GNU Cobol Manual

for GNU Cobol 1.1

Keisuke Nishida / Roger While

Edition 1.1
Updated for GNU Cobol 1.1
20 January 2014

Table of Contents

GNU Cobol Manual

GNU Cobol is a free COBOL compiler, which translates COBOL programs to C code and compiles it using GCC or other native operating system C compiler.

This manual corresponds to GNU Cobol 1.1.

1 Getting Started

1.1 Hello World!

This is a sample program that displays "Hello World":

```
---- hello.cob -------------------------
      * Sample COBOL program
       IDENTIFICATION DIVISION.
       PROGRAM-ID. hello.
       PROCEDURE DIVISION.
       DISPLAY "Hello World!".
       STOP RUN.
----------------------------------------
```

The compiler is `cobc`, which is executed as follows:

```
$ cobc -x hello.cob
$ ./hello
Hello World!
```

The executable file name (i.e., `hello` in this case) is determined by removing the extension from the source file name.

You can specify the executable file name by specifying the compiler option `-o` as follows:

```
$ cobc -x -o hello-world hello.cob
$ ./hello-world
Hello World!
```

2 Compile

This chapter describes how to compile COBOL programs using GNU Cobol.

2.1 Compiler Options

The compiler `cobc` accepts the options described in this section.

2.1.1 Built Target

The following options specify the target type produced by the compiler:

`-E` Preprocess only. Compiler directives are executed. Comment lines are removed. COPY statements are expanded. The output goes to the standard-out.

`-C` Translation only. COBOL source files are translated into C files. The output is saved in file `*.c`.

`-S` Compile only. Translated C files are compiled by cc. The output is saved in file `*.s`.

`-c` Compile and assemble. This is equivalent to `cc -c`. The output is saved in file `*.o`.

`-m` Compile, assemble, and build a dynamically loadable module (i.e., a shared library). The output is saved in file `*.so`.[1].

`-x` Include the main function in the output.

This option takes effect at the translation stage. If you give this option with `-C`, you will see the main function at the end of the generated C file.

Without any options above, the compiler tries to build a dynamically loadable module.

2.1.2 Source Format

GNU Cobol supports both fixed and free source format.

The default format is the fixed format. This can be explicitly overwritten by one of the following options:

`-free` Free format.

`-fixed` Fixed format.

2.1.3 Warning Options

`-Wall` Enable all warnings

`-Wcolumn-overflow`
Warn any text after column 72

`-Wend-evaluate`
Warn lacks of END-EVALUATE

`-Wend-if` Warn lacks of END-IF

`-Wparentheses`
Warn lacks of parentheses around AND within OR

[1] The extension varies depending on your host.

2.2 Multiple Sources

A program often consists of multiple source files. This section describes how to compile multiple source files.

This section also describes how to build a shared library that can be used by any COBOL programs and how to use external libraries from COBOL programs.

2.2.1 Static Linking

The easiest way of combining multiple files is to compile them into a single executable.

One way is to specify all files on the command line:

```
$ cobc -x -o prog main.cob subr1.cob subr2.cob
```

Another way is to compile each file with the option `-c`, and link them at the end. The top-level program must be compiled with the option `-x`:

```
$ cobc -c subr1.cob
$ cobc -c subr2.cob
$ cobc -c -x main.cob
$ cobc -x -o prog main.o subr1.o subr2.o
```

You can link C routines as well using either method:

Method 1:

```
$ cobc -o prog main.cob subrs.c
```

Method 2:

```
$ cc -c subrs.c
$ cobc -c -x main.cob
$ cobc -x -o prog main.o subrs.o
```

Any number of functions can be contained in a single C file.

The linked programs will be called dynamically; that is, the symbol will be resolved at run time. For example, the following COBOL statement

```
CALL "subr" USING X.
```

will be converted into an equivalent C code like this:

```
int (*func)() = cob_resolve("subr");
if (func != NULL)
  func (X);
```

With the compiler options `-fstatic-call`, more efficient code will be generated like this:

```
subr(X);
```

Note that this option is effective only when the called program name is a literal (like `CALL "subr".`). With a data name (like `CALL SUBR.`), the program is still called dynamically.

2.2.2 Dynamic Linking

There are two methods to achieve this. Method 1 (Using driver program). Compile all programs with the option `-m`:

```
$ cobc -m main.cob subr.cob
```

This creates shared object files `main.so subr.so`[2].

