Cowgol, C and Z80 assembler development environment

hosted on Z80 computers running CP/M 2.2

User’s manual

Szilagyi Ladislau, July 2025

Introduction

This development environment for Z80 computers running CP/M 2.2 allows the construction of CP/M applications, starting from source files written in the Cowgol, C and Z80 assembler programming languages.

Components taken from the HiTech C compiler v3.09 are used, plus the Cowgol compiler built by David Given.

The object code format is compatible with the HiTech C compiler's object code format, thus facilitating the possibility of mixing code written in Cowgol, C and Z80 assembler.

Operating Details

Under CP/M 2.2, a single command will compile, assemble and link into a CP/M 2.2 executable file several Cowgol and optionally, C and Z80 assembler source files.

The syntax of the command is as follows:

>cowgol [options] files

The options are zero or more options, each consisting of a dash ('-'), a single key letter, and possibly an argument, following the key letter with no intervening space.

The files are one or more Cowgol, C source files, Z80 assembler source files, or Cowgol, C or assembler object files.

The command will, as determined by the specified options, compile any Cowgol, C source files given, assemble them into object code unless requested otherwise, assemble any Z80 assembler source files specified, and then link the results of the assemblies with any object or library files specified.

Upper and lower case can be used to specify the options or file names, the command does not distinguish between cases.

The options recognized are as follows:

-C The files will be only compiled/assembled.

-Mfile Builds the "file.map" memory map file for the executable.

-Lfile Adds the file "file.lib" to the list of files used as input by the LINK linker.

-O Z80 assembler code optimizations will be performed

-Tfile Build a table of symbols named "file.sym", to be used when debugging the executable

-S Type to the console a map of the Cowgol subroutines and variables used in these subroutines, with their exact addresses. This is very valuable when debugging the executable.

-X Reserve a larger stack (1KB) for the executable (the usual stack size is 128 bytes). This might be important if re-entrant C code is mixed into the project.

If the option -C is not specified, the files will be first compiled/assembled, then linked into a CP/M executable (named after the first Cowgol file name in the list of the source files).

More than one source file may be specified (with extensions: .cow = cowgol source file, .c = C source file, .as = assembler source file)

Also, C or AS object files (with extension .obj) and Cowgol object files (with extension .coo) may be specified.

The first source file must be a Cowgol source file. If more Cowgol source files are used, the first one will give the name of the executable being built, but the last one must contain the 'main' routine. C and assembler routines may be called from Cowgol source files.

Examples of compilation commands

1. >cowgol -c hexdump.cow

The Cowgol “hexdump.cow” source file will be compiled to the Cowgol object file “hexdump.coo”.

2. >cowgol testas.cow rand.as

The Cowgol “testas.cow” source file will be compiled, the Z80 assembler “rand.as” source file will be assembled, then the resulting object code files will be linked to the “testas.com” executable.

3. >cowgol -Llibc -O dynmsort.cow merges.c rand.as

The Cowgol “dynmsort.cow” source file will be compiled, the C source file “merges.c” will be compiled, the Z80 assembler file “rand.as” will be assembled, all with code optimization, then the resulting object code files and parts of the “libc.lib” C runtime library will be linked to the “dynmsort.com” executable.

4. >cowgol -o -s -tsymbols alloc.coo testmem.cow xrnd.obj

The Cowgol source file “testmem.cow” will be compiled, with code optimization, then the resulting object code will be linked with the “alloc.coo” and “xrnd.obj” to the “testmem.com” executable. The Cowgol subroutines and variables will be typed to the console, and a file named “symbols.sym” will be created, to facilitate the debugging of the executable.

5. >cowgol -o misc.coo string.coo seqfile.coo startrek.cow

The Cowgol source file “startrek.cow” will be compiled, with code optimization, then linked to the “misc.coo”, “string.coo” and “seqfile.coo” to the “strartrek.com” executable.

6. >cowgol -o misc.coo string.coo ranfile.coo advent.cow advtrav.cow advmain.cow

The Cowgol source files “advent.cow”, “advtrav.cow” and “advmain.cow” will be compiled, with code optimization, then linked to the “misc.coo”, “string.coo” and “ranfile.coo” to the “advent.com” executable.

