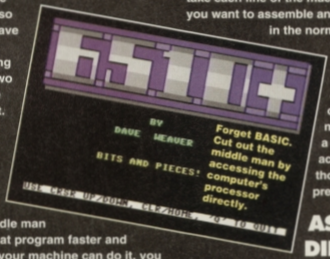
6510+

Unlock the true programming power of your C64 with this excellent machine language assembler.

An assembler, for all those who haven't been following Dr Finch's Casebook, is a program that takes your assembly language instructions and turns them into a machine code program that your C64's processor (big chip) can understand. BASIC, essentially, is just a machine code program, you see, so BASIC programs have got to be run by a program that's being run on the chip – two programs to do the job of one, in effect. This is slow. Programming in machine code means you're running one program directly, cutting out the middle man (BASIC), making that program faster and more powerful – if your machine can do it, you can make it do it in machine code.



You think you don't need this? Every arcade game that's ever been given a good mark has been written in machine code – BASIC just can't cut it. You still need to know how to use this program, though, don't you?

Right – the program you'll find on the disc will install a copy of *6510+* on the tape or disc of your choice. The package itself is way too complex to go into here, but anyone who's been coding for any amount of time will already know exactly how to use it.

What if you're a beginner? Well I can't explain how to program in machine code here (there's a new feature starting on page 23 that does all that stuff), but I can explain how this assembler works, so you'll be able to use it with the various excellent technical features *CF* has.

Your listing is written in the form of a BASIC program using mnemonics as opposed to normal BASIC commands, that 6510+ will take and assemble; in other words, when you take each line of the machine language listing you want to assemble and give it a line number in the normal BASIC way.

After the actual assembly instruction (that's the bit that Jason will teach you over the next six months), you can place a comment which will act like a REM statement, though it will have to be preceded with a ';' or a '!'.

ASSEMBLY DIRECTIVES

Not all commands have to mnemonics, by the way – there are some directives that, instead of assembling into real instructions, give the assembler itself specific instructions (if you're new to assembly language most of most make much sense yet, but it will soon if you carry on reading the Mean Machine series):

BYT – This is used to place bytes directly into memory. The correct format for this command would be something like `BYT 57,38,53`, which would place the bytes 57, 38 and 53 consecutively. By the same token, you can give the command a text string and the assembler will place the ASCII values of each character into memory instead.

TXT – This is the same as `BYT`, but you can use it if you're used to that command instead (some people prefer it that way apparently)

WOR – This one places a two-byte word into memory. The command `WOR $1234` would place the bytes $34 and $12 consecutively in memory.

END – This command is used to indicate the last line of source code.

CHN – You can also use `LNK`, as both commands are used to chain two source files together; if you used `CHN "filename"`, the assembler would switch to the new source file to continue assembly. Bear in mind that all chained files must end with an `END` directive.

LIB – This is very similar to `CHN`, but allows you to insert a chunk of source code from elsewhere into your listing. This command doesn't so much switch files as paste another one in.

ASSEMBLE IT, THEN

When you've finished your program, you'll want to assemble it, won't you. The easy way to do this is with the ASSEMBLE command, that doesn't need any other parameters. If you'd like to start assembly from a particular line number, though, you can state that line number (ASSEMBLE 50) and *6510+* will happily oblige.

One point – *6510+* modifies a few other BASIC commands in order that it can work properly. While this is great and everything, it does mean that some normal BASIC programs may not work with the assembler in memory. You have been warned – steer clear of running normal BASIC programs with *6510+*

6510+ IDIOT'S GUIDE

We're going to use a program from Mean Machine Code, the tutorial series that starts this month on page 23. The listing can be found under the heading 'Hands on time' within the feature, and here's how we'd use it with 6510+.

1. Load 6510+
2. Enter the following program in BASIC:

```
10 LDA #$00
20 STA $D020
30 RTS
```

3. Type the command ASSEMBLE.
4. There we go. All done. Easy, wasn't it?

MAYHEM MEGAMIX

More melodic mayhem from the master mixers John and Steve Rowlands. Sit back and listen to some of the grooviest games music in the history, all of 'em from Apex games, including *Creatures*, *Creatures 2* and *Mayhem in Monsterland*.

