A68k

Charlie Gibbs

COLLABORATORS

	TITLE : A68k		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Charlie Gibbs	July 1, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

A68k

1.1 A68k

```
  A68k - a freely distributable assembler for the Amiga

      by Charlie Gibbs

         with special thanks to
    Brian R. Anderson and Jeff Lydiatt

                (Version 2.61 - January 11, 1990)
```

Note: This program is Freely Distributable, as opposed to Public Domain. Permission is given to freely distribute this program provided no fee is charged, and this documentation file is included with the program.

This assembler is based on Brian R. Anderson's 68000 cross-assembler published in Dr. Dobb's Journal, April through June 1986. I have converted it to produce AmigaDOS-format object modules, and have made many enhancements, such as macros and INCLUDE files.

My first step was to convert the original Modula-2 code into C. I did this for two reasons. First, I had access to a C compiler, but not a Modula-2 compiler. Second, I like C better anyway.

The executable code generator code (GetObjectCode and MergeModes) is essentially the same as in the original article, aside from its translation into C. I have almost completely rewritten the remainder of the code, however, in order to remove restrictions, add enhancements, and adapt it to the AmigaDOS environment. Since the only reference book available to me was the AmigaDOS Developer's Manual (Bantam, February 1986), this document describes the assembler in terms of that book.

RESTRICTIONS

Let's get these out of the way first:

o The verification file (-v) option is not supported. Diagnostic messages always appear on the console. They also appear in the listing file, however (see extensions below). You can produce an error file by redirecting console output to a file - the line number counter and final summary are displayed on stderr so you can still see what's happening.

o The file names in the INCLUDE directory list (-i) must be separated by commas. The list may not be enclosed in quotes.

o Labels assigned by EQUR and REG directives are case-sensitive.

o Strange things will happen if your source code (including INCLUDE files and macro expansions) exceeds 32,766 lines. Tough darts. Break up your source file. Can you actually read that monster? :-)

o The following directives are not supported, and will be flagged as invalid op-codes:

```
OFFSET
NOPAGE
LLEN
PLEN
NOOBJ
FAIL
FORMAT
NOFORMAT
MASK2
```

I feel that NOPAGE, LLEN, and PLEN should not be defined within a source module. It doesn't make sense to me to have to change your program just because you want to print your listings on different paper. The command-line switch "-p" (see below) can be used as a replacement for PLEN; setting it to a high value (like 32767) is a good substitute for NOPAGE. The effect of LLEN can be obtained by running the listing file through an appropriate filter.

EXTENSIONS

Now for the good stuff:

o Labels can be any length that will fit onto one source line (currently 127 characters maximum). Since labels are stored on the heap, the number of labels that can be processed is limited only by available memory.

o The first character of a label can be '@' if the next character is not numeric (this avoids confusion with octal constants). This provides compatibility with the Lattice C compiler.

o Since section data and user macro definitions are stored in the symbol table (see above), they too are limited only by available memory. (Actually, there is a hard-coded limit of 32,767 sections, but I doubt anyone will run into that one.)