Executables

The following executables are needed:

- $EXEC.COM , the "batch processor" from the HiTech's C compiler, who launches all the subsequent executables from the Cowgol toolchain
- COWGOL.COM (a modified variant of the HiTech's C.COM), the component that interprets the command line and feeds into $EXEC run requests for the subsequent executables from the Cowgol toolchain
- COWFE.COM , the "cowgol front end", who parses the source file, part of the Cowgol compiler, optimized
- COWBE.COM , the "cowgol back end", who builds the "cowgol object file", part of the Cowgol compiler, optimized
- COWLINK.COM , the "cowgol linker", who binds all the "cowgol object files" and outputs a Z80 assembler file, part of the Cowgol compiler, optimized
- COWFIX.COM , interface to Z80AS , who performs also code optimizations
- Z80AS.COM, the Z80 assembler (see https://github.com/Laci1953/Z80AS), who assembles the output of Cowfix and any Z80 assembler source file included in the command line, producing HiTech compatible object files
- LINK.COM , the HiTech's linker, who builds the final executable, using as input the object files and, if requested, the library file and producing the final executable
- CPP.COM , the HiTech's C pre-processor (needed only when C source files will be compiled), modified to accept // - style comments
- P1.COM , the HiTech's C compiler pass 1 (needed only when C source files will be compiled)
- CGEN.COM , the HiTech's C compiler pass 2 (needed only when C source files will be compiled)
- OPTIM.COM , the HiTech's C compiler optimizer (needed only when C source files will be compiled)
- LIBR.COM , the HiTech's librarian (needed only to build and manage .lib libraries)

Also, the Cowgol object file "cowgol.coo", containing basic Cowgol run-time support routines must be present on the disk.

Linkers

The author of the Cowgol language and compiler, David Given, designed the compiler as a two-step procedure: the “front-end” COWFE, that parses the Cowgol source file and builds an intermediate format, which is then processed by the “back-end” COWBE, to produce the “cowgol object code” file (with extension .COO).

Then, the “linker” COWLINK will read all the necessary “cowgol object files”, adding the system cowgol object file “cowgol.coo” to produce a Z80 assembler output.

This file is then assembled by Z80AS to produce an “object code” file (with extension .OBJ).

Then, the HiTech’s LINK will link all the necessary object codes (produced by the Cowgol compiler, or C compiler, or Z80AS assembler) , possibly adding routines from the specified libraries, into a CP/M executable (with extension .COM).

The sequence of execution of all these components is:

Cowgol source file >> COWFE >> COWBE >> COWLINK >> COWFIX >> Z80AS >> .OBJ file
C source file >> CPP >> P1 >> CGEN >> OPTIM >> Z80AS >> .OBJ file
Z80 assembler source file >> Z80AS >> .OBJ file
then *.OBJ object files & .LIB library files >> LINK >> .COM file*

Variables map (compile option -S)

Example of building a cowgol application variables map and using this map to debug the application.

We use the source file "hexdump.cow".

Here is the cowgol source code of the "print_hex_i8" subroutine, used in "hexdump.cow":

```
sub print_hex_i8(value: uint8) is
   var i: uint8 := 2;
   loop
      var digit := value >> 4;
         if digit < 10 then
         digit := digit + '0';
         else
            digit := digit + ('a' - 10);
         end if;
         print_char(digit);
         value := value << 4;
         i := i - 1;
         if i == 0 then
           break;
         end if;
                  end loop;
end sub;
```

Here is a fragment of the Cowgol command log , when building "hexdump.com" :