And it's all wrapped up in one of the best-looking option screens you're ever likely to see. Just hit the function keys to select what game you want and then hit the letter corresponding to the tune that you want to hear. To start and stop the music (plus a few other effects) use the Mayhem cursor at the bottom of the screen, which is controlled by your joystick. Then sit back and enjoy.

QUICKSTART INFO

F1-F7	Select game.
A-Z	Select tune.
JOYSTICK	Move Mayhem.

Other Info You select the various functions (play, stop, etc) by moving Mayhem over the monsters at the bottom of the screen and pressing fire to select. It's got to be one of the best-looking control systems we've ever seen.

MEAN machine CODE

PART ONE

Programming in Basic is more restrictive than living on a student grant, but the prospect of learning machine language – with all those meaningless numbers and weird symbols – can seem daunting. But never fear, Jason Finch is here with a beginners' guide that's about as daunting as the Pepsi challenge.

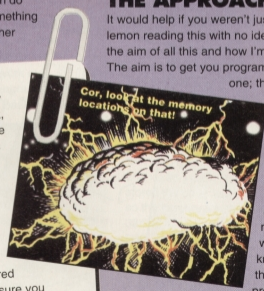
The two words 'machine' and 'language', when used consecutively, can make even the sanest of people quiver at the knees, and can send less hardy specimens crying for their mothers. I can even hear some of you screaming and frantically trying to turn the page already, but don't; it will be worth the pain, the frustration, and the mental torment in the end – mastering machine code opens up limitless possibilities as far as programming goes.

For months you have seen Basic loaders in Techie Tips, short programs full of DATA statements which load some machine code into memory and then do something rather stunning. By the time this series has finished you will know exactly why they all work and you will know how to change them to do what you want. And now, using either the assembler on this month's covertape or the Action Replay cartridge, you'll get the chance throughout this series to mess about with real machine language, or ML as it's known in the trade. For the moment I'll be referring to the AR cart when I'm dealing with how to run the machine language listings, as it's slightly more straightforward to see what's going on, but from next month, I'll be gearing the series towards using *6510+*. But feel free to experiment in the meantime.

WHAT IS... A MEMORY LOCATION?

Although I have covered memory locations in Dr Finch's Casebook, I'm going to repeat them because the concept of memory locations is fundamental to learning machine language.

Numbers are stored in memory locations; those numbers are called bytes. The different memory locations have different functions and depending on the numbers stored in them, do different things. Ensure you understand memory locations before you continue with the series. Memory locations are also used in ML to store values because you do not have variables.



THE APPROACH

It would help if you weren't just sat there like a lemon reading this with no idea why. So I'll tell you the aim of all this and how I'm going to approach it. The aim is to get you programming in ML from day one; that's simple. I'll do it by telling you about the commands and the different ways of using the commands. I'll tell you about the concepts involved and about different memory locations. I won't assume you know anything about this type of programming at all; I won't assume you've ever programmed a computer at all before, although I will make reference to Basic. I won't bore you with pointless details about hardware, about the CPU and about other complicated issues that are of little relevance to the get-your-hands-in-and-do-it approach that I'm going to take. The deal is that I babble and give examples, and you listen and test out the examples I give. Fair enough, yes?

WHAT IS... MACHINE LANGUAGE?

This seems like the most obvious question for a series like this. Machine language is a set of instructions that your computer understands without having to convert them into anything else. It is also a whole new way of thinking, so far as programming is concerned. Oh yes, forget about variables. The days of arrays and using every known letter of the alphabet have gone. Strings don't exist any more either. Say goodbye to your colons and your line numbers. The real programming is about to begin.

YOUR FIRST TIME

The first time you do anything is always the worst time because you have to learn a load of things before you can even begin to do anything constructive. But hang on in there. It might seem a bit an uphill struggle now, but persevere – the ecstasy of printing your name on the screen in Machine Language comes a close second to eating a really good pizza (*er, yes, okay, I'll have to take your word on that on, – Dave*).