o The only values a label cannot take are the register names - A68k can distinguish between the same name used as a label,

```
instruction name or directive, macro name, or section name.

    o Section and user macro names appear in the symbol table dump,
and will also be cross-referenced.  Their names can be the same
as any label (see above); they will be listed separately.

    o INCLUDEs and macro calls can be nested indefinitely, limited
only by available memory.  The message "Secondary heap
overflow - assembly terminated" will be displayed if memory
is exhausted.  You can increase the size of this heap using
the -w switch (see below).  Recursive macros are supported;
recursive INCLUDEs will, of course, result in a loop that
will be broken only when the heap overflows.

    o The EVEN directive forces alignment on a word (2-byte)
boundary.  It does the same thing as CNOP 0,2.
(This one is left over from the original code.)

    o Backward references to labels within the current CODE section
will be converted to PC-relative addressing with displacement
if this mode is legal for the instruction.  This feature is
disabled by the -n switch.

    o If a MOVEM instruction only specifies one register, it is
converted to the corresponding MOVE instruction.  Instructions
such as MOVEM D0-D0,label will not be converted, however.
This feature is disabled by the -n switch.

    o ADD, SUB, and MOVE instructions will be converted to ADDQ,
SUBQ, and MOVEQ respectively if possible.  Instructions coded
explicitly (e.g. ADDA or ADDI) will not be converted.  This
feature is disabled by the -n switch.

    o ADD, CMP, SUB, and MOVE to an address register are converted to
ADDA, CMPA, SUBA, and MOVEA respectively, unless (for ADD, SUB,
or MOVE) they have already been converted to quick form.

    o ADD, AND, CMP, EOR, OR, and SUB of an immediate value are
converted to ADDI, ANDI, CMPI, EORI, ORI, and SUBI respectively
(unless the address register or quick conversion above has
already been done).

    o If both operands of a CMP instruction are postincrement mode,
the instruction is converted to CMPM.

    o Operands of the form 0(An) will be treated as (An) except for
the MOVEP instruction, which always requires a displacement.
This feature is disabled by the -n switch.

    o The SECTION directive allows a third parameter.  This can be
specified as either CHIP or FAST (upper or lower case).  If
this parameter is present, the hunk will be written with the
MEMF_CHIP or MEMF_FAST bit set.  This allows you to produce
"pre-ATOMized" object modules.

    o The synonyms DATA and BSS are accepted for SECTION directives
starting data or BSS hunks.  The CHIP and FAST options (see
above) can also be used, e.g. BSS name,CHIP.

    o The following synonyms have been implemented for compatibility
with the Aztec assembler:
  CSEG is treated the same as CODE or SECTION name,CODE
  DSEG is treated the same as DATA or SECTION name,DATA
  PUBLIC is treated as either XDEF or XREF, depending on
      whether or not the symbol in question has been
      defined in the current source module.
      A single PUBLIC directive can name a mixture
      internally- and externally-defined symbols.

    o The ability to produce Motorola S-records is retained from the
original code.  The -s switch causes the assembler to produce
S-format instead of AmigaDOS format.  Relocatable code cannot
be produced in this format.

    o Error messages consist of three parts.
    The position of the offending line is given as a line
number within the current module.  If the line is within a
macro expansion or INCLUDE file, the position of the macro
call or INCLUDE statement in the outer module is given as
well.  This process is repeated until the outermost source
module is reached.
    Next, the offending source line itself is listed.
    Finally, the errors for that line are displayed.  A flag
(^) is placed under the column where the error was detected.

    o Named local labels are supported.  These work the same as the
local labels supported by the Metacomco assembler (nnn$) but
are formed in the same manner as normal labels, except that
they must be preceded by a backslash (\).

    o The following synonyms have been implemented for compatibility
with the Assempro assembler:
  ENDIF is treated the same as ENDC
  = is treated the same as EQU
  | is treated the same as ! (logical OR)

    o Quotation marks (") can be used as string delimiters
as well as apostrophes ('). Any given string must begin
and end with the same delimiter.  This allows such statements
as the following:
  MOVEQ "'",D0
  DC.B  "This is Charlie's assembler."
Note that you can still define an apostrophe within a string
delimited by apostrophes if you double it, e.g.
  MOVEQ '''',D0
  DC.B  'This is Charlie''s assembler.'

    o If any errors are found in the assembly, the object code file
will be scratched, unless you include the -k (keep) flag on
the command line.

    o The symbols .A68K, .a68k, .a68K, and .A68k are automatically
defined as SET symbols having absolute values of 1.
This enables a source program to determine whether it is
```

being assembled by this assembler, and is effectively
insensitive as to whether or not it is checked in upper case.