(executed on a 512 KB RAM machine)
(the -S option is used to build the cowgol variables map)
(the -O option is used to optimize the assembler code produced by Cowlink...)
(the -T option is used by the LINK linker to build the s.sym file containing a table of the variables and their absolute values)

```
>cowgol -o -s -ts hexdump.cow

COWGOL COMPILER (CP/M-80) V2.0
Copyright (C) David Given
0:COWFE     HEXDUMP.COW $CTMP1.$$$ -S
COWFE: 19kB free main memory
448kB free extended memory
> HEXDUMP.COW
          > stdcow.coh
...
< HEXDUMP.COW
done: 16kB free main memory
425kB free extended memory
0:COWBE $CTMP1.$$$ HEXDUMP.COO
COWBE: 23kB free
__main
...
Hexdump
done: 16kB free
```

```
ERA $CTMP1.$$$
ERA $CTMP2.$$$
0:COWLINK COWGOL.COO HEXDUMP.COO -o $CTMP1.$$$ -S
COWLINK: 44kB free
Adding input file: COWGOL.COO
Adding input file: HEXDUMP.COO
Analysing...
Workspace sizes:
#0: 231 bytes
Subroutines and Variables map

Subroutine Hexdump variables list:

k       @ ws+00d4H     (212)
c       @ ws+00d3H     (211)
j       @ ws+00d1H     (209)
i       @ ws+00d0H     (208)
buf     @ ws+00c0H     (192)
...
Subroutine print_hex_i8 variables list:

digit   @ ws+00dbH     (219)
i       @ ws+00daH     (218)
value   @ ws+00d9H     (217)
...
Creating output file: $CTMP1.$$$
Copying from input file: COWGOL.COO
Copying from input file: HEXDUMP.COO
done: 38kB free
ERA HEXDUMP.COO
0:COWFIX $CTMP1.$$$ $CTMP2.$$$ -O
0:Z80AS -J -N -OHEXDUMP.OBJ $CTMP2.$$$
Z80AS Macro-Assembler V4.8

Errors: 0

Jump optimizations done: 58
Finished.
0:LINK -Z -Ptext=100H,data,bss -C100H -DS.SYM -OHEXDUMP.COM HEXDUMP.OBJ
ERA $SUBSYMS.$$$
ERA $CTMP1.$$$
ERA $CTMP2.$$$
ERA $$EXEC.$$$
>
```

Here is a relevant fragment of the assembler file generated by Cowlink:

```
        ; print_hex_i8 workspace at ws+217 length ws+3
f45_print_hex_i8:
            ld (ws+217), a
            ld a,2
            ld (ws+218), a
c1c_002b:
            ld a, (ws+217)
```

```
        ld b,4
        call f2__lsr1
        ld (ws+219), a
        cp 10
        jp nc, c1c_0031
....
```

Here is a relevant part of the symbols map file generated by LINK:

```
>type s.sym
0782 _exit
...
0783 ws
...
01A0 f45_print_hex_i8
...
>
```

(notice the value of ws is 0783H, and 01A0 is the address of print_hex_i8 subroutine)

Therefore, at last, we can now calculate the exact addresses of the variables used in print_hex_i8:

digit @ ws+00dbH (219) = 0783H + 00DBH = 085EH i @ ws+00daH (218) = 0783H + 00DAH = 085DH value @ ws+00d9H (217) = 0783H + 00D9H = 085CH

Let's now debug hexdump.com: (we list the print_hex_i8 routine)

```
>zsid hexdump.com

ZSID VERS 1.4
NEXT  PC  END
0800 0100 BFFF
#L1A0
01A0  LD   (085C),A
01A3  LD   A,02
01A5  LD   (085D),A
01A8  LD   A,(085C)
01AB  LD   B,04
01AD  CALL 0107
01B0  LD   (085E),A
01B3  CP   0A
01B5  JR   NC,0A
01B7  LD   A,(085E)
01BA  ADD  A,30
```

,and we now know that the first line:

```
01A0 LD (085C),A
```

stores the parameter "value" (passed in register A) to the variable placed at the address 085CH.

This is valuable when debugging a Cowgol application.

In the case when the Cowgol compiler command line specifies multiple Cowgol source files, the variable map will contain data related to all the source files included.