In later months you'll be able to refer back to the 'What is ...?' sections and easily find what

WHAT IS... HEXADECIMAL?

It's even more complicated than binary, basically. Hexadecimal is a number system that uses the numbers 0 to 9 and the letters A to F. Eh? Yes indeedy, letters are involved as well. The following table gives a conversion of normal (decimal) and hex numbers. We'll use a dollar ($) sign to show a hex number (it's the standard way).

Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
0	$0	4	$4	8	$8	12	$C
1	$1	5	$5	9	$9	13	$D
2	$2	6	$6	10	$A	14	$E
3	$3	7	$7	11	$B	15	$F

You see, although 'hexadecimal' has got more syllables than there are letters in most words (well the kind of words you get in *The Sun*, anyway), it's not all that complicated. Decimal 16 is hex $10, and you carry on counting from there. You can use hexadecimal without actually understanding how numbers are converted between it and decimal, assuming you have either a calculator with that function, a cartridge with it or a nifty little program like the *HEX VEX* one that I listed in Techie Tips a short while back. Hex is used merely because it is convenient; single byte numbers are represented by two 'hex digits' and double byte numbers are represented by four.

WHAT IS... A MACHINE LANGUAGE PROGRAM?

A machine language program is a list of instructions that is stored in memory from a particular memory location onwards. For example, you might start a Basic program at line 10; with a ML program you may start it at address 49152. This is a convenient memory location on the C64 and is $C000 in hex.

you're looking for. The main fun will begin towards the end of this first part and by which time you'll have written your very first machine language program. It would help if you had an Action Replay cartridge – and I'm assuming you have one – purely for its machine language monitor, though. Most utility cartridges have monitors, so be prepared to be flexible if you don't actually own an AR cartridge. I'll also be making reference to things out of Dr Finch's Casebook which I hope you have been following (*if not, well there are back issues available, see page 34 – Ed*).

THE FUN BEGINS

Okay, people, hang on to your seats because the fun is about to begin. You've read all the 'What is ...?' bits, you've read all the relevant Dr Finch's Casebook stuff, now you're ready to begin.

In ML, a load is an LD and a store is an ST. You follow this with the register name that you want to load from or store to. So LDA, LDX, LDY and STA, STX, STY are all valid ML instructions. Loads need something to load and stores need somewhere to store things. Imagine you want to load the number 0 into the accumulator (the A register). In Basic you'd do A=0. Decimal 0 is hex $00, but because we want the NUMBER 0 and not MEMORY LOCATION 0, we put a hash sign before it: #$00. This is the operand (see the What is...? box on operators and operands with the brilliant accompanying picture).

The instruction to load something into the accumulator is LDA (LoaD into the A register). So `LDA #$00` is the first instruction. Wow, we've actually got a machine language instruction here!

What about storing? Well STA seems reasonable (STore the A register) but you need to specify somewhere to store it. We give the instruction a memory address as the operand.

WHAT IS... AN OPERATOR AND AN OPERAND?

An operator is a thing that operates; it carries out some particular job. An operand is the thing that is operated on. When you get a jar of coffee off a shelf, you are the operator. The coffee is the operand. Referring to things in computer-speak, the act of getting the coffee is the, 'loading' and is split into knowing what you want – the coffee (the operand) – and you thinking about reaching for it (the operator).

53,280 is $D020 in hex, and controls the border colour. So `STA $D020` could work, right?

The final thing we need to do is stop the program (it does help). In Basic this is done automatically, or with an END statement; in ML you use an instruction called RTS (ReTurn from Subroutine) to get you out of a program.

HANDS-ON TIME

You want to write a machine language program consisting of the following three commands (yes you do, you know you do, go on admit it):

```
LDA #$00
STA $D020
RTS
```

In order to do this, you must 'assemble' the program, one instruction at a time. The computer needs to know where you want to put this program in memory, and the most common place for beginners is 49,152, or $C000 in hex. The Action Replay monitor must be told the start address for assembling an instruction. It will then prompt you for the next instruction until you press Return without giving an instruction.

