THIS IS A PRE-RELEASE VERSION OF 6502ASM.. IT MAY NOT WORK ENTIRELY CORRECTLY YET. If you discover errors, please report them to me!

6502asm v0.04

user guide

Introduction

6502asm is a simple assembler for 6502 code. It supports a variety of assembler commands to control output and extend the 6502 instruction set.
The instruction set provided is the legal NMOS 6502 set, and the CMOS 65C02 instruction set according to the Western Design Center W65C02S datasheet (not the MOS 65CE02 variant).
The NMOS "undocumented" instructions are not supported.

The command syntax is:

```
6502asm <input file> <output file>
```

Purpose

6502asm is part of the *Amélie* project. It was written in order to allow the BIOS code to be written and compiled without requiring the use of a BBC microcomputer (or emulator). While it is primarily intended for this purpose, it is flexible enough to be used for the assembly of 6502 code for other environments, such as the BBC micro (or compatible) or other 6502-based devices.

6502asm fits in with *Amélie*'s philosophy of "**keep it simple**".

6502asm assembles in two passes, much as code is usually assembled under BBC BASIC. Currently, *6502asm* does not support macros or conditional assembly, though these features are "in planning".

If you are used to Acorn systems, you may be pleased to know that *6502asm* considers '`&`' to mean "this is a hex number", which is one of the main reasons I wrote this instead of using one of the many other 6502 assemblers available. If, however, you are not used to Acorn, you can use the Pascal prefix '`$`' or the C prefix '`0x`' to denote a hex number.

> The DOS-like (an '`h`' suffix) and the VisualBasic-like ('`$H`' prefix) methods are ***not*** supported.
> The *Amélie* project source code uses '&' throughout.

The format of a source file

The source code provided to *6502asm* is a plain text file consisting of a number of lines. These lines can be comments, directives, compiler commands, label definitions, or instructions.

The only thing you must note *right* from the outset is that *the assembler only understands one entity per line*. If, for example, the line is a label, then any code following must begin on the next line. Furthermore, the use of the colon to write multiple statements on one line is not supported. In difference to the BBC BASIC assembler, you should use ';' for comments and not '\' or 'REM'; though the assembler reads only what is necessary so there is some leeway.

Don't write code like this: `.zeroregs LDX #0 : LDY #0 : LDA #0 : RTS`
The assembler will see: `.zeroregsldx` (a label called "zeroregsldx")

Don't write code like this: `LDX #0 : LDY #0 : LDA #0 : RTS`
The assembler will see: `LDX #0` (only the first instruction, the "LDX")

The source file is *always* read from the start to the end. It is *possible* to assemble in any location in the `&FFFF` (64K) addressing space at any given time by setting the address with the **org** command, however over-use of this can lead to messy code and other complications.

Where execution starts depends on what you are assembling. Typically a program is entered at the beginning of the code; while a(n) (EP)ROM image is entered at the location of the processor's *reset* vector.

The format of a line

There are three types of line:

a. Labels

```
.<label>
```

Labels are on a line by themselves, and consist of a period followed by the label name. The names are unique to forty-seven characters. Label space is allocated as the label is encountered, so there shouldn't be any restrictions other than available memory. It is perfectly valid to define multiple labels at the same location, for example:

```
   ORG  &A000
.via
.via_base
.via_iorb
   DCB  0
```

After that, any reference to *via* or *via_base* or *via_iorb* will be treated as a reference to &A000. This is how the memory-mapped hardware is set up in *Amélie*'s source code. The DCB inserts a 'dummy byte' (as only the EPROM code is actually saved). It is a lot tidier to do that than to alter the address with **org** for every label we define (the VIA has 16 of them...).

Labels are unique to 47 characters. Anything longer will be truncated.

It is currently possible to define the same label multiple times. If this happens, only the *first* will be used when referring to that label.

b. Assembler commands

```
<command> [<parameters>]
```

Assembler commands are special commands understood by the assembler. More on these later.

c. Instructions

```
<opcode> [<parameter>]
```

Instructions are the usual three letter mnemonics, with optional parameters as necessary.

Comments are introduced using the '`;`' character. Anything following a semicolon until the end of the line is ignored. You can include comments at the ends of command lines and instruction lines.

To reiterate: Each line is a separate entity, up to 80 characters long.
You cannot split commands across lines, nor can you put multiple commands on one line.

The 6502 (instruction set)

The 6502 processor is an 8 bit design from the late '70s. In its day it was more popular than the Intel 8086/8088; and in many ways was more advanced than the Z80 which was its main competitor. Indeed, objective tests (running a BBC BASIC interpreter) shows the Z80 clocking at 4MHz to be only about 12% faster than a 6502 clocking at 2MHz. Later extensions and optimisations (such as those found in the CMOS variants of the processor) will enhance the effective speed.

With its direct and simplified instruction set, the 6502 has even been described as a forerunner to modern RISC microprocessors.

It offers two 8 bit 'index' registers, though these are fairly general purpose. They are referred to as '`X`' and '`Y`'. The results of all mathematical operations are placed in the *Accumulator*, often called '`A`'. The data bus is 8 bits wide.

Instructions are between one and three bytes long.

The address bus (and '`PC`' (*Program Counter*) register) is sixteen bits wide with no translation, meaning a 6502 can only directly address 64K of memory. It is possible to extend this with external fiddling. The BBC Master 128 computer contained 128K accessed via a paging mechanism, and I dimly recall a version with more than that!

Many popular home computers of the '80s were built around the 6052; the *BBC micro* (including the *Electron* and *Master*/*Master Compact*, and "*Acorn Communicator*" used in many a travel agent), the *Apple II*, the *Dragon*, the *Oric*, Commadore's *PET*...