Variables map - Implementation details:

Cowfe, the Cowgol compiler's pass 1, when the option -S is used in the Cowgol command line, writes to a temporary file SUBSYMS.$$$ (erased at the end of the compiling batch process) a list of the subroutines (and their variables) for each source file. Because Cowfe handles each source file separately, if the Cowgol command line contains more than one Cowgol source file, the temporary file is created when processing the first source file, then it is "oppened for append" for the subsequent source files. The temporary file will be read by Cowlink to build the variables map, containing the exact addresses of the variables.

Here is a fragment of the dump of the SUBSYMS.$$ file:

```
>dumpx $subsyms.$$$
0000 : 5F 5F 6D 61 69 6E 00 4C 4F 4D 45 4D 00 00 00 5F : __main.LOMEM..._
0010 : 5F 6D 61 69 6E 00 48 49 4D 45 4D 00 02 00 41 6C : _main.HIMEM...Al
0020 : 69 67 6E 55 70 00 69 6E 00 00 00 41 6C 69 67 6E : ignUp.in...Align
0030 : 55 70 00 6F 75 74 00 02 00 67 65 74 5F 63 68 61 : Up.out...get_cha
0040 : 72 00 63 00 00 00 70 72 69 6E 74 5F 63 68 61 72 : r.c...print_char
...
```

Why COWFIX?

COWFIX reads the output produced by COWLINK (a Z80 assembler file) and performs some code optimization to output the final Z80 assembler file which will be assembled by Z80AS.

Examples of code optimization:

1. *Using conditional return*

```
; jp nz, c01_000c
; ret
;c01_000c:
   ret z
```

means that the 3 statements "commented-out" were replaced by "ret z".

2. *Using jp instead of call + ret*

```
; print_nl workspace at ws+1432 length ws+0
f10_print_nl:
   ld a,10
   jp   f7_print_char ;  call f7_print_char
;end_f10_print_nl:
; ret
```

means that the lines

```
call f7_print_char
ret
```

were substituted by the line "jp f7_print_char"

3. *Optimizing jump-to-jump*

```
    jp nz, c01_0014 ; c01_001f
...
c01_001f:
   jp c01_0014
```

means that the jump-to-jump c01_001f --> c01_0014

```
jp nz, c01_001f
...
c01_001f:
  jp c01_0014
```

was solved by inserting the correct final jump address " jp nz, c01_0014"

4. *Dropping useless store statements*

call f11_UIToA

; ld (ws+1450), hl

; ld (ws+1452), hl

means that the two "ld" instructions were "commented out", because the addresses ws+1450 and ws+1452 are never accessed for read.

5. *Dropping useless load statements*

...

sbc hl,de

; ld (ws+1414), hl

; ld hl, (ws+1414)

ret

means that the two "ld" were commented out, they are useless, because the routine just wants to return HL...

6. *Dropping 'dead' labels*

...

jp nc, c01_00c3

;c01_00c2:

ld de,4

...

means that "c01_00c2" was commented out, because it's a "dead" label (no jumps or calls are directed to this label)

The overall gain is significant:

- smaller size
- faster code

Includes and Libraries

The include files (with extension .COH) can be “included” to a Cowgol source file by using the Cowgol *include* statement.

1. The original include files (and their content as subroutines) designed by David given are:

cowgol.coh (Exit, ExitWithError, AlignUp, get_char, print_char, MemSet)

common.coh (print, print_nl, UIToA, ItoA, print_i32, print_i16, print_i8, print_hex_i8, print_hex_i16, print_hex_i32, AtoI, MemZero)

argv.coh (ArgvInit, ArgvNext)

malloc.coh (Alloc, Free, StrDup)

strings.coh (StrCmp, ToLower, StrICmp, StrLen, CopyString, MemCopy)

file.coh (FCBOpenIn, FCBOpenOut, FCBClose, FCBSeek, FCBPos, FCBExt, FCBGetChar, FCBPutChar)

commfile.coh (FCBPutString, FCBGetBlock, FCBPutBlock)

These original include files contain raw Cowgol source code.

