XA 2.1.4

65(c)02 Cross Assembler

(c) Andre Fachat

email: fachat@galileo.rhein-neckar.de

What it is

This Cross-Assembler makes programms for another computer that has a 6502-like CPU. This CPU has been widely used in the famous Apple II, all the Commodore 8-Bit Computers (PET, VC20 and a derivate in the C64) and many others. Some are still used in one-chip microcontrollers, e.g. the Rockwell modem chipset. All these chip share a common set of standard machine language commands, some of them (e.g. the CMOS versions) have additional (software) features.

I had the idea for this assembler when I built my small 6502 System that had place for 32kByte ROM to take the kernel and lots of other programms. (After all, it became a multitasking micro-kernel with file-systems for IBM and Commodore, I can even use the IBM drives as Floppy for my C64 with this computer as controller. Piping and i/o-redirection included, of course) Development on my old C64 began to suck with programms growing. So I decided to do a Cross-Assembler on my new Atari ST.

First versions were very like the old Assembler on the C64, not really using the resources (Reading all files two times completely etc). With files growing the assembler also became more sophisticated. Now hashcodes are used for mnemonics, preprocessor definition and label search (Version >= 2.0.5). The files are only read once, putting the preassembled code into memory (Version >= 2.0), taking it from there on pass 2. Now it makes about 350kByte Source Code to about 30kByte ROM code in less then 2 Minutes on an 8 MHz Atari ST with 2.5 MByte RAM and Harddisk. (Well, the Atari is not fast. On my 486DX4/100 it takes about 2 seconds...) But adding the whole relocation stuff slowed it down again.

Parameters and features

The assembler contains only a single programm called "xa" (for Atari: XA.TTP). It takes one or more Source files into one object file, that can directly be used. But the assembler also has a mode to produce relocatable files, conforming to the 'o65' fileformat (See fileformat.txt).

Call:

```
xa [options] Source1 [Source2 ...]
```

Object: this is the name, the output (object) file gets Error: Here you will find the Error listing. Label: this is the label list

```
'-C'            gives error codes when using CMOS-opcodes. Default is not to
                complain.
'-c'            do not produce o65 executable, but object files that can
                contain undefined references.
'-v'            go into verbose mode
'-x'            old filename behaviour (overrides -o, -e and -l)
'-R'            do not produce absolute code, but do relocation and all that.
'-o filename'   set output filename
'-e filename'   set errorlog filename
'-l filename'   set labellist filename
'-r'            add crossreference list to labellist output
                (i.e list of filename/line where label is used)
'-M'            allow ':' to appear in comments after a semicolon (MASM mode)
'-b? adr'       set segment start address for ? = t(ext), d(ata), b(ss) or
                z(ero) segment.
'-A adr'        If the _file_ starts at adr in a ROM, then the text segment
                need not be relocated. That of course only works, if the
                data/bss/zero segments are not occupied by other programs too!
'-G'            omit writing the exported globals to the file.
'-B'            Show lines with '.(' or '.)' pseudo opcodes
'-Llabel'       defines 'label' as absolute, undefined reference
'-DDEF=TEXT'    define a preprocessor replacement
'-Ipath'        additional include path for include files. Is evaluated before
                the XAINPUT environment variable. One path per '-I',
                multiple '-Ipath' allowed.
```

Omitting the errorfile or labelfile Parameter will cause xa to not write these files. Using '-x' will cause xa to take the name of the first source file and change the extension (on an Atari there is only one, like in DOS) to 'obj', 'err' and 'lab' respectively - if the old behaviour is selected with the '-x' option or the files are defined with "-l" and "-e". If no output file is given, "a.o65" is used.

Environment variables:

You can use the variables XAOUTPUT and XAINPUT to adjust the directory structure. If source or include files are not found, the Path in XAINPUT is being searched for the files. The different paths are separated by a comma (','). XAINPUT gives the directory where the *.obj, *.err and *.lab files are put. If they are not set, there will be no search, respectively the files are saved to the current directory.

