Rx,SI
 Trap if Word is Logically Less Than Immediate
 twi 2,Rx,SI

twlng Rx,Ry
 Trap if Word is Logically Not Greater Than
 tw 6,Rx,Ry

twlngi Rx,SI
 Trap if Word is Logically Not Greater Than Immediate
 twi 6,Rx,SI

twlnl Rx,Ry

Trap if Word is Logically Not Less Than
tw 5,Rx,Ry

twlnli Rx,SI
Trap if Word is Logically Not Less Than Immediate
twi 5,Rx,SI

twlt Rx,Ry
Trap if Word is Less Than
tw 16,Rx,Ry

twlti Rx,SI
Trap if Word is Less Than Immediate
twi 16,Rx,SI

twne Rx,Ry
Trap if Word is Not Equal
tw 24,Rx,Ry

twnei Rx,SI
Trap if Word is Not Equal Immediate
twi 24,Rx,SI

twng Rx,Ry
Trap if Word is Not Greater Than
tw 20,Rx,Ry

twngi Rx,SI
Trap if Word is Not Greater Than Immediate
twi 20,Rx,SI

twnl Rx,Ry
Trap if Word is Not Less Than
tw 12,Rx,Ry

twnli Rx,SI
Trap if Word is Not Less Than Immediate
twi 12,Rx,SI

Other Extended Mnemonics

This section describes the other extended mnemonics that PPCAsm supports. Although double-word-comparison extended mnemonics are listed, they will be available only when PPCAsm supports 64-bit instructions.

Extended mnemonic
Operation
Base mnemonic equivalent

clrldi Rx,Ry,n
Clear Left Double Immediate
rldicl Rx,Ry,0,n

clrslldi Rx,Ry,b,n
Clear Left and Shift Left Double Immediate
rldic Rx,Ry,n,b-n

clrlwi Rx,Ry,n
Clear Left Word Immediate
rlwicl Rx,Ry,0,n

clrslwi Rx,Ry,b,n
Clear Left and Shift Left Word Immediate
rlwic Rx,Ry,n,b-n

clrrdi Rx,Ry,n
Clear Right Double Immediate
rldicr Rx,Ry,0,63-n

clrrwi Rx,Ry,n
Clear Right Word Immediate
rlwicr Rx,Ry,0,63-n

cmpd crfD,Rx,Ry
Compare Doubleword
cmp crfD,1,Rx,Ry

cmpdi crfD,Rx,SI
Compare Doubleword Immediate
cmpi crfD,1,Rx,SI

cmpld crfD,Rx,Ry
Compare Logical Doubleword
cmpl crfD,1,Rx,Ry

cmpldi crfD,Rx,UI
Compare Logical Doubleword Immediate
cmpli crfD,1,Rx,UI

cmpw crfD,Rx,Ry
Compare Word
cmp crfD,0,Rx,Ry

cmpwi crfD,Rx,SI
Compare Word Immediate
cmpi crfD,0,Rx,SI

cmplw crfD,Rx,Ry
Compare Logical Word
cmpl crfD,0,Rx,Ry

cmplwi crfD,Rx,UI
Compare Logical Word Immediate
cmpli crfD,0,Rx,UI

crclr Bx
Condition Register Clear

crxor Bx,Bx,Bx

crmmove Bx,By
Condition Register Move
cror Bx,By,By

crnot Bx,By
Condition Register Not
crnor Bx,By,By

crset Bx
Condition Register Set
creqv Bx,Bx,Bx

extldi Rx,Ry,n,b
Extract and Left Justify Double Immediate
rldicr Rx,Ry,b,n-1

extrdi Rx,Ry,n,b
Extract and Right Justify Double Immediate
rldicl Rx,Ry,b+n,64-n