2. I prefered to add new include files, containing only subroutine declarations:

misc.coh, containing:

```
@decl sub exit() @extern(“exit”);
@decl sub get_char(): (c: uint8) @extern(“get_char”);
@decl sub get_line(p: [uint8]) @extern(“get_line”);
@decl sub print_char(c: uint8) @extern(“print_char”);
@decl sub print(ptr: [uint8]) @extern(“print”);
@decl sub print_nl() @extern(“print_nl”);
@decl sub print_hex_i8(char: uint8) @extern(“print_hex_i8”);
@decl sub print_hex_i16(word: uint16) @extern(“print_hex_i16”);
@decl sub print_hex_i32(dword: uint32) @extern(“print_hex_i32”);
@decl sub print_i8(v: int8) @extern(“print_i8”);
@decl sub print_i16(v: int16) @extern(“print_i16”);
@decl sub isdigit(ch: uint8): (ret: uint8) @extern(“isdigit”);
@decl sub itoa(i: int16): (pbuf: [uint8]) @extern(“itoa”);
@decl sub uitoa(i: uint16): (pbuf: [uint8]) @extern(“uitoa”);
@decl sub ltoa(i: int32): (pbuf: [uint8]) @extern(“ltoa”);
@decl sub atoi(p: [uint8]): (ret: int16) @extern(“atoi”);
@decl sub atol(p: [uint8]): (ret: int32) @extern(“atol”);
@decl sub atofixed(p: [uint8]): (ret: uint16) @extern(“atofixed”);
# fdigits: number of digits in fractional part
@decl sub fixedtoa(f: uint16, fdigits: uint8): (ret: [uint8]) @extern(“fixedtoa”);
@decl sub memcpy(dest: [uint8], src: [uint8], size: uint16): (ret: [uint8])
@extern(“memcpy”);
@decl sub memset(dest: [uint8], char: uint8, size: uint16): (ret: [uint8])
@extern(“memset”);
@decl sub xrnd()  😵ret: uint16) @extern(“xrnd”);
@decl sub xrndseed() @extern(“xrndseed”);
```

```
@decl sub ArgvInit() @extern("ArgvInit");
@decl sub ArgvNext(): (arg: [uint8]) extern("ArgvNext");
```

string.coh, containing:

```
@decl sub strlen(str: [uint8]): (len: uint16) @extern("strlen");
@decl sub strcpy(dest: [uint8], src: [uint8]): (ret: [uint8]) @extern("strcpy");
@decl sub strcat(dest: [uint8], src: [uint8]): (ret: [uint8]) @extern("strcat");
@decl sub strcmp(str1: [uint8], str2: [uint8]): (ret: int8) @extern("strcmp");
@decl sub strcasecmp(str1: [uint8], str2: [uint8]): (ret: int8) @extern("strcasecmp");
@decl sub strstr(str: [uint8], tosearch: [uint8]): (ret: [uint8]) @extern("strstr");
@decl sub strchr(str: [uint8], tosearch: uint8): (ret: [uint8]) @extern("strchr");
@decl sub tolower(char: uint8): (ret: uint8) @extern("tolower");
```

ranfile.coh, containing:

```
@decl sub FCBOpenIn(fcb: [FCB], filename: [uint8]): (errno: uint8)
@extern("FCBOpenIn");
@decl sub FCBOpenUp(fcb: [FCB], filename: [uint8]): (errno: uint8)
@extern("FCBOpenUp");
@decl sub FCBOpenOut(fcb: [FCB], filename: [uint8]): (errno: uint8)
@extern("FCBOpenOut");
@decl sub FCBClose(fcb: [FCB]): (errno: uint8) @extern("FCBClose");
@decl sub FCBSeek(fcb: [FCB], pos: uint32) @extern("FCBSeek");
@decl sub FCBPos(fcb: [FCB]): (pos: uint32) @extern("FCBPos");
@decl sub FCBExt(fcb: [FCB]): (len: uint32) @extern("FCBExt");
@decl sub FCBGetChar(fcb: [FCB]): (c: uint8) @extern("FCBGetChar");
@decl sub FCBPutChar(fcb: [FCB], c: uint8) @extern("FCBPutChar");
```