The label file is a readable ASCII-file and lists all the labels together with their block-count (see below) and their address. The error file lists the version of the assembler, date and time of the assembler run, all the error messages and the stuff being printed with #echo and #print and last but not least a statistics of used resources.

6502 Assembler

xa supports both the standard 6502 opcodes as well as the CMOS versions (Rockwell 65c02). Not supported are the 6502 undocumented opcodes, they have to be put in by hand (with ".byte" directives).

For an introduction to 6502 Assembler please see elsewhere. A (very) short introduction is given in the german version of this text.

Some Assembler specific details:

When using addressing modes that could be zeropage or absolute, zeropage will be taken if possible. This can be prevented by prefixing the address with a '!'. Then absolute addressing is taken, regardless of the address.

Values or Addresses can be expressed by arithmetik expressions with hierachy and bracket. The following operands are understood:

```
123       -decimal
$234      -hexadecimal
&123      -octal
%010110   -binary
*         -program counter
"A"       -ASCII-code
labelx    -label
-(lab1+1) -expression
```

The following operands can be used (third column is priority):

```
+         -addition                   9
-         -subtraction                9
*         -multiplication             10
/         -integer-division           10
<<        -shift left                 8
>>        -shift right                8
>=,=>     -more or equal              7
<=,=<     -less or equal              7
<         -less                       7
>         -more                       7
=         -equal                      6
<>,><     -not equal                  6
&&        -logical AND                2
||        -Logical OR                 1
&         -Bitwise AND                5
|         -Bitwise OR                3
^         -Bitwise XOR                4
```

Operators with higher priority are evaluated first. Brackets can be used as usual.

Valid expressions are, e.g.:

```
LDA       base+number*2,x
```

For Addressing modes that do not start with a bracket, you can even use a bracket at the beginning of an expression. Otherwise try this:

```
LDX       (1+2)*2,y               ; Wrong!
LDX       2*(1+2),y               ; Right!
```

Before an expression you can use these unitary operators:

```
<      Gives the low byte of the expression
>      Gives the high byte

LDA  #<adresse
```

Single Assembler statements are being separated by a ':' (You remember the C64 :-) or a newline. Behind Each statement, separated by a ';' you can write some comments. The next colon or a newline ends the comment and starts a new statement. In MASM compatibility mode ('-M' command line option), then a colon in a comment is ignored, i.e. the comment lasts till the newline.

Pseudo opcodes, Block structures and where Labels are valid

In addition to the 6502 opcodes you have the following Pseudo opcodes:

```
.byt      value1,value2,value3, ...
.word     value1,value2, ...
```

```
.asc      "text1","text2", ...
.dsb      length ,fillbte
.fopt     value1, value2, ...
.text
.data
.bss
.zero
.align    value
*=
.(
.)
```

'.byt' and '.asc' are identical and save values to the memory (object file) bytewise. '.word' does the same with words (2 Bytes). '.dsb' fills a block with a given length with the value of fillbyte. If fillbyte is not given, zero is taken.

'*=' changes the programm counter. The programm counter is not saved to the file as no rewind is being done. Just the internal counter is reloaded. If a value is given when the assembler has been started in relocation mode ('-R command line option), the assembler goes into no-relocation mode, i.e assembles everything without creating relocation table entries. For '*=' without a value, the assembler switches back to relocation mode. The absolute code is 'embedded' in the text segment, so the text segment program counter is increased by the length of the absolute code.

'.(' opens a new 'block'. All labels in a block are local, i.e. are only visible from inside the block - including sub-blocks. An error is returned if a label is defined that is already defined 'above'. With '.)' the block is closed. You can have a stack of up to 16 blocks in each other (i.e. 16 times '.(' before the first '.)' will work, 17 not).