The 6502 has three built-in branch points (IRQ, RESet, NMI) in the upper six words of memory (`&FFFA` to `&FFFF`). These are known as the hardware vectors. The first of these is the interrupt handler (and by using the BRK instruction, you can force an interrupt from software). The second is called when a reset condition occurs. Finally, the last is the non-maskable interrupt. This differs from the normal IRQ in that you cannot switch it off and it can occur while handling a normal interrupt. It is used for things that need an extremely fast response time (in the case of the BBC micro, this was usually Econet and the floppy disc drive). The *Amélie* project uses the NMI for a debounced 'panic' button which will halt the system regardless of whatever else is happening (and even if interrupts are disabled).

The hardware stack lives in page one (`&0100` to `&01FF`). Certain operations performed using the zero page addressing mode (relating to memory locations `&0000` to `&00FF`) operate much more quickly than do the same instructions applied to any other part of the memory map.

As you can understand, this design *dictates* that RAM will be at the bottom end of memory and (EP)ROM will be at the top; though some systems (such as the Acorn *FileStore*s) copy the entire firmware to RAM at startup, and then page out the ROM entirely. This is because RAM can have a faster access time than EPROMs...

The 6502 instruction set is well documented on-line.

Briefly:

ADC	ADd with Carry	CPY	ComPare with Y register	PLP	PulL Processor status from the stack
ADC	ADd with Carry	DEC	DECrement	ROL	ROtate Left
ASL	Aritmetic Shift Left	DEX	DEcrement X register	ROR	ROtate Right
BCC	Branch if Carry Clear	DEY	DEcrement Y register	RTI	ReTurn from Interrupt
BCS	Branch if Carry Set	EOR	logical Exclusive OR	RTS	ReTurn from Subroutine
BEQ	Branch if EQual	INC	INCrement	SBC	SuBtract with Carry
BIT	test BITs	INX	INcrement X register	SEC	SEt Carry
BMI	Branch if MInus	INY	INcrement Y register	SED	SEt Decimal mode
BNE	Branch if Not Equal	JMP	absolute JuMP	SEI	SEt Interrupt disable
BPL	Branch if PLus	JSR	absolute Jump to SubRoutine	STA	STore Accumulator
BRK	BReaK	LDA	LoaD Accumulator	STX	STore X register
BVC	Branch if oVerflow Clear	LDX	LoaD X register	STY	STore Y register
BVS	Branch if oVerflow Set	LDY	LoaD Y register	TAX	Transfer Accumulator to X register
CLC	CLear Carry	LSR	Logical Shift Right	TAY	Transfer Accumulator to Y register
CLD	CLear Decimal mode	NOP	No OPeration	TSX	Transfer processor Status to X register
CLI	CLear Interrupt disable	ORA	logical OR (with Accumulator)	TXA	Transfer X register to Accumulator
CLV	CLear oVerflow	PHA	PusH Accumulator to the stack	TXS	Transfer X register to processor Status
CMP	CoMPare	PHP	PusH Processor status to the stack	TYA	Transfer Y register to Accumulator
CPX	ComPare with X register	PLA	PulL Accumulator from the stack		

The CMOS version adds:

BBR#	Branch if Bit # Reset (clear)	BBS#	Branch if Bit # Set	BRA	BRanch Always
PHX	PusH X register to the stack	PHY	PusH Y register to the stack	PLX	PulL X register from the stack
PLY	PulL Y register from the stack	RMB#	Reset (clear) Memory Bit #	SMB#	Set Memory Bit #
STP	SToP	STZ	STore Zero (to address specified)	TRB	Test and Reset memory Bit
TSB	Test and Set memory Bit	WAI	Wait (for interrupt)		

For more details, I will refer you to `http://www.6502.org/`

Within *6502asm*, you can enter the instructions in upper case or lower case, as suits you.

For more advanced users, the instruction set recognised is loaded from a file ("`opcode.dat`"), so you could alter several of the mnemonics if you desire – for example to 'correct' how the processor status register is called '`P`' in the stack instructions and '`S`' in the transfer instructions; or to alias '`ADD`' and '`SUB`' to '`ADC`' and '`SBC`' respectively...
It will not be possible to include support for the 65CE02 because of major design differences (16 bit stack pointer, additional addressing modes, a 'Z' register...).

The 'Exx' instructions at the end (all opcode `&xB`) are specific to *AmélieEm*; if you wish to use *6502asm* for other purposes, you can safely remove these instructions.

Note that there is a clash with the instruction `&CB`. This is used as a breakpoint instruction in *AmélieEm*, and it is also the `WAI` instruction on CMOS processors. This is not seen as a problem as the emulator only supports the NMOS instruction set.

You could also use the opcode table to implement various "undocumented" instructions that may be found in the NMOS versions of the 6502, some examples would be:

AXS This ANDs the contents of the X register and the Accumulator and stores the result in memory. Neither register is altered, and the processor flags are not changed either.
Regular code equivalent to "`AXS &20`" would be:

```
STX &20
PHA
AND &20
STA &20
PLA
```

DCM This DECs the contents of a memory location, then CMPs the result with the contents of the Accumulator.
Regular code equivalent to "`DCM &20`" would be:

```
DEC &20
CMP &20
```

HLT Halts the processor. Causes some sort of internal crash. No interrupts will be handled, the only way out is to wibble the RST pin in hardware. Because the exact operation of this instruction is unknown, it might be preferable to set up dummy IRQ and NMI handlers and then enter a recursive loop if you really want the processor to appear to be 'halted'.