seqfile.coh, containing:

```
# CP/M Z80 sequential files
#
# supports two kind of files: text files (0x1A is EOF) & binary files
#
# FCBOpenIn : opens specified existing file for read (type: IO_TEXT or IO_BIN)
# FCBOpenOut : opens new, empty specified file for write (creates file) (type: IO_TEXT or
IO_BIN)
# FCBOpenInOut : opens existing specified file for read/write (just opens, NOT creates file)
(type: IO_TEXT or IO_BIN)
# FCBOpenForAppend : opens existing specified binary file for write & positions the write
cursor after the last actual 128-bytes record,
# : or creates a new, empty binary file, if the specified file was not found
# FCBClose : closes the specified file (writing all the file data to disk if the file was opened
for write)
# FCBRewind : equivalent to FCBClose + FCBOpenIn, works only for files already opened
for read
# FCBGetChar : reads a byte from a file already opened for read or read/write
# FCBPutChar : writes a byte to a file already opened for write or read/write
#
# file types
const IO_TEXT := 0;
const IO_BIN := 1;
```

```
# I/O return codes (error numbers)
const SUCCESS := 0;
const ERR_NO_FILE := 1;
const ERR_BAD_IO := 2;
const ERR_DIR_FULL := 3;
const ERR_DISK_FULL := 4;
const ERR_EOF := 5;
@decl sub FCBOpenIn(fcb: [FCB], filename: [uint8], filetype: uint8): (errno: uint8)
@extern("FCBOpenIn");
@decl sub FCBOpenOut(fcb: [FCB], filename: [uint8], filetype: uint8): (errno: uint8)
@extern("FCBOpenOut");
@decl sub FCBOpenInOut(fcb: [FCB], filename: [uint8], filetype: uint8): (errno: uint8)
@extern("FCBOpenInOut");
#only for binary files
@decl sub FCBOpenForAppend(fcb: [FCB], filename: [uint8]): (errno: uint8)
@extern("FCBOpenForAppend");
@decl sub FCBGetChar(fcb: [FCB]): (c: uint8, errno: uint8) @extern("FCBGetChar");
@decl sub FCBPutChar(fcb: [FCB], c: uint8): (errno: uint8) @extern("FCBPutChar");
@decl sub FCBClose(fcb: [FCB]): (errno: uint8) @extern("FCBClose");
# only for files open for READ
@decl sub FCBRewind(fcb: [FCB]): (errno: uint8) @extern("FCBRewind");
```

When using these include files, you must add in the command line the files representing the libraries containing the actual Cowgol code.

E.g. if your Cowgol source file "test.cow" contains the line:

```
include "misc.coh";
```

, then you must add the library file "misc.coo" to the list of files included in the command line, e.g:

```
>cowgol -o misc.coo test.cow
```

3. There is still another way to use include files, combined with the use of libraries. These include files contain Cowgol source code, optimized by using Z80 assembly, referencing some routines stored to libraries.

libbasic.coh (Exit, MemSet, MemCopy)
libbios.coh(BiosSetup, ConOut, ConIn, ConSts, putstr)
libconio.coh (get_char, print_char, print, print_nl, print_i8, print_i16, print_hex_i8, print_hex_i16)
libstr.coh (IsDigit, ToLower, CopyString, StrCmp, StrICmp, StrLen)
libfp.coh (positive, nef, fpmul, fpdiv, fpsqrt, fpsin, fpcos, fparctan, xdifytofp)

When using these include files, you must add in the command line a reference to "cowgol.lib".

E.g. if your Cowgol source file "test.cow" contains the line:

```
include "libconio.coh";
```

, then the command line will be:

```
>cowgol -o -lcowgol test.cow
```

Compiling large Cowgol source files

Because of the limited TPA's size, on Z80 computers provided with 64KB RAM, some large Cowgol source files fail to compile ("not enough memory").

For Z80 computers provided with 128/512 (or more) KB RAM, including RomWBW systems, to compile these large Cowgol source files, an enhanced COWFE.COM is provided, able to allocate part of the structures used by the compiler to the extra RAM space, beyond the usual 64KB (see https://github.com/Laci1953/Cowgol_on_CP_M/tree/main/128_512_KB).