extlwi Rx,Ry,n,b
Extract and Left Justify Word Immediate
rlwicr Rx,Ry,b,n-1

extrwi Rx,Ry,n,b
Extract and Right Justify Word Immediate
rlwicl Rx,Ry,b+n,64-n

insrdi Rx,Ry,n,b
Insert from Right Double Immediate
rldimi Rx,Ry,64-(b+n),b

insrwi Rx,Ry,n,b
Insert from Right Word Immediate
rlwimi Rx,Ry,64-(b+n),b

la Rx,D(Ry)
Load Address
addi Rx,Ry,D

la Rx,v
Load Address
addi Rx,Rv,Dv

li Rx,SI
Load 16-bit Signed Immediate
addi Rx,0,SI

lis Rx,SI
Load 16-bit Signed Immediate, Shifted
addis Rx,0,SI

mr Rx,Ry
Move Register
or Rx,Ry,Ry

nop
No Operation
ori 0,0,0

not Rx,Ry
Complement Register (Logical Not)
nor Rx,Ry,Ry

rotldi Rx,Ry,n
Rotate Left Double Immediate
rldicl Rx,Ry,n,0

rotrdi Rx,Ry,n
Rotate Right Double Immediate
rldicl Rx,Ry,64-n,0

rotld Rx,Ry,Rz
Rotate Left Double
rldcl Rx,Ry,Rz,0

rotlwi Rx,Ry,n
Rotate Left Word Immediate
rlwicl Rx,Ry,n,0

rotrwi Rx,Ry,n
Rotate Right Word Immediate
rlwicl Rx,Ry,64-n,0

rotlw Rx,Ry,Rz
Rotate Left Word
rlwcl Rx,Ry,Rz,0

sldi Rx,Ry,n
Shift Left Double Immediate
rldicr Rx,Ry,n,63-n

slwi Rx,Ry,n
Shift Left Word Immediate
rlwicr Rx,Ry,n,63-n

srdi Rx,Ry,n
Shift Right Double Immediate
rldicl Rx,Ry,64-n,n

srwi Rx,Ry,n
Shift Right Word Immediate
rlwicl Rx,Ry,64-n,n

sub Rx,Ry,Rz
Subtract

subf Rx,Rz,Ry

subc Rx,Ry,Rz
Subtract Carrying
subfc Rx,Rz,Ry

subi Rx,Ry,val
Subtract Immediate
addi Rx,Ry,-val

subic Rx,Ry,val
Subtract Immediate Carrying
addic Rx,Ry,-val

subic. Rx,Ry,val
Subtract Immediate Carrying and Record
addic. Rx,Ry,-val

subis Rx,Ry,val
Subtract Immediate Shifted
addis Rx,Ry,-val

Mac Resources Types List

This list is composed of the most commonly used resource types. There are many more resource types not on this list. All resource types are 4 characters long. When only 3 are used it is always followed by a space.

N = No ResEdit editor or template. Viewed by Hex Editor.

T = ResEdit has a template.

E = ResEdit has an editor.

* = Third party ResEdit editor or template available.

C = Apple's ResEdit Code Editor Extension.

Name = Description

actb = Alert color look-up table. E,T.

acur = Animated cursor resource. T.

ADBS = Apple Desktop bus driver code. C.

aedt = AppleEvents. *.

alis = alias information.*.

ALRT = Location and size of alert window. E,T.

APPL = Application list from Desktop file. T.

bmap = Bitmap used by old versions of Control Panels. N.

boot = Boot blocks in system file. C.

BNDL = Bundle- used to attach icons to applications and documents. E,T.

caps = Connection Tool protocol list. *.

cbnd = Communication Toolbox Bundle. *.

CASH = RAM cache control code. C.

card = Video card names. *.

cctb = Control color look-up table. T.

CDEF = Control Definition. N.