LAX This loads both the Accumulator and the X register with the contents of a given address.
Regular code equivalent to "`LAX &DEAD`" would be:

```
LDA &DEAD
LDX &DEAD
```

It is important to remember that these four instructions (four of many) are totally *unofficial* and operate as side effects of other instruction decoding. They are not present on the CMOS versions of the processor, and it may also depend on who manufactured that particular NMOS 6502...

But, if you do have them, then they might just provide an interesting little speed tweak!

Here are the additions to make to the *opcodes.dat* file.

```
;     AccImmAbsZpaZpxZpyAbxAbyImpRelInxInyAbiEmuZpi
axs,0,--,--,8F,87,--,97,--,--,--,--,83,--,--,--,--
dcm,0,--,--,CF,C7,D7,--,DF,DB,--,--,C3,D3,--,--,--
hlt,0,--,--,--,--,--,--,--,--,02,--,--,--,--,--,--
lax,0,--,--,AF,A7,--,B7,--,BF,--,--,A3,B3,--,--,--
```

You'll find a comprehensive list of the known "undocumented" 6502 instructions on-line.

Assembler commands

6502asm commands are all three letters long (except `EQUx` which is handled specially).
They look like processor instructions. This is intentional.

BOT <address> [**FORCE**]

This specifies the lower address of the assembly. If no address is specified, `&0000` is assumed...
...*unless* the **ROM** command has been previously specified, in which case `&E000` will be assumed.

Once **BOT** has been set, you cannot re**BOT** to a *higher* address *unless* you include the **FORCE** option.
The **FORCE** option is primarily for use with ROM code where you want to denote a specific address (i.e. `&E000`) as being the 'bottom' of memory and the start address of the output file ... but you also wish to define labels in memory, such as in page zero.
You could use code such as:

```
ROM
ORG   &0000
<define some labels here>
ORG   &A000
<define some hardware-related labels here>
BOT   &E000 FORCE
<ROM code goes here>
```

CNT "<file>"

The "continue" command will switch to processing the contents of the named file. You could use this to 'chain' multiple sources to create one assembled file.

CPU [**NMOS**|**CMOS**]

This permits you to specify which processor you are assembling for.

Undefined	– all instructions and addressing modes are valid (*default behaviour*)
NMOS	– warnings will be given if CMOS instructions or addressing modes are encountered (but the instructions will be assembled)
CMOS	– all instructions and addressing modes are valid

DCB <byte> *also* **EQUB <value>**

Include a byte value in the output.

DCS "<string>" *also* **EQUS "<string>"**

Include a string in the output. Several 'codes' may be embedded within the string, these are described later on.

DCW <value> *also* **EQUW <value>**

Include a word (16 bit) value in the output, low byte first (6502-style).

DCZ "<string>" *also* **EQUZ <value>**

Include a zero-terminated string in the output.

FIL <count>, <value>

Insert <count> bytes (of <value>) into the output, to fill or pad as desired.

INF "<name>"

Load and embed the contents of the named file. You can use this to insert raw data or *pre-compiled* code.

ORG

Set the value of "**PC**", and where the assembly will write to next.
It is permissible to jump all around the addressing space, but for obvious reasons this is not to be recommended.

Note that there are side effects – setting an address lower than that specified by **BOT** or higher than that specified by **TOP** will *update* the respective marker. Bear this in mind if you **ORG** to a low location after a **ROM** command.

ROM

The ROM command performs a number of actions that may be useful to quickly set up firmware intended to be held in ROM or EPROM.
Initially, *all* bytes in the memory map will be set to the value &FF. This should permit faster EPROM programming (as blank EPROMs are all-bytes-&FF, so the programmer will skip them).

Then, the following sequence is performed:

```
BOT  &E000          ; ROM starts here (&E000-&FFFF = 8192 bytes)
TOP  &FFFF          ; ROM ends here
ORG  &FFFA          ; set up vectors
DCW  nmi_vector     ;   NMI
DCW  reset_vector   ;   RESet
DCW  irq_vector     ;   IRQ
ORG  &E000          ; back to start
```

It is then up to you to provide the labels (and code) for *nmi_vector*, *reset_vector*, and *irq_vector*.

The default base address, `&E000`, was chosen because it is the start point of the *Amélie* EPROM. If this address is not suitable, you can easily do something like:

```
ROM
BOT  &A000 FORCE
ORG  &A000
```

TOP <address>

This specifies the higher address of the assembly. If no address is specified, the address after the last instruction assembled is assumed...
...*unless* the **ROM** command has been previously specified, in which case `&FFFF` will be assumed.

Once **TOP** has been set, you cannot re**TOP** to a *lower* address. This is important as you cannot use the **ROM** command to set up an arbitrary EPROM environment for, say, BBC paged ROMs. An example of how to do this would be something like:

```
BOT  &8000                ; set the bottom
ORG  &8000                ; go there
FIL  &3FFF, &FF           ; fill the area with &FF bytes
TOP  &BFFF                ; affirm the end location
ORG  &8000                ; back to the beginning
; code follows, i.e.:
DCW  entrypoint_language
DCW  entrypoint_service
DCB  %11100010            [...etc...]
```

It is important to note that the values set by the **BOT**, **ROM**, and **TOP** commands directly relate to the range saved in the output file.

For example:

```
BOT &0000
TOP &FFFF
```

will cause 65,535 bytes (64K) to be saved to file.

```
BOT &F000
INX
```

will cause one byte to be saved to file.

```
ROM
```

will cause 8192 bytes to be saved to file.

Ways of specifying a numerical value

When you need to specify a value, say a constant or an address, you can use:

Base 16 (hex):

`&12`	This is the Acorn way.
`$12`	This is the Pascal / Commadore way.
`0x12`	This is the C way.