Before running the main program, install the module files in your library directory:

```
$ cp subr.so /your/cobol/lib
```

Set the environment variable `COB_LIBRARY_PATH` to your library directory, and run the main program:

```
$ export COB_LIBRARY_PATH=/your/cobol/lib
```

Note: You may set the variable to directly point to the directory where you compiled the sources.

Now execute your program:

```
$ cobcrun main
```

Method 2. The main program and subprograms can be compiled separately.

The main program is compiled as usual:

```
$ cobc -x -o main main.cob
```

Subprograms are compiled with the option `-m`:

```
$ cobc -m subr.cob
```

This creates a module file `subr.so`[3].

Before running the main program, install the module files in your library directory:

```
$ cp subr.so /your/cobol/lib
```

Now, set the environment variable `COB_LIBRARY_PATH` to your library directory, and run the main program:

```
$ export COB_LIBRARY_PATH=/your/cobol/lib
$ ./main
```

2.2.3 Building Library

You can build a shared library by combining multiple COBOL programs and even C routines:

```
$ cobc -c subr1.cob
$ cobc -c subr2.cob
$ cc -c subr3.c
$ cc -shared -o libsubrs.so subr1.o subr2.o subr3.o
```

2.2.4 Using Library

You can use a shared library by linking it with your main program.

Before linking the library, install it in your system library directory:

```
$ cp libsubrs.so /usr/lib
```

or install it somewhere else and set `LD_LIBRARY_PATH`:

[2] The extension varies depending on your host.

[3] The extension varies depending on your host.

```
$ cp libsubrs.so /your/cobol/lib
$ export LD_LIBRARY_PATH=/your/cobol/lib
```

Then, compile the main program, linking the library as follows:

```
$ cobc -x main.cob -L/your/cobol/lib -lsubrs
```

2.3 C Interface

This chapter describes how to combine C programs with COBOL programs.

2.3.1 Writing Main Program in C

Include `libcob.h` in your C program. Call `cob_init` before using any COBOL module:

```
#include <libcob.h>

int
main (int argc, char **argv)
{
  /* initialize your program */
  ...

  /* initialize the COBOL run-time library */
  cob_init (argc, argv);

  /* rest of your program */
  ...

  /* Clean up and terminate - This does not return */
  cob_stop_run (return_status);

}
```

You can write `cobc_init(0, NULL);` if you do not want to pass command line arguments to COBOL.

You can compile your C program as follows:

```
cc -c `cob-config --cflags` main.c
```

The compiled object must be linked with libcob as follows:

```
cc -o main main.o `cob-config --libs`
```

2.3.2 Static linking with COBOL programs

Let's call the following COBOL module from a C program:

```
---- say.cob --------------------------
       IDENTIFICATION DIVISION.
       PROGRAM-ID. say.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       LINKAGE SECTION.
       01 HELLO PIC X(6).
```

```
        01 WORLD PIC X(6).
        PROCEDURE DIVISION USING HELLO WORLD.
        DISPLAY HELLO WORLD.
        EXIT PROGRAM.
----------------------------------------
```

This program accepts two arguments, displays them, and exit.

From the viewpoint of C, this is equivalent to a function having the following prototype:

```
extern int say(char *hello, char *world);
```

So, your main program will look like as follows:

```
---- hello.c ---------------------------
#include <libcob.h>

extern int say(char *hello, char *world);

int
main()
{
  int ret;
  char hello[7] = "Hello ";
  char world[7] = "World!";

  cob_init(0, NULL);

  ret = say(hello, world);

  return ret;
}
----------------------------------------
```

Compile these programs as follows:

```
$ cc -c `cob-config --cflags` hello.c
$ cobc -c -static say.cob
$ cobc -x -o hello hello.o say.o
$ ./hello
Hello World!
```

2.3.3 Dynamic linking with COBOL programs

You can find a COBOL module having a specific PROGRAM-ID by using a C function `cob_resolve`, which takes the module name as a string and returns a pointer to the module function.

`cob_resolve` returns NULL if there is no module. In this case, the function `cob_resolve_error` returns the error message.