cdev = Control Panel code. C.
 cicon = Color icon. E.
 clst = Cashed icon list. N.
 clut = Color look-up table. E,T.
 CMDK = List of Command keys used in ResEdit. T.
 cmnu = MacApp temporary menu resource. E,T.
 CMNU = Command Menu. MacApp menu's. E,T.
 CNTL = Control Definition Table List. T.
 CODE = Application's code. C.
 crsr = Color mouse pointer. E.
 CTY# = City list for MAP Control Panel device. T.
 CURS = Mouse pointer. E.
 dctb = Dialog color look-up table. E,T.
 dflg = ddev Flags. *.
 DITL = Dialog Items Table List. E,T.
 DLOG = Dialog location and size definition. E,T.
 dpsr = Edition Manger section. N.
 DRVR = Driver. printer, network or, desk accessory. C, T.
 DSAT = Dialog System Alter Table. Start-up and bomb alerts and all their code. *.
 eppc = EPP Configuration. *.
 fAni = Finder Stripe List. List of PICT ID's. Power Book. *.
 faps = File Transfer Tool protocol list. *.
 fbnd = Communication Toolbox Bundle. *.
 FBTN = MiniFinder button. T.
 fctb = Font color look-up table. T.
 FCMT = Finder's Get Info. comments stored in the Desktop file. T.
 FDIR = MiniFinder button directory ID. T.
 fdmn = Finder Definition Menu. A list of menu items for Power Books. *.
 FILE = Contents of ResEdit's "open Special" menu. N.
 finf = Font information. T.
 FKEY = Function key code. C.
 fld# = List of folder names. T.
 flst = Font List. T.
 fmenu = Finder Menu List. *.
 FMTR = Format record for 3 1/2 inch disks. C.
 fmts = Edition Manager formats. N.
 FOBJ = Folder information. N.
 FOND = Font family description. T.
 FONT = Font description. E,T.
 FREF = File reference. E,T.
 fval = Finder data. T.
 fview = Finder View. *.
 FRSV = ROM Font ID. T.
 FWID = Font width table. T.
 gama = Gamma table. Color correction for monitors. N.
 gmcd = Guard Mechanism for Compression and Decompression. *.
 GNRL = ResEdit General Preference Resource. T.
 hdlg = Balloon Help for dialogs. *.
 hldr = Balloon Help for Finder Icons. *.
 hmenu = Balloon Help for menus. *.
 hovr = Balloon Help Override List. *.
 hrct = Balloon Help - rectangles. *.
 hwin = Balloon Help for windows. *.
 icl4 = 4-bit 32 x 32 Finder icon. E.

icl8 = 8-bit 32 x 32 Finder icon. E.
 icmt = Comment for Installer 3.0 and later. T.
 icm# = 1-bit 8 x 8 and 16 x 16 icons with mask. Used by some applications. N.
 icm4 = 4-bit 8 x 8 and 16 x 16 icons. Used by some applications. N.
 icm8 = 8-bit 8 x 8 and 16 x 16 icons. Used by some applications. N.
 ICN# = 1-bit 32 x 32 Finder icon with mask. E.
 ICON = Icon used in dialogs, alters and, menus. E.
 ics# = 1-bit 16 x 16 Finder icon with mask. E.
 ics4 = 4-bit 16 x 16 Finder icon. E.
 ics8 = 8-bit 16 x 16 Finder icon. E.
 ictb = Item Color Table Bundle. Color dialog item list. N.
 inbb = Apple Installer 3.0 or later scripts. T.
 indm = Apple Installer 3.0 or later scripts. T.
 infa = Apple Installer 3.0 or later scripts. T.
 infs = Apple Installer 3.0 or later scripts. T.
 INIT = Code run at boot up . C.
 inpk = Apple Installer 3.0 or later scripts. T.
 inra = Apple Installer 3.0 or later scripts. T.
 insc = Apple Installer 3.0 or later scripts. T.
 INT# = Integer list. N.
 INTL = Old style itl0 and itl1. E.
 itl0 = Date, Time, and number formats. E.
 itl1 = International date and time information. E.
 itl2 = International string comparison. C.
 itl4 = International number conversion table. C.
 itlb = International script bundle. T.
 itlc = International script configuration. T.
 itlk = International exception dictionary for KCHR. T.
 KCAP = Physical layout of the keyboard. *.
 KCHR = Mapping Virtual key to character codes. E.
 kcs# = 1-bit 16 x 16 Keyboard Icon with mask. *.
 kcs4 = 4-bit 16 x 16 Keyboard Icon. *.
 KEYC = Old keyboard layout. N.
 KMAP = Keyboard mapping from raw key code to virtual key code. N.
 kscn = Small icons that correspond to KCHR. N.
 KSWP = Key plus modifier combinations. N.
 LAYO = Old Finder layout. T.
 LDEF = List Definition Table. List Manager. C.
 lmem = Globals to be switched by MultiFinder. N.
 mach = cdev filtering. *.
 MACS = Version number in System file. T.
 MBAR = MENU display set. T.
 MBDF = Menu Bar Definition code. C.
 mcky = Mouse Key. Speed in Mouse Control Panel. T.
 mctb = MENU color look-up table. E,T.
 mcod = MacroMaker information. N.
 mdct = MacroMaker information. N.
 MDEF = Menu Definition. C.
 mem! = MacApp memory utilization. N.
 MENU = Definition for standard application menus. E,T.
 minf = MacroMaker macro information. T.
 mitq = Default queue sizes for the MakeITable procedure. *.
 MMAP = Mouse tracking code. N.
 mntb = Command number to MacApp menus. N.