The DOS ("`12h`") and VisualBasic ("`$H12`") forms are *not* supported.

Base 10 (denary):

`123`	As expected...

Base 2 (binary):

`%1100`	The Acorn way.

To recap, prefix with '`&`' for hex and '`%`' for binary. No prefix is necessary for denary numbers.
You can also use '`$`' and/or '`0x`' to prefix hex values if you are used to doing it that way. Note that the C style, if you've never used C before, is '`0x`' which is *zero-ecks* (not *oh-ecks*).

Alternatively, in the case of addresses, simply define a label and then use the name of the label. For example:

```
.never_ending
   JMP  never_ending
```

In-line calculations

When you are assigning values, such as:

```
LDX  #43
```

you can also use *calculations*. Calculations are marked using square brackets. Within these square brackets you can enter a sum which is evaluated left-to-right, like:

```
LDX  #[12+42-11]    ; evaluates to be 43!
```

The available mathematical operators are:

+	Addition
-	Subtraction
*****	Multiplication
/	Division
%	Modulus (remainder after division)

The available logical operators are:

>>	Logical shift right
<<	Logical shift left
||	Logical OR
&&	Logical AND
^^	Logical Exclusive OR

This will have slightly more use when it comes to inserting the addresses of things, such as:

```
TXA
STA buffer
TYA
STA [buffer + 1]
```

The calculation works by extracting the parts of the calculation to a 'stack', so our example **[12+42-11]** would break down to be:

location:	0	1	2	3	4	5	6	7	
value:	**12**	**+**	**42**	**-**	**11**	**]**	**x**	**x**	('`]`' meaning end and '`x`' meaning unused)

The stack can hold up to **eight** items (locations 0 to 7). More complex calculations will be faulted.

Addressing modes (NMOS)

This is a look at the addressing modes provided on the 6502 processor. It is given as a useful reference, and also to describe how to format instructions that use a particular addressing mode.

Implicit (or Implied) addressing

```
<instruction>
```

No value is required. Memory may be referenced, but it will be done in ways you cannot control. For example `PHA` will push the contents of the accumulator to the next free stack location.
You can set up the stack, but not with the `PHA` instruction...

Examples:

```
CLI
RTS
```

Accumulator addressing

```
<instruction> A
```

The operation is performed with/to the accumulator. ***The 'A' is important***, unlike some assemblers *6502asm* will not 'guess' whether or not you meant to use the accumulator addressing mode.

Examples:

```
ASL  A
ROR  A
```

Immediate addressing

```
<instruction> #<value>
```

The value is directly provided. Note the '#' character before the value; if it was not there then it would be taken as an address – the '#' is important!

Examples:

```
LDA  #123
```

Relative addressing

```
<branch instruction> <offset|label>
```

Relative addressing is used only with branch instructions. The second byte of the instruction establishes a branch point which may be -128 to +127 bytes away from the following instruction (i.e. -128 to +127 bytes away from PC).

This is why you cannot `BEQ` across large expanses of code.

Examples:

```
BEQ  <label>
BNE  <label>
```

The assembler sorts out the offset, you don't have to.

Absolute addressing
Zero page addressing

```
<instruction> <address>
```

The parameter to the instruction is a byte held at a fixed address.
If the address is less than `&100`, zero page addressing is used otherwise absolute addressing is used.
Zero page addressed instructions can operate a lot faster, so careful use of page zero can be a definite speed bonus.

Examples:

```
CMP  &FE02   ; compare A with byte at &FE02
LDA  &C5     ; load A from &C5 ==ZERO PAGE==
```

It is *very important* to note that *all* references to a label fall into this category of addressing. If the label has been resolved *before* it is encountered in the code, then *6502asm* will pick Absolute or Zero page addressing as is deemed appropriate.
However, if the label has *not* been seen before it is used (i.e. you are assembling a forward branch) then *6502asm* will ***always*** treat this as an full absolute (not zero-page) address.

In 99.9% of cases, this behaviour is correct, however if the label is later defined as being Zero Page, then *6502asm* will *then* assemble the code as if it was Zero Page and everything else will be wrong, your software will crash, etc. *This is only of concern if you are assembling code to be located or used within Page Zero itself. Simply pre-define all of the labels first, and then go back and assemble code. Don't use forward references.*

Indexed absolute addressing (aka **Absolute, X** or **Y** addressing)
Indexed zero page addressing (aka **Zero page, X** addressing)

```
<instruction> <address>, X
<instruction> <address>, Y
```

This behaves much like the Absolute / Zero page addressing previously described, however in addition to retrieving the address, the contents of the X or Y register (as applicable) is added.

For example:

```
    LDX  &DD         ; set X to &DD
    LDA  #&F000, X   ; load A from (&F000
                     ;              + &DD)
```

The zero-page versions work entirely in zero page, so adding `&FF` to `&FF` won't work. The effective address would be `&FE`.

Examples:

```
    CMP  &FE02, Y    ; compare A with byte
                     ;   at (&FE02 + Y)
    LDA  &C5, X      ; load A from &C5
                     ;   ==ZERO PAGE==
```

Zero page, Y addressing

```
LDX <value>, Y
STX <value>, Y
```

There is, generally, no such thing as Zero page, Y addressing as the zero page indexed addressing uses the X register.
The specific exception to the rule is when we are loading and saving the X register – we can't indirect it with itself!

Examples:

```
    LDX  &0F, Y
    STX  123, Y
```

Indexed indirect addressing (**pre-indexed**)

```
<instruction> (<zero page address>, X)
```

Ready? The contents of the second byte of the instruction are added to the X register. This then gives us an address in page zero where we can expect to find our 'real' target address.