Bad visual gag alert! Bad visual gag alert!

WHAT IS... A REGISTER?

Good question, that one. A register is a place where things are stored, or registered. In machine language you have three registers, called the accumulator, the X-index and the Y-index; A, X and Y for short. Each register can store a one byte number, that is a number from 0 to 255.

ASSEMBLING

Get your monitor up and running and enter `A C000 LDA #$00` and hit Return. If you've done it right, you should see this on your screen:

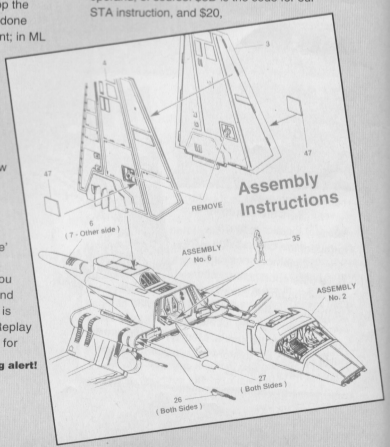
```
.> C000 A9 00     LDA #$00
.A C002
```

Now, where the cursor is flashing, enter STA $D020 and press the Return key, now enter RTS and press the Return key twice. You should have this on your screen now:

```
.> C000 A9 00     LDA #$00
.> C002 8D 20 D0  STA $D020
.> C005 60        RTS
.> C006
.
```

You have now assembled a three-line machine language program; it has one load instruction, one store instruction, and one return. The numbers down the left are memory locations and the numbers immediately to their right are the values stored at those locations. So, $A9 (169 in decimal) is stored at $C000 (49152 in decimal), $00 is stored at the next location, $8D at the next, then $20, $D0 and finally $60 is stored at $C005.

These all mean something (*and I thought they were just there to confuse me – Dave*). $A9 is the code for our particular LDA instruction, $00 is the operand, of course. $8D is the code for our STA instruction, and $20,



WHAT IS... BINARY?

If you haven't been following Dr Finch's Casebook then you have committed the greatest of sins, you haven't learned about binary. Binary is a number system that uses only 0s and 1s. The following table gives a conversion of normal (decimal) and binary numbers. We'll use a percent (%) sign to show a binary number.

Dec .....Bin	Dec .....Bin
0.....%0000	1.....%0001
2.....%0010	3.....%0011
4.....%0100	5.....%0101
6.....%0110	7.....%0111
8.....%1000	9.....%1001
10.....%1010	11.....%1011
12.....%1100	13.....%1101
14.....%1110	15.....%1111

$D0 is the two-byte address. You should remember this notation from the 'What is a byte?' question. The $60 is the code for RTS. You do not need to remember those codes; they are the raw machine code – the same as the values that are POKEd into memory when you use a Basic loader to store code.

RUNNING

To run that machine language program you use the G command, for 'Go'. So enter G C000 and be impressed! Okay, so the border has turned black. You're not particularly excited, I can tell. Perhaps we can turn the background black as well to make you a bit more happy. With the cursor flashing after a dot, enter `A C005 LDA #$00` and press the Return key. Now enter `STA $D021`, press Return again, enter RTS and press Return twice. You should see:

```
.> C005 A9 00     LDA #$00
.> C007 8D 21 D0  STA $D021
.> C00A 60        RTS
.> C00B
.
```

Enter `G C000` again and the whole screen should go black. Why does that happen? (You've tripped over the power lead? – Dave) Well, you take the number 0 and you put it into the accumulator. You then take the number in the accumulator and store it at memory location 53,280. That changes the border colour the same way as `POKE 53280,0` would do in Basic. The same occurs with the second load and store; you take the number 0 again and put it into the accumulator. You then store that at location 53,281, changing the background colour. Check out the colour codes in the table below and change the $00 to experiment.

$00 black	$08 orange
$01 white	$09 brown
$02 red	$0A pink
$03 cyan	$0B dark grey
$04 purple	$0C medium grey
$05 green	$0D light green
$06 blue	$0E light blue
$07 yellow	$0F light grey

WHAT IS... A MACHINE LANGUAGE MONITOR?