mntr = Monitor Extension code. items used in the Options dialog in the Monitors Control Panel. N.
 mppc = MPP configuration. *.
 mst# = MultiFinder string list. N.
 mstr = MultiFinder string list. N.
 NFNT = New Font Description. N.
 nrct = Rectangle position list. T.
 PACK = Packages of code used as ROM extensions. C.
 PAPA = Printer access protocol address used by AppleTalk. T.
 PAT = 1-bit 8 x 8 pattern. E.
 PATC = Pattern code. C.
 PAT# = Pattern list. A list of PAT resources. E.
 PDEF = Printer Driver code. C.
 PICK = Pickler Definition. ResEdit. T.
 PICT = Picture resource. E,T.
 pltt = Palette Color. E,T.
 POST = PostScript code. T.
 ppat = Pixel Pattern. Color patterns of variable sizes. E,T.
 ppcc = NBP Look-up Interval. Power Macintosh. *.
 ppci = Location of PPC Toolbox. Power Macintosh. N.
 ppt# = Pixel Pattern List. a list of the ppat resources. E.
 PREC = Printer driver data. T.
 PREF = Preference. ResEdit editor's preference information. N.
 PRER = Non-serial printer Chooser code. N.
 PRES = Serial printer Chooser code. N.
 proc = Procedure. C.
 prvw = Edition Manager. Similar to a PICT. N.
 pslt = NuBus Pseudo. Slot mapping. *.
 PTCH = ROM Patch. C.
 ptch = ROM Patch. C.
 qrsc = Database Access Manager query record. T.
 RDEV = Network Chooser code. N.
 RECT = Coordinates of a single rectangle. *.
 resf = Reserved Fonts. T.
 RMAP = Resource MAP. ResEdit's resource map. T.
 ROv# = ROM Overrides. A list of ROM resources to override. T.
 ROvr = ROM Override Code. C.
 RSSC = ResEdit editors and pickles. C.
 RVEW = ResEdit pickler view information. T.
 RZS = Owner Resource Registered by RZS(tm) for RZS(tm) ResBook. N.
 RZS™ = Owner Resource Registered by RZS(tm) for RZS(tm) ResBook. N.
 scrn = Screen Configuration. T.
 seg! = Segmentation Control. For MacApp. N.
 SERD = RAM serial driver code. C.
 sfnt = True Type outline font description. N.
 SICN = List of 1-bit 16 x 16 icons. E.
 SIZE = MultiFinder size information. T,*.
 snd = Sound resource. *.
 snth = Sound synthesizer code. C.
 STR = Strings characters. T.
 STR# = List of Strings characters. T.
 styl = Style. Used with TEXT resource. E.
 sysz = Size resource. Used by Extensions / Control Panels. Amount of RAM needed. *.
 taps = Terminal Tool Protocol list. *.

tbnf = Communication Toolbox Bundle. *.
 TEXT = Text. E,T.
 tlst = Title list. N.
 TMPL = Template. ResEdit resource template. T.
 TOOL = Fatbits editors tool layout. ResEdit editors. T.
 vers = Version Information. Finder's Get Info. E,T.
 wctb = Window color look-up table. E,T.
 WDEF = Window Definition code. C.
 WIND = Window location, size and type. E,T.
 wstr = Query string. Used by qrsc resource. T.

Mac OS X Info Property List

Name Type Required Description

CFBundleDevelopmentRegion String No The native region for the bundle. Usually corresponds to the native language of the author.

CFBundleDisplayName String No The localized display name of the bundle.

CFBundleDocumentTypes Array No An array of dictionaries describing the document types supported by the bundle

CFBundleExecutable String Yes Name of the bundle executable file.

CFBundleGetInfoHTML String No A string for displaying richer content in the Finder's Get Info panel

CFBundleGetInfoString String No A string for display in the Finder's Get Info panel

CFBundleHelpBookFolder String No The name of the folder containing the bundle's help files.

CFBundleHelpBookName String No The name of the help file to display when help is launched for the bundle.