This addressing mode is mainly used for interfacing multiple peripherals. You can store a list of pointers in memory, and by using `base address + index`, you can read data from each peripheral in turn simply by repeating the same code using the index register to select which to load.

Example:

```
    LDA  (&12, X)    ; load A from
                     ; address pointed to
                     ; at (&12 + X)
```

Indirect indexed addressing (post-indexed)

```
    <instruction> (<zero page address>), Y
```

This time... The second byte of the instruction points to an address in page zero. We add this new address to the contents of the Y register to obtain our 'real' target address.

Example:

```
    ORA  (&12), Y    ; ORA with byte at
                     ; address pointed to
                     ; by:
                     ;  (Y + value at &12)
```

For a walk-through, refer to the memory dump below:

```
&0000  A4 02    LDY  &02
&0002  B1 05    LDA  (&05), Y
&0004  60       RTS
&0005  07       ???
&0006  00       BRK
&0007  01 02    ORA  (&02, X)
&0009  03       ???

&0000 A6 02 A1 05 60 07 00 01 02 03 ª·í·`.....
```

First we set `Y` to `&02`.
The **LDA** command retrieves the value "`&0007`" from address `&0005`.
So now the **LDA** looks to `&0007 + Y`, which is `&0009`.
The value actually loaded into the accumulator is `&03`.

Where *indexed indirect* could be used to interrogate multiple peripherals, the *indirect indexed* is more suited to reading and writing multiple *registers* within one peripheral.

Consider the 6522 VIA. It contains 16 registers:

```
 0  ORB       5  T1C-L     10  SR
 1  ORAh      6  T1L-L     11  ACR
 2  DDRB      7  T1L-H     12  PCR
 3  DDRA      8  T2C-L     13  IFR
 4  T1C-L     9  T2C-H     14  IER
                           15  ORA
```

If the parameter address pointed to the base address of the 6522 in the memory map, you could use the index register to select which register to read. On the BBC micro, the user port is at `&FE60` to `&FE6F`; so we could *indirectly index* memory locations `&FE60 + Y`...

Absolute indirect

```
JMP (<address>)
```

This mode is only available for a specific type of **JMP** instruction.
The contents of the address pointed to are read, and this in turn becomes the **JMP** address.

For example:

If `&F000` was `&A000`...

```
JMP (&F000)
```

JMP will look at `&F000` and 'see' `&A000`. Instead of jumping to `&F000`, it will indirect, and we'd jump to `&A000`.

Example:

```
JMP  (&0F00)
```

Amélie's BIOS makes extensive use of this for interrupt and event vectoring, by indirecting though a known location, we only add a few extra cycles (and waste five bytes). The upside is that the application code can intercept the interrupt handling at various points.
Additionally, if you look at the BIOS startup code (from *rst_vector*), you'll see that the BIOS sets up a dummy "just in case" interrupt handler while the hardware is set up, and then the proper interrupt handler is put in place after the hardware has been initialised. Without indirect jumps, such things would not be possible.

Zero-page absolute indirect

```
<instruction>  (<address>)
```

This mode is only available on the CMOS versions of the processor. The address specified is a location in zero page which points to a two-byte effective address.
This is provided on the 65C02 for use with `ADC`, `AND`, `CMP`, `EOR`, `LDA`, `ORA`, `SBC`, and `STA`.

Support for four-byte instructions

The 65C02 provides use with a number of instructions that are correctly four characters in length:

BBR0	BBR1	BBR2	BBR3	BBR4	BBR5	BBR6	BBR7	BBS0	BBS1	BBS2	BBS3	BBS4	BBS5	BBS6	BBS7
RMB0	RMB1	RMB2	RMB3	RMB4	RMB5	RMB6	RMB7	SMB0	SMB1	SMB2	SMB3	SMB4	SMB5	SMB6	SMB7

It was intended that there be a translation table to read these four-character instructions and convert them to three-character versions. This is still 'in the works' as it tended to break a lot more than it would have fixed (because of the assumption of three-character instructions, which was fine for original 6502 code).

There is a work-around. The assembler will recognise these instructions if you omit the middle letter, so that an instruction such as `BB`***`B`***`R3` will become `BR3`. Here is the above table in *6502asm*-friendly format:

BR0	BR1	BR2	BR3	BR4	BR5	BR6	BR7	BS0	BS1	BS2	BS3	BS4	BS5	BS6	BS7
RB0	RB1	RB2	RB3	RB4	RB5	RB6	RB7	SB0	SB1	SB2	SB3	SB4	SB5	SB6	SB7

If you absolutely require the four-character versions to be supported, please get in touch.

String codes

You can 'embed' special sequences within strings:

`\r`	Insert an `&0D` byte (linefeed)
`\n`	Insert an `&0A` byte (newline)
`\"`	Insert a double quote
`\'`	Insert a single quote
`\x##`	Insert a hex code directly (i.e. `\x07` for TTY bell)
`\d###`	Insert a decimal code directly (i.e. `\d169` for © symbol)
`\0`	Insert a null byte
`\\`	Insert a backslash

This has been taken from the C method of inserting codes into strings...