o A zeroth positional macro parameter (\0) is supported. It
is replaced by the length of the macro call (B, W, or L,
defaulting to W). For instance, given the macro:

```
  moov  MACRO
    move.\0 \1,\2
    ENDM
```

the macro call

```
    moov.l  d0,d1
```

would be expanded as

```
    move.l  d0,d1
```

o If an INCLUDE file doesn't generate any code and no listing
file is required (including suppression of the listing using
NOLIST), it won't be read again in pass 2. The statement
numbers will be bumped to keep in proper alignment. This
can really speed up assemblies that INCLUDE lots of EQUates.

o The ORG directive is supported. It works like RORG, except
that it takes the actual address to be jumped to, rather
than an offset from the start of the current section.
The given address must be in the current section.
As far as A68k is concerned, the only real difference
between ORG and RORG is that the ORG value must be
relocatable, while the RORG value must be absolute.

o Branch (Bcc, including BRA and BSR) instructions will be
converted to short form if possible. Shortening a branch
may bring other branches within range of shortening - this
can set up a ripple effect, and A68k may not catch all
branches that could theoretically be optimized. Any branches
which A68k misses (there shouldn't be too many under normal
circumstances) can be displayed by specifying the -f switch
(see below). Branch optimization is disabled by the -n switch.

THE SMALL CODE / SMALL DATA MODEL

Version 2.4 implements a rudimentary small code/data model.
It consists of converting any data reference to one of the following
three addressing modes:

```
address register indirect with displacement using a
  specified address register, defaulting to A4
  (for references to the DATA or BSS section)
program counter indirect with displacement
  (for references to the CODE section)
absolute word
  (for absolute and 16-bit relocatable values)
```

These conversions do not take place unless a NEAR directive is encountered. The NEAR directive can take one operand, which must be either an address register or a symbol which has been equated (using EQUR) to an address register. Register A7 (SP) may not be used. If no register is given, A4 is assumed.

Conversion is done for all operands until a FAR directive is encountered. NEAR and FAR directives can occur any number of times, enabling conversion to be turned on and off at will.

Backward references which cannot be converted (e.g. external labels declared as XREF) will remain as absolute long addressing. All forward references are assumed to be convertible, since during pass 1 A68k has no way of telling whether conversion is possible. If conversion turns out to be impossible, invalid object code will be generated - an error message ("Invalid forward reference") will indicate when this occurs.

Although the small code/data model can greatly reduce the size of assembled programs, several restrictions apply:

o Small code and small data models are active simultaneously. You can't have one without the other, since during pass 1 A68k doesn't know whether forward references are to CODE or to DATA/BSS.

o Programs can consist of a maximum of two sections, one CODE, the other DATA or BSS. If you try to define a third section, the message "Too many SECTIONs" will be displayed. The NEAR directive is active only within the CODE section.

o While the NEAR directive is active, external labels (XREF) must be declared before they are used, CODE section references must be with 32K of the current position (i.e. expressible as PC-relative), and DATA/BSS section references must be in the first 64K of the DATA/BSS section (i.e. expressible as address register indirect with displacement). Any instructions which do not satisfy these requirements cannot be detected in pass 1, so A68k has no choice but to display an error message in pass 2 ("Invalid forward reference") which in this case indicates that invalid code has been generated. To properly assemble such instructions, you can temporarily disable conversion with a FAR directive, then resume afterwards with another NEAR directive.

o Conversion cannot be done for references between modules. All external references must be left as absolute long.

o A68k assumes that the base register (normally A4) points to the start of the DATA/BSS section plus 32768 bytes. The register must be preloaded with this value before executing any code converted by the NEAR directive. One way to do this is to code the instruction that loads the register prior to the NEAR directive. Another way is to use a MOVE.L with immediate mode, which is never converted. Here are examples of the two methods:

```
    LEA data+32768,a4   NEAR
    NEAR  ;defaults to A4   MOVE.L  #data+32768,a4
      <remainder of code>       <remainder of code>
    BSS       BSS
  data:       data:
      <data areas>        <data areas>
    END       END
```

I'll be the first to admit that this is a very crude and ugly implementation. I hope to improve it in future versions.

FILES

A68k uses the following files:

- o The source code file - this file contains the program to be assembled. This file is an ASCII text file whose last line must be an END statement.