A machine language monitor, or MLM, is a program that allows you to enter and view machine language programs. They often do other things as well, but they are secondary. The Action Replay cartridge has a MLM built into it which can be accessed from Basic by either pressing F8, entering MON and pressing the Return key or select MON from the option screen that comes up when when you press the smaller button on the back. You should see something like this come up on your screen:

```
    ADDR AR XR YR SP 01 NV-BDIZC
.;E37B 00 FF 00 FB 37 00000000
.
```

The cursor will be flashing next to the dot – you have entered the realm of ML. No-one can hear you scream! I suggest you read the MLM section of your manual to work out exactly how to use it. For AR owners, enter X and press Return to get back to Basic.

DISASSEMBLING

As Newton discovered, every action has its opposite and equal reaction, and the opposite of assembling is disassembling, you probably won't be too surprised to hear. Assuming that you've been diligently following all the instructions I've been giving you so far (well I like to think there is some purpose in my life) within your AR monitor, enter D C000 C005 and you should see:

```
.D C000 C005
.> C000 A9 00     LDA #$00
.> C002 8D 20 D0  STA $D020
.
```

This should not come as a great shock to you because that is what you originally entered (if, as I said, you've been following my instruction, otherwise god knows what you might get - some rude error message most likely); this is part of your machine language program.

You can go up and edit any of the lines, in exactly the same way you would with a Basic program. For example, move the cursor over the second 0 of the `LDA #$00` instruction, press 2 and then Return. Press Return twice more to get back to the dot prompt. Now type in `G C000` again and you'll see the effect. Ah, the wonders of modern science. Okay, it's not exactly ground-breaking stuff, and your Gran is the only person who'll say she's impressed (and not even she means it), but, come on, what do you expect after one introductory lesson – *Mayhem In Monsterland*? You've got to walk before you can run. From little acorns do mighty oaks grow. One small step for… (*yes, thank you, Jason, see you next month – Dave*).

WHAT IS... A BYTE?

Apart from being the cause of many a hilarious joke in early 80s sitcoms (in conjunction with the many culinarily named computers there were about then), what is a byte? Your computer has 65,536 different memory locations; think of them as boxes into which you can place bytes of data. 53,280 is a memory location that just happens to control the border colour; POKE 53280,0 sets it to black. One byte can be stored at each location. A byte, on the C64, is a number between 0 and 255 inclusive. 255 in decimal is %11111111 and in hex it is $FF.

A two-byte number is one which takes up two bytes in memory (shocker); numbers from 256 up to 65,535 need two bytes – a 'low' byte and a 'high' byte. To get the final number you take the high byte, multiply it by 256 and add the low byte. So, for example, 53,280 is stored with high byte 208 and low byte 32 because 208 x 256 + 32 = 53,280. In your C64 they are stored the other way around, to be confusing, so the number 53,280 would be stored as 32,208 in memory. In hex, these are $20 and $D0. And 53280 in decimal is $D020 in hex.

Check out the similarity with the high byte and the low byte conversions. That's why hex is so convenient. It's a lot easier to see that the high byte of $D020 is $D0 than it is to work in decimal and say the high byte of 53,280 is 208.

WHAT IS... LOADING AND STORING?

Imagine you are in a supermarket and one of the things you want is a loaf of bread. You pick up the bread and you put it in your trolley (*hang on, I can feel an extended metaphor coming on here – Dave*). This is loading and storing. Loading is the action of getting the bread, and storing is the action of putting it in the trolley. The same happens in Basic when you do `POKE 53280,0`. The computer 'loads' something or other with the number 0 and then 'stores' it at location 53,280. A simple analogy for the simplest of operations.

NEXT MOVE

Before next month I want you to make sure you have read your MLM manual, or the relevant section of your Action Replay booklet. I'll explain about addressing modes, give more details on loading and storing, show you how to use the other registers, how to display text on the screen and how to use the compare instructions. You'll get more chance to get your hands dirty, as it were, because the background information has been covered here.