My so-called example

We will create a file called "*mybios.s65*". It contains a very stripped-down EPROM image, basically to save cluttering up this document with a lot of trivialities!

```
; example BIOS code (based upon Amélie)
;
; To be compiled with 6502asm
; http://www.heyrick.co.uk/amelie/
;

   ROM             ; base & org = &E000, top = &FFFF,
                   ; inserts vector links


;          P A G E   Z E R O
;          =================
;
; Data labels live here, but no tables or code can reside here...

   ORG  &0020      ; at +32 we have various BIOS locations

   ; WATCHDOG:
   ;   Application code MUST periodically reset this to zero.
.watchdog
   DCB  0

   ;           ******* much stuff omitted *******
```

```
   BOT  &E000 FORCE; force low address to be &E000, EPROM start

   ; The above is important!
   ; Because we are assembling ROM code, we want only &E000 to
   ; &FFFF to be saved, but we also want to set up locations
   ; in page zero so we can refer to them by label.


;          B I O S   C O D E   F O L L O W S
;          ================================

   ORG  &F000      ; The BIOS sits in the second half of the EPROM


; R E S E T          The "reset_vector" label is required by
; ---------          required by the ROM instruction

.nmi_vector        ; because NMIs are not used in this example...
.reset_vector
   NOP
   SEI             ; should be already, but no guarantees
   CLD             ; undefined at NMOS 6502 init

   LDX  #&FF       ; reset stack pointer
   TXS
   ; we'd reset hardware devices here too
   ; and suspend all sources of IRQs

   ; ******* lots snipped *******


   CLI             ; Re-allow interrupts

   ; BIOS startup complete! Call application...
   JMP  app_code_entry


.irq_vector
   NOP
   RTI             ; In reality, a LOT more would happen!


   CNT  "appcode.s65"  ; Continue in the "appcode" file.
```

Meanwhile the "*appcode.s65*" file will look like:

```
;          A P P L I C A T I O N   C O D E   v0.01  2001/01/01 at 01h01
;          ==============================

   ORG  &E000      ; The application code is the first half of the EPROM

.app_code_entry
   JMP  app_code_entry  ; do nothing, it's just an example!
```

This is just to give you an 'idea' of the layout of the code. To make things tidier, the BIOS and the application code are in separate files. This would also be the case if a debugger was included. In this manner it is possible to mix and match parts; perhaps different application codes (depending on needs/features) using the same BIOS? Perhaps debugging support in test versions but not release versions? More flexible than a giant single source file.

To assemble, enter at the command line:

```
6502asm mybios.s65 mybios.img
```

For a successful assembly, the display will look like:

```
S:\Amélie>6502asm mybios.s65 mybios.img
6502asm v0.04 (11th June 2008) initialising...
Written by Rick Murray, email me at <heyrick1973 -at- yahoo.co.uk>

Assembling, pass 1...
Continued into file "appcode.s65"...
Assembling, pass 2...
Continued into file "appcode.s65"...
Saved 8192 bytes to "amebios.img" - &E000 to FFFF

6502asm v0.04 (11th June 2008) exited. Thank you for using this software.
Don't forget to check for newer versions and related software!
        http://www.heyrick.co.uk/amelie/


S:\Amélie>
```

The file "*mybios.img*" will now contain the assembled BIOS image; as a raw dump; suitable for use with *AmélieEm* or burning into EPROM.

Error messages

ERROR: Unable to open input file. – *you may see this regarding the* **output** *file*
Cannot open "*<filename>***" (error** *<error code>***)**
6502asm cannot open files specified using "long" names. Please use the DOS filename (probably something like "APPLIC~1.S65"). The error code provided may give some clues to more experienced users, if the reason for the problem is not evident.

ERROR: Unable to open opcode data table.
Is the "OPCODE.DAT" file present in the currently selected directory? If you are running *6502asm* from a PIF or shortcut, ensure the "Working directory" is correctly set.

ERROR: Unable to allocate memory for opcode data table.
Free up some memory and try again...

Base address (&*<address>***) is higher than Top address (&***<address>***), cannot continue!**
You've set BOT higher than the current TOP value.

Base address (&*<address>***) is same as Top address - nothing to save!**
The end of the file has been reached and the bottom and top addresses match. Did you try to assemble an empty file?

ERROR: Attempting to assign BOT to a higher address at line *<line>***.**
(use "BOT &xxxx FORCE" if this is what you meant to do)
As it says – use the FORCE option if you intend to reassign the bottom marker to a higher address.

ERROR: Unable to extend label array.
There isn't enough free memory to add a new label definition.

ERROR: Memory allocation failure.
Please free up an additional 72K and try again.

In order to begin assembly, *6502asm* allocates a 64K block of memory at start-up. The additional memory (it says 72K) is because if you are *this* short of memory, we'd better ensure we have room for the opcode table and some labels...

Under MS-DOS™, you may also see:

Available memory is only <*filename*> **bytes, need at least 73728 (72K)...**

or:

Oops, total memory cockup!

This is because the DOS memory allocation scheme is really sucky. You may even get this on a 128Mb machine if you boot directly to DOS (not via Windows™).

ERROR: CNT command cannot open file at line <*line*>.
(filename is "<*filename*>**")**

You cannot continue into the file specified as the file cannot be opened.

ERROR: Attempting to DCB a value larger than &FF at line <*line*>.

The DCB command inserts a byte, but you've specified a value that is too large to fit into a single byte...

ERROR: Unknown EQUx at line <*line*>.
Should be EQUB, EQUW, EQUS, or EQUZ (re: DCx)

The `EQUx` commands are simply aliases for the `DCx` commands. "`EQU`" alone has no meaning.

ERROR: Invalid count (<*count*>**) in FIL command at line** <*line*>.
(syntax is "FIL <count>, <value>" - are the parameters transposed?)

You will probably see this if you wanted to write a number of null bytes, but put the parameters the wrong way around; i.e.

```
FIL 0, 256
```

ERROR: Missing ',' in FIL command at line <*line*>.

Perhaps you only specified one parameter assuming a default state (like "`FIL 32`" would write out 32 null bytes?).

ERROR: INF command cannot open file at line <*line*>.
(filename is "<*filename*>**")**

You cannot insert the contents of the file specified as the file cannot be opened.