- o The object code file - this file is created by A68k, replacing any previous version which may exist. If any errors are encountered during the assembly, this file will be scratched, unless the -k (keep) switch is specified (see below). Although this file is normally written in AmigaDOS format, the -s switch (see below) will cause it to be written in Motorola S-record format instead.

- o The listing file - this file is optionally created by A68k and contains a listing complete with page headings (including form feeds), generated object code, and error messages if any. It is suitable for feeding to a printer.

- o An equate file - this file is optionally created by A68k and consists of a leading comment line followed by EQU statements, one for each symbol encountered by A68k whose value is absolute. This file is only created if the -e command-line switch is specified (see below).

- o A header file - if requested, this file is read by A68k immediately prior to the source code file. It treated exactly as if it were requested by an INCLUDE statement at the front of the source file, but is selected only if the -h command-line switch is specified (see below).

- o Include files are selected by INCLUDE directives within the source file, and are inserted in the source code in place of the INCLUDE directive itself. A68k first searches the current directory for INCLUDE files; the -i command-line switch (see below) specifies additional directories which can be searched.

FILE NAMES

The names of the above files can be explicitly specified.

However, A68k will generate default file names in the following cases:

```
      o If the -o switch is omitted, a default name will be assigned
  to the object code file.

      o If the -e switch is specified with no file name, a default
  name will be assigned to the equate file.

      o If the -l or -x switch is specified with no file name, a
  default name will be assigned to the listing file.
```

A default name is generated by deriving a stem name from the source code file name, and appending .o for an object code file name (.s if the -s switch is specified to produce Motorola S-records), .equ for an equate file name, or .lst for a listing file name. The stem name consists of all characters of the source file name up to the last period (or the entire source file name if it contains no period). Here are some examples:

```
                     Default names
      --------------------------------------------
  Source file Object file Equate file Listing file
  ----------- ----------- ----------- ------------
  myprog.asm  myprog.o  myprog.equ  myprog.lst
  myprog    myprog.o  myprog.equ  myprog.lst
  new.prog.asm  new.prog.o  new.prog.equ  new.prog.lst
```

HOW TO USE A68k

The command-line syntax to run the assembler is as follows:

```
  a68k <source file name>
       [<object file name>]
       [<listing file name>]
    [-d[[!]<prefix>]]
    [-e[<equate file name>]]
    [-f]
    [-h<header file name>]
    [-i<INCLUDE directory list>]
    [-k]
    [-l[<listing file name>]]
    [-n]
    [-o<object file name>]
    [-p<page depth>]
    [-q[<quiet interval>]]
    [-s]
    [-t]
    [-w[<hash table size>][,<secondary heap size>]]
    [-x[<listing file name>]]
    [-y]
    [-z[<debug start line>][,<debug end line>]]
```

These options can be given in any order. Any parameter which is not a switch (denoted by a leading hyphen) is assumed to be a file name; up to three file names (assumed to be source, object, and listing file names respectively) can be given. A source file name is always required. If a switch is being given a value, that value must immediately follow the switch letter with no intervening spaces. For instance, to specify a page depth of 40 lines, the specification "-p40" should be used; "-p 40" will be rejected.

Switches perform the following actions:

-d causes symbol table entries (hunk_symbol) to be written to the object module for the use of symbolic debuggers. If the switch is followed by a string of characters, only those symbols beginning with that prefix string will be written. This can be used to suppress internal symbols generated by compilers. If the first character is an exclamation mark (!), only symbols which do NOT begin with the following characters are written out.

Here are some examples:

```
  -d  writes all symbols
  -dabc writes only symbols beginning with "abc"
  -d!x  writes symbols which do not begin with "x"
```

-e causes an equate file (see above) to be produced. A file name can be specified; otherwise a default name will be used.