CFBundleIconFile String Yes File name for icon image file

CFBundleIdentifier String Yes Unique identifier string for the bundle. This string should be in the form of a java package name, for example com.apple.myapp

CFBundleInfoDictionaryVersion String Yes Version information for the Info.plist format.

CFBundleName String Yes The short display name of the bundle

CFBundlePackageType String Yes The four-letter code identifying the bundle type

CFBundleShortVersionString String Yes The marketing-style version string for the bundle

CFBundleSignature String Yes The four-letter code identifying the bundle creator

CFBundleURLTypes Array No An array of dictionaries describing the URL schemes supported by the bundle

CFBundleVersion String Yes Build number of the executable.

CFBundleTypeExtensions Array This key contains an array of filename extensions that map to this type. To open documents with any extension, specify an extension with a single asterisk "*". This key is required.

CFBundleTypeIconFile String This key specifies the name of the icon file to be used when displaying documents of this type. The icon filename can have an extension or be without one. If it is without an extension, the system appends an extension appropriate to the platform (for example, .icns in Mac OS 9).

CFBundleTypeName String This key contains the abstract name for the document type and is used to refer to the type. This key is required and can be localized by including it in the corresponding InfoPlist.strings files. This value is the main way to refer to a type and it is recommended that you use a Java-style package identifier to ensure its uniqueness. If the type is a common Clipboard type supported by the system, you can use one of the standard types listed in the NSPasteboard class description.

CFBundleTypeOSTypes Array This key contains an array of four-letter type codes that map to this type. To open documents of any type, specify the four - letter type code "****". This key is required

CFBundleTypeRole String This key specifies the application's role with respect to the type. The value can be Editor, Viewer, Printer, Shell, or None. This key is required.

NSDocumentClass String This key specifies the NSDocument subclass used to instantiate instances of this document. Used for Cocoa applications only.

NSExportableAs Array This key specifies an array of other types that documents of this type can be exported as (write-only types). Used for Cocoa applications only.

CFBundleTypeRole String This key specifies the application's role with respect to the URL type. The value can be Editor, Viewer, Printer, Shell, or None. This key is required.

CFBundleURLIconFile String This key contains a string entry with the name of the icon image file (minus the extension) to be used for this URL type.

CFBundleURLName String This key contains a string entry with the abstract name for this URL type. This is the main way to refer to a particular type. To ensure uniqueness, it is recommended that you use a Java-package style identifier. This name is also used as a key in the InfoPlist.strings file to provide the human-readable version of the type name.

CFBundleURLSchemes Array This key contains an array of the URL schemes handled by this type. Examples of URL schemes include http, ftp, and so on.

CFAppleHelpAnchor String No The bundle's initial HTML help file.

NSAppleScriptEnabled String No Specifies whether AppleScript is enabled.

NSHumanReadableCopyright String Yes A copyright string used for display in dialog boxes.

- NSJavaNeeded** Boolean or String No Specifies whether the program requires a running Java VM.
- NSJavaPath** Array No An array of paths to classes whose components are preceded by NSJavaRoot.
- NSJavaRoot** String No The root directory containing the java classes.
- NSMainNibFile** String Yes The name of an application's main nib file.
- NSPrincipalClass** String Yes The name of the bundle's main class.
- NSServices** Array No An array of dictionaries specifying the services provided by an application.
- NSPortName** String This key specifies the name of the port your application monitors for incoming service requests.
- NSMessage** String This key specifies the name of the instance method to invoke for the service. In Objective-C, the instance method must be of the form `messageName:userData:error:`. In Java, the instance method must be of the form `messageName (NSPasteBoard,String)`.
- NSSendTypes** Array This key specifies an array of data type names that can be read by the service. The NSPasteboard class description lists several common data types. You must include this key, the NSReturnTypes key, or both.
- NSReturnTypes** Array This key specifies an array of data type names that can be returned by the service. The NSPasteboard class description lists several common data types. You must include this key, the NSSendTypes key, or both.
- NSMenuItem** Dictionary This key contains a dictionary that specifies the text to add to the Services menu. The only key in the dictionary is called `default` and its value is the menu item text. This value must be unique. You can use a slash character `/` to specify a submenu. For example, `Mail/Send` would appear in the Services menu as a menu named `Mail` with an item named `Send`.
- NSKeyEquivalent** Dictionary This key is optional and contains a dictionary with the keyboard equivalent used to invoke the service menu command. Similar to `NSMenuItem`, the only key in the dictionary is called `default` and its value is a single character. Users invoke this keyboard equivalent by pressing the `Command` and `Shift` key modifiers along with the character.
- NSUserData** String This key is an optional string that contains a value of your choice.
- NSTimeout** String This key is an optional numerical string that indicates the number of milliseconds Services should wait for a response from the application providing a service when a response is required.
- LSBackgroundOnly** String No Specifies whether the application runs only in the background. (Mach-0 applications only)
- LSPrefersCarbon** String No Specifies whether an application prefers running in the Carbon environment. **LSPrefersClassic** String No Specifies whether an application prefers running in the Classic environment.