ERROR: Relative address to &<*address*> **(**<*offset*>**) out of range (-128...127) at line** <*line*>.

Due to the way relative addressing works, you can only branch to an address that is 128 bytes *before* or 127 bytes *after* the address of the following instruction. The '<offset>' lets you know what the range calculated was, in case it is something you could fix with a little bit of jiggling, like an offset of -130 bytes...

ERROR: Unexpected end of calculation at line <*line*>.

Calculations ***must*** end with the '`]`' character.

ERROR: Division by zero in calculation at line <*line*>.

Obvious.

ERROR: Calculation too complex at line <*line*>.

The calculation 'stack' can only hold eight discrete items. Every number and operator is an item in the stack, thus `[1+2+3]` is six items. Why six? Don't forget the end-of-calculation marker.

ERROR: The instruction "<*instruction*>**" is not recognised.**
(at line <*line*>**)**

You have entered something that is not understood as a valid instruction or pseudo-instruction.
A common cause of this is forgetting the period before labels, for example:

```
mylabel
```

will be parsed as the instruction "`myl`", which is not a valid instruction, hence this error message...

ERROR: Value is too large at line <*line*>.

An eight bit (0-255) result was expected.

ERROR: Unable to resolve label "<*label*>**" (at line** <*line*>**).**

The label (or its definition) has been incorrectly typed, or has not been defined.

WARN!: Use of CMOS instruction when NMOS processor specified;
"<*opcode*>**" at line** <*line*>**...**

You specified `CPU NMOS` and the assembler encountered a 65C02-only instruction.

ERROR: Unable to match opcode with addressing mode at line <*line*>.
[lots of addition information output]

You should not see this error. It occurs if the addressing mode cannot be determined from the input; possibly due to a damaged opcode data file, possibly due to really screwy input.

ERROR: DCS or DCZ command with no string parameter at line <*line*>.

If you use DCS or DCZ, you must supply a parameter, even if it is a blank parameter, like:

```
DCS   ""
```

ERROR: String contains \d value greater than &FF at line <*line*>.

You can only insert bytes into strings, so the `\d` values can only range from 0 to 255.

ERROR: Unrecognised escape code "<*code*>**" at line** <*line*>.
(valid codes are \r \n \" \' \x## \d### and \0)

6502asm does not support all of the escape codes that are supported by the C programming language, i.e. `\a`*lert*, `\b`*ackspace*, `\t`*ab*, `\o`*ctal*, `\?`, etc).

ERROR: Memory overshoot at line <*line*>**, current address is already &**<*address*>**!**

This occurs if the assembly would pass beyond the &FFFF addressing range of the processor. The following snippet illustrates this:

```
ROM
ORG   &FFFD
DCW   &1234
DCW   &5678
DCW   &9ABC
```

#commands not yet implemented!

You cannot currently use "preprocessor" commands such as `#ifdef`.

Known bugs

Lines are clipped at 80 characters. This means if a comment on a line exceeds this, it will be read in as part of the *next* line, causing an error.
If you receive the error "`unknown command 'xxx' at line <blah>`", where "xxx" is part of a word in the comment, this is why.

In some cases, forward references to a label does not work. This is especially important in zero page where the label is not already known to be in zero page, so it is assumed to be an absolute address. Define zero page labels *first*, then `ORG` back...

No other bugs are known about.
That doesn't mean there aren't any... `:-)`

Licence

I, Richard Murray, permit you to use *6502asm* according to the following conditions:

1. This software, source, and documentation are copyright: Copyright © 2004-2008 Rick Murray

2. Source modifications *must* be reported back to me.
   All copyright and URL displays must be retained.
   If you release your modified version, it must be under these same conditions and you must also state clearly, both in the documentation and on-screen that the user is *not* using the official release of *6502asm*.

3. This software (and source) may be re-distributed, provided the end-user can access it *anonymously* and *at no cost*.
   For example, if I were to call your BBS (or connect to your portal), I *expect* to be able to download this immediately, from the 'Guest' login.

4. If this software is *not* supplied for free (including magazine cover-mounts, compilation CD-ROMs, etc) then I expect either a freebie or profit participation, at my discretion. This also includes 'bundling' this software with or within a commercial application.
   The definition of "commercial" is: *something for which the user must pay (directly or indirectly)*

5. If any part of the source is used in a third-party application, condition #4 applies to that as well. If this source is used in a "free" application, then simply provide a credit (with URL link, if possible) in an 'info' window or some other form of on-screen message.

6. This software is supplied as-is, and comes with no warranty or guarantee that it will be suitable for your needs or that its operation will be error free. I can accept no liability for loss or damage through incorrect use or incorrect behaviour of this software. Please report all problems to me.

7. Your "sole remedy" in case of problem is to contact me by email regarding the problem that you are experiencing. I *may* then provide a work-around or updated version in order to resolve your problem.
   No official end-user support is provided, nor can it be expected. You did *not* pay for this software, and source code is supplied. I will help out where I can, but please note that I have other stuff happening – it's called "having a life" and it's a whole new experience for me! `:-)`

8. Please note that this source code has ***not*** been released under the GNU Public Licence ("GPL"). Furthermore, I ***expressly prohibit*** any GPLisation of this source code in any way, shape, or form.
   Remember – the GPL is *just another licence* and it has no legal rights or special privileges that allow it to "override" or "replace" any existing licence *unless* the lawful owner of the source code permits this.
   As lawful owner of this source code, I *do not* permit this.