-f causes any branches (Bcc, BRA, BSR) that could be converted to short form to be flagged. A68k will convert as many branches as possible to short form (unless the -n switch is is specified), but certain combinations of instructions may set up a ripple effect where shortening one branch brings another one into range. This switch will cause A68k to flag any branches that it may have missed; during pass 2 it is possible to tell this, although during pass 1 it might not be. If the -n switch (see below) is specified along with this switch (suppressing all optimization), no branches will be shortened, but all branches which could be shortened will be flagged.

-h causes a header file to be read prior to the source code file. A file name must be given. The action is the same as if the first statement of the source file were an INCLUDE statement naming the header file. To find the header file, the same directories will be searched as for INCLUDE files (see the -i switch below).

-i specifies directories to be searched for INCLUDE files in addition to the current directory. Several names, separated by commas, may be specified. No embedded blanks are allowed. For example, the specification

```
  -imylib,df1:another.lib
```

will cause INCLUDE files to be searched for first in the current directory, then in "mylib", then in "df1:another.lib".

-k causes the object file to be kept even if any errors were found. Otherwise, it will be scratched if any errors occur.

-l causes a listing file to be produced. If you want the listing file to include a symbol table dump and cross-reference, use the -x switch instead (see below).

-n causes all object code optimization (see above) to be disabled.

-o allows the default name for the object code file (see above) to be overridden.

-p causes the page depth to be set to the specified value. This takes the place of the PLEN directive in the Metacomco assembler. Page depth defaults to 60 lines (-p60).

-q changes the interval at which A68k displays the line number it has reached in its progress through the assembly. The default is to display every 10 lines (-q10). Specifying larger values reduces console I/O, making assemblies run slightly faster.

If you specify a negative number (e.g. -q-10), line numbers will be displayed at an interval equal to the absolute value of the specified number, but will be given as positions within the current module (source, macro, or INCLUDE) rather than as a total statement count - the module name will also be displayed.

A special case is the value zero (-q0 or just -q) - this will cause all console output, except for error messages, to be suppressed.

-s causes the object file to be written in Motorola S-record format, rather than AmigaDOS format. The default name for an S-record file ends with ".s" rather than ".o"; this can still be overridden with the -o switch, though.

-t allows tabs in the source file to be passed through to the listing file, rather than being expanded. In addition, tabs will be generated in the listing file to skip from the object code to the source statement, etc. This can greatly reduce the size of the listing file, as well as making it quicker to produce. Do not use this option if you will be displaying or listing the list file on a device which does not assume a tab stop at every 8th position.

-w specifies the sizes of fixed memory areas that A68k allocates for its own use. You should normally never have to specify this switch, but it may be useful for tuning.

The first parameter gives the number of entries that the hash table (used for searching the symbol table) will contain. The default value of 2047 should be enough for all but the very largest programs. The assembly will not fail if this value is too small, but may slow down if too many long hash chains must be searched. The hashing statistics displayed by the -y switch (see below) can be used to tune this parameter. I’ve heard that you should really specify a prime number for this parameter, but I haven’t gone into hashing theory enough to know whether it's actually necessary.

The second parameter of the -w switch specifies the size (in bytes) of the secondary heap, which is used to store nested macro and INCLUDE file information (see below). It defaults to 1024, which should be enough unless you use very deeply nested macros and/or INCLUDE files with long path names.

You can specify either or both parameters. For example:

```
  -w4093       secondary heap size remains at 1024 bytes
  -w,2000      hash table size remains at 2047 entries
  -w4093,2000  increases the size of both areas
```

If you're really tight for memory, and are assembling small modules, you can use this switch to shrink these areas below their default sizes. At the end of an assembly, a message will be displayed giving the sizes actually used, in the form of the -w command you would have to enter to allocate that much space. This is primarily useful to see how much secondary heap space was used.

NOTE: All other memory used by A68k (e.g. the actual symbol table) is allocated as required (currently in 8K chunks).

-x works the same as -l (see above), except that a symbol table dump, including cross-reference information, will be added to the end of the listing file.