Let's see an example:

```
---- hello-dynamic.c -------------------
#include <libcob.h>
```

```
static int (*say)(char *hello, char *world);

int
main()
{
  int ret;
  char hello[7] = "Hello ";
  char world[7] = "World!";

  cob_init(0, NULL);

  /* find the module with PROGRAM-ID "say". */
  say = cob_resolve("say");

  /* if there is no such module, show error and exit */
  if (say == NULL) {
    fprintf(stderr, "%s\n", cob_resolve_error ());
    exit(1);
  }

  /* call the module found and exit with the return code */
  ret = say(hello, world);

  return ret;
}
---------------------------------------
```

Compile these programs as follows:

```
$ cc -c `cob-config --cflags` hello-dynamic.c
$ cobc -x -o hello hello-dynamic.o
$ cobc -m say.cob
$ export COB_LIBRARY_PATH=.
$ ./hello
Hello World!
```

2.3.4 Static linking with C programs

Let's call the following C function from COBOL:

```
---- say.c ----------------------------
int
say(char *hello, char *world)
{
  int i;
  for (i = 0; i < 6; i++)
    putchar(hello[i]);
  for (i = 0; i < 6; i++)
    putchar(world[i]);
```

```
  putchar('\n');
  return 0;
}
----------------------------------------
```

This program is equivalent to the foregoing `say.cob`.

Note that, unlike C, the arguments passed from COBOL programs are not terminated by the null character (i.e., `\0`).

You can call this function in the same way you call COBOL programs:

```
---- hello.cob -------------------------
       IDENTIFICATION DIVISION.
       PROGRAM-ID. hello.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 HELLO PIC X(6) VALUE "Hello ".
       01 WORLD PIC X(6) VALUE "World!".
       PROCEDURE DIVISION.
       CALL "say" USING HELLO WORLD.
       STOP RUN.
----------------------------------------
```

Compile these programs as follows:

```
$ cc -c say.c
$ cobc -c -static -x hello.cob
$ cobc -x -o hello hello.o say.o
$ ./hello
Hello World!
```

2.3.5 Dynamic linking with C programs

You can create a dynamic-linking module from a C program by passing an option `-shared` to the C compiler:

```
$ cc -shared -o say.so say.c
$ cobc -x hello.cob
$ export COB_LIBRARY_PATH=.
$ ./hello
Hello World!
```

3 Customize

3.1 Customizing Compiler

These settings are effective at compile-time.

Environment variables (default value):

`COB_CC` C compiler ("gcc")

`COB_CFLAGS`
: Flags passed to the C compiler ("-I$(PREFIX)/include")

`COB_LDFLAGS`
: Flags passed to the C compiler ("")

`COB_LIBS` Standard libraries linked with the program ("-L$(PREFIX)/lib -lcob")

`COB_LDADD`
: Additional libraries linked with the program ("")

3.2 Customizing Library

These settings are effective at run-time.

Environment variables (default value):

`COB_LIBRARY_PATH`
: Dynamic-linking module path (".:$(PREFIX)/lib/gnu-cobol")

`COB_DYNAMIC_RELOADING`
: Set to “yes” if modules must be reloaded when they are replaced (“no”)

4 Optimize

4.1 Optimize Options

There are three compiler options for optimization: `-O`, `-Os` and `-O2`. These options enable optimization at both translation (from COBOL to C) and compilation (C to assembly) levels.

Currently, there is no difference between these optimization options at the translation level.

The option `-O`, `-Os` or `-O2` is passed to the C compiler as it is and used for C level optimization.

4.2 Optimize Call

When a CALL statement is executed, the called program is linked at run time. By specifying the compiler option `-fstatic-call`, you can statically link the program at compile time and call it efficiently. (see Section 2.2.1 [Static Linking], page 6)

4.3 Optimize Binary

By default, data items of usage binary or comp are stored in the big-endian form. On those machines whose native byte order is little-endian, this is not quite efficient.

If you prefer, you can store binary items in the native form of your machine. Set the config option `binary-byteorder` to `native` in your config file (see Chapter 3 [Customize], page 13).

In addition, setting the option `binary-size` to `2-4-8` or `1-2-4-8` is more efficient than others.

5 Debug

5.1 Debug Options

The compiler option `-debug` can be used during the development of your programs. It enables all run-time error checking, such as subscript boundary checks and numeric data checks, and displays run-time errors with source locations.

The compiler option `-g` includes line directives in the output. This helps debugging your COBOL programs using GDB, but this feature is not complete and not very useful yet.