'.text', '.data', '.bss', '.zero' switch between the different segments. The text segment is where the code goes in. The data segment is where some initialized data goes in (it's actually like a second text segment). The data segment might be allocated separated from the text segment. The contents of the bss and the zero segment are not saved, just the labels are evaluated. Here goes the uninitialized data stuff. The zero segment allows allocation of zeropage space, bss is normal address space. These opcodes can be used in relative and absolute mode.

'.align' aligns the current segment to a byte boundary given by the value. Allowed values are 2, 4, and 256. When using relative mode, the align value is written to the file header, such that relocation keeps the alignment.

'.fopt' works like ".byte", but saves the bytes as a fileoption (see fileformat.txt). The length is computed automatically, so the first byte in the ".fopt" list of values should be the type. For example, the following line sets the filename for the object file.

```
.fopt 0, "filename", 0
```

A label is defined by not being an opcode:

```
label1 LDA #0              ; assignes the programm counter
label2 =1234               ; explicit definition
label3 label4 label5       ; implicit programm counter
label6 label7 = 3          ; label6 becomes the program counter, while
                           ; label7 is set to 3
label8:   sta label2       ; As ':' divides opcodes, this is also
                           ; working
```

You can use more than one label for definition, except for explicit definition. Labels are case sensitive. If a label is proceeded by a '+', this label is defined global. If a label is proceeded by a '&', this label is defined one level 'up' in the block hierachy, and you can use more than one '&'.

Redefinition of a label is possible by proceeding it with a dash '-'.

```
-sysmem   +=4  ; here you can use ==, +=, -=, *=, /=, &=, |=
-syszp    =123
```

Preprocessor

The preprocessor is very close to the one of the language C. So in addition to the ';'-comments you can also use C-like comments in '/*' and '*/'. Comments can be nested.

```
#include  "filename"     includes a file on exactly this position.
                        if the file is not found, it is searched using
                        XAINPUT.

#echo  comment          gives a comment to the error file.

#print expression       prints an expression to the error file (after
                        preprocessing and calculating)

#printdef DEFINED       prints the definition of a preprocessor define to
                        the error file.

#define DEF  text       defines 'DEF' by 'text'

#ifdef DEF              The source code from here to the following #endif
                        or #else is only assembled if 'DEF' is defined
                        with #define.

#else                   just else... (optionally)

#endif                  ends an #if-construct. This is a must to end #IF*

#ifndef  DEF            .... if DEF is not defined

#if expression          .... if expression is not zero

#iflused label          .... if a label has already been used

#ifldef label           .... if a label is already defined
```

#iflused and #ifldef work an labels, not on preprocessor defs! With these commands a kind of library is easily built:

```
#iflused  label
#ifldef   label
#echo     label already defined, not from library
#else
label     lda #0
          ....
#endif
#endif
```

You can have up to 15 #if* on stack before the first #endif

You can also use #define with functions, like in C.

```
#define mult(a,b)   ((a)*(b))
```

The preprocessor also allows continuation lines. I.e. lines that end with a '\' directly before the newline have the following line concatenated to it.

Utilities

There now are a few utilities that come with the assembler:

```
file65  : prints some information about an o65 file. Can compute the
          "-A" parameter for xa, to built the following file in a ROM.
reloc65 : relocates o65 files.
mkrom.sh: example shell (bash) script to show how to use the file65 utility
```

```
          to build a ROM image with several in the ROM runnable programs.
ld65    : a linker for o65 files. The given files are linked together and
          one o65 executable file is produced. All header options of all files
          are put in the new file. There must not be any undefined reference
          left, otherwise the output file is corrupt, because
          for now, ld65 cannot produce object files. But you get a warning.
```

Literature

- "Das Maschinensprachebuch zum Commodore 64"
  Lothar Englisch, Data Becker GmbH
- "Controller Products Data Book"
  Rockwell International, Semiconductor Products Division
- "Programmieren in C"
  Kernighan, Ritchie, Hanser Verlag