   My problem is not with the GPL, it is with numerous people that support the GPL and make it into something that it is not; if I was completely against the principles of open source then I wouldn't have bothered to release the source code. Some people may object to my referring to this as "open source" while putting conditions on its use. I apply

these conditions firstly to benefit the end-user, so that they will (hopefully) encounter the correct and up to date version and not spin-offs; and secondly to benefit myself, as I've been ripped off before and taken flak for problems in software versions that were not my own. Once bitten, twice shy.

9. I am a British citizen living in France.
This *entire* software product, and all documentation, was created within the confines of département 35, France.
Thus, this software and its use is under the jurisdiction of appropriate French and European legislation.
The application of American law is ***expressly prohibited***.

10. This software is *enirely* my own creation. I have used concepts (such as the "org" command) common in other assembler, notably Acorn's *objasm* and BBC BASIC II, but the actual implementation and source code is entirely my own. I'm not going to discuss the appalling crap known as "infringement of software patents". Such nonsense, very thankfully, is untenable within the European Union... *let's keep it that way!*

Them's the rules.
If you don't like them, you know what you can do about it...

Contact

You can email me at:

heyrick1973 -at- **yahoo** -dot- **co** -dot- **uk**
I request that you do ***not*** make this email address 'public' with the '@' and the '.'; and certainly *do not* write it in a newsgroup posting! ***At all!***

The Amélie project can be found at:

http://www.heyrick.co.uk/amelie/

Misc

6502asm was mostly written while sitting in bed feeling ill. Ironically, this document was first written about a year after the first version of *6502asm* was made... having just been to the dentist and having all four nerves removed from a rear molar awaiting a crown fixing, and one nerve totally refused to stop, aaaaaaagh! The revision was written whilst feeling very ill from eating something which is best described as "possibly tainted". It isn't at all helpful that the toilet is the other end of the house through half a dozen doors and windy passageways...

There must be something about *6502asm* and pain. Gee, and silly little me thought that x86 code was where true pain lay?

A technical achievement, the first working versions were written on a DOS-based laptop that used a 256 colour LCD panel comprising of four segments...and *only* segments 1 and 3 (top quarter height and middle-lower quarter) worked! I listened to various CDs, mainly *Evanescence*'s "Fallen", *The Corrs'* "Unplugged", *Alizée*'s "Mes Courants Electriques", and *Laura Pausini*'s "(Best Of) e ritorno da te".

For the updates and the user guide, I listened to *Dido*'s "Life For Rent" album, as well as *Sita*'s "L'Envers Du Décor" album. I also borrowed an album of *Céline Dion*'s from the library just to see what she did with her cover of the song "First Time Ever I Saw Your Face". A choice word might be "calamitous".

Please excuse any typing errors – I was watching (again) "*La Morte Vivante*" (or "*The Living Dead Girl*" in English), while writing this. :-)

Thanks to:

Ewen Cathcart for lots of help, support, and introducing me to something better than my old *Erasure* and *Pet Shop Boys* tapes... I guess it could have been worse, it could have been *Rick Astley*...

Glenn Richards for hosting my website (and you thought it was just a bunch of pictures of Alyson Hannigan, right?) and a lot of time chatting to me on the phone. I miss those days. And thank you also for laughing at me when I gave you the wrong hex opcode for the NOP instruction. It is &EA. I know that now, and it's come in useful recently!

John and ***Irene Williams*** for all those creature comforts that the French just can't seem to get right; like tea that tastes like tea, apple pies that taste like apple pies†, and cheese that tastes anything like cheddar! We won't discuss the French concept of baked beans... Thanks also to John for those eclectic bits of hardware, too!

† they'd lynch me over in Normandy for saying something like that, but to my mind *any* apple pie made without a Bramley or two is doomed to failure...

Zone Horror (formerly *The Horror Channel*) for a nice line-up of movies.
Find over 100 reviews at **`http://www.heyrick.co.uk/ricksworld/digibox/thcreview.html`**

Jean Rollin for surprising me with "*Le Rose De Fer*", which is quite different from the other films of his, and is a perfect example (to my mind) about what the Hollywood industry just *doesn't get* about European film-making.

FilmFour for a nice line-up of movies designed to stimulate your mind.
Find over 100 reviews at **`http://www.heyrick.co.uk/ricksworld/digibox/film4review.html`**

The now-defunct (sob! sob!) ***AnimeCentral*** for... well, isn't the name a big hint?
Find lots of information and reviews at **`http://www.heyrick.co.uk/ricksworld/anime/`**

Motorola and ***Rockwell*** for the 6502 processor.

Acorn for using the 6502 processor, and making the BBC micro which really shows it off (unlike, say, an Oric!).

EnVol for the French course, interesting new friends, and more besides.

SuperU for adding a 500g pack of tagliatelles fraîches to the "*Bien Vu!*" (budget) range, and also for having checkout girls worth looking at... :-) One day, I might even be able to talk to them too! Thank you especially to a newly built local SuperU which carries a decent amount of British food. Sadly no apple pies, but at least cheddar is sorted...

Sandrine, ***Françoise***, and ***Tiphaine*** at the local library for trusting me to use their computers with a USB memory device, and also permitting me to write my downloads to a multisession CD-R. Without that, *none* of this would be possible.

Anne-Marie, ***Caroline***, ***Philibert***, ***Odile***, ***Mme Hervoir*** and others whose name I don't know for their efforts to find me a job and the help in negotiating the beaurocracy.

Mike, ***Jo***, ***Lucy***, and ***Emily***... Now I'm no longer a Care Assistant, I can say that ***Doris*** was my favourite. I don't think any of the family were geek-inclined so they'll probably never see this. No matter, it's the thought that counts. :-)

I'm sure I've forgotten enough people to fill several more pages. Oh well, look at the end of helpfiles for some of my *other* software!

And finally...

Thank ***you*** for reading.

Rick; 2008/06/12