-y causes hashing statistics to be displayed at the end of the assembly. First the number of symbols in the table is given, followed by a summary of hash chains by length. Chains with length zero denote unused hash table entries. Ideally (i.e. if there were no collisions) there should be as many chains with length 1 as there are symbols, and there should be no chains of length 2 or greater. I added this option to help me tune my hashing algorithm, but you can also use it to see whether you should allocate a larger hash table (using the first parameter of the -w switch, see above).

-z was provided to help debug A68k itself. It causes A68k to list a range of source lines, complete with line number and current location counter value, during both passes. Lines are listed immediately after they have been read from the source file, before any processing occurs.

Here are some examples of the -z switch:

```
  -z    lists all source lines
  -z100,200 lists lines 100 through 200
  -z100   lists all lines starting at 100
  -z,100    lists the first 100 lines
```

TECHNICAL INFORMATION

The actual symbol table entries (pointed to by the hash table, colliding entries are linked together) are stored in 8K chunks which are allocated as required. The first entry of each chunk is reserved as a link to the next chunk (or NULL in the last chunk) - this makes it easy to find all the chunks to free them when we're finished. All symbol table entries are stored in pass 1. During pass 2, cross-reference table entries are built in the same group of chunks, immediately following the last symbol table entry. Additional chunks will continue to be linked in if necessary.

Symbol names and macro text are stored in another series of linked chunks. These chunks consist of a link pointer followed by strings (terminated by nulls) laid end to end. Symbols are independent entries, linked from the corresponding symbol table entry. Macros are stored as consecutive strings, one per line - the end of the macro is indicated by an ENDM statement. If a macro spans two chunks, the last line in the original chunk is followed by a newline character to indicate that the macro is continued in the next chunk.

Relocation information is built during pass 2 in yet another series of linked chunks. If more than one chunk is needed to hold one section's relocation information, all additional chunks are released at the end of the section.

The secondary heap is built from both ends, and it grows and shrinks according to how many macros and INCLUDE files are currently open. At all times there will be at least one entry on the heap, for the original source code file. The expression parser also uses the secondary heap to store its working stacks - this space is freed as soon as an expression has been evaluated.
The bottom of the heap holds the names of the source code file and any macro or INCLUDE files that are currently open. The full path is given. A null string is stored for user macros. Macro arguments are stored by additional strings, one for each argument in the macro call line. All strings are stored in minimum space, similar to the labels and user macro text on the primary heap. File names are pointed to by the fixed table entries (see below) - macro arguments are accessed by stepping past the macro name to the desired argument, unless NARG would be exceeded.
The fixed portion of the heap is built down from the top. Each entry occupies 16 bytes. Enough information is stored to return to the proper position in the outer file once the current macro or INCLUDE file has been completely processed.
The diagram below illustrates the layout of the secondary heap.

```
  Heap2 + maxheap2 ----------->  ______________________________
             |          |
             |   Input file table    |
  struct InFCtl *InF ---------> |______________________________|
             |          |
             |   Parser operator stack    |
  struct OpStack *Ops --------> |______________________________|
             |          |
             |   (unused space)     |
  struct TermStack *Term -----> |______________________________|
             |          |
             |   Parser term stack    |
  char *NextFNS --------------> |______________________________|
             |         |
             |   Input file name stack   |
  char *Heap2 ----------------> |___________________________|
```

     The "high-water mark" for NextFNS is stored in char *High2,
and the "low-water mark" (to stretch a metaphor) for InF is stored
in struct InFCtl *LowInF. These figures are used only to determine
the maximum heap usage.

AND FINALLY...

     Please send me any bug reports, flames, etc. I can be reached
on Mind Link (604/533-2312), at any meeting of the Commodore
Computer Club / Panorama (PAcific NORthwest AMiga Association),
or via Jeff Lydiatt or Larry Phillips. I don't have the time
or money to live on Compuserve or BIX, but my Usenet address is
Charlie_Gibbs@mindlink.UUCP (...uunet!van-bc!rsoft!mindlink!a218).

Charlie Gibbs
2121 Rindall Avenue
Port Coquitlam, B.C.
Canada
V3C 1T9