LSRequiresCarbon String No Specifies whether the application must run as a Carbon application.

LSRequiresClassic String No Specifies whether the application must run in the Classic environment.

LSUIElement String No Specifies whether the application is a user-interface element, that is, an application that should not appear in the Dock or Force Quit window.

APInstallerURL String Yes A URL-based path to the files you want to install.

APFiles Array Yes An array of dictionaries describing the files or directories that can be installed.

APFileDescriptionKey String A short description of the item to display in the Finder's Info window

APDisplayedAsContainer String If "Yes" the item is shown with a folder icon in the Info panel; otherwise, it is shown with a document icon

APFileDestinationPath String Where to install the component as a path relative to the application bundle

APFileName String The name of the file or directory

APFileSourcePath String The path to the component in the application package relative to the APInstallerURL path.

APInstallAction String The action to take with the component: "Copy" or "Open"

PPC Processor Overview

USER MODEL UISA

General-Purpose Registers

GPR0 (32)
GPR1 (32)
⋮
GPR31 (32)

Floating-Point Registers

FPR0 (64)
FPR1 (64)
⋮
FPR31 (64)

Condition Register

CR (32)

Floating-Point Status and Control Register

FPSCR (32)

XER Register

XER (32)	SPR 1
----------	-------

Link Register

LR (32)	SPR 8
---------	-------

Count Register

CTR (32)	SPR 9
----------	-------

USER MODEL VEA

Time Base Facility¹ (For Reading)

TBL (32)	TBR 268
TBU (32)	TBR 269

SUPERVISOR MODEL OEA

Configuration Registers

Machine State Register

MSR (32)

Processor Version Register

PVR (32)	SPR 287
----------	---------

Memory Management Registers

Instruction BAT Registers

IBAT0U (32)	SPR 528
IBAT0L (32)	SPR 529
IBAT1U (32)	SPR 530
IBAT1L (32)	SPR 531
IBAT2U (32)	SPR 532
IBAT2L (32)	SPR 533
IBAT3U (32)	SPR 534
IBAT3L (32)	SPR 535

Data BAT Registers

DBAT0U (32)	SPR 536
DBAT0L (32)	SPR 537
DBAT1U (32)	SPR 538
DBAT1L (32)	SPR 539
DBAT2U (32)	SPR 540
DBAT2L (32)	SPR 541
DBAT3U (32)	SPR 542
DBAT3L (32)	SPR 543

SDR1

SDR1 (32)	SPR 25
-----------	--------

Segment Registers

SR0 (32)
SR1 (32)
⋮
SR15 (32)

Exception Handling Registers

Data Address Register

DAR (32)	SPR 19
----------	--------

DSISR¹

DSISR (32)	SPR 18
------------	--------

SPRGs

SPRG0 (32)	SPR 272
SPRG1 (32)	SPR 273
SPRG2 (32)	SPR 274
SPRG3 (32)	SPR 275

Save and Restore Registers

SRR0 (32)	SPR 26
SRR1 (32)	SPR 27

Floating-Point Exception Cause Register (Optional)

FPECR	SPR 1022
-------	----------

Miscellaneous Registers

Time Base Facility (For Writing)

TBL (32)	SPR 284
TBU (32)	SPR 285

Data Address Breakpoint Register (Optional)

DABR (32)	SPR 1013
-----------	----------

Decrementer

DEC (32)	SPR 22
----------	--------

External Access Register (Optional)

EAR (32)	SPR 282
----------	---------

Processor Identification Register (Optional)

PIR	SPR 1023
-----	----------

HAVE FUN WITH THIS STUFF AND GOOD LEARNING

ManiacoMac™ @ 02/2004
All softwares that are good enough deserve to be cracked

