The CMOC C-like 6809-targeting cross-compiler

Pierre Sarrazin (sarrazip@sarrazip.com)

2020-06-07

http://sarrazip.com/dev/cmoc.html

Version of CMOC covered by this manual: **0.1.67**

Contents

Introduction

CMOC is a Unix-based program that compiles a C-like language, generates Motorola 6809 assembly language programs and assembles them into executables for the Color Computer's Disk Basic environment. Targeting the Dragon computer and the Vectrex video game console is also supported.)

The efficiency of the generated machine language is modest, but the resulting machine language will be faster than the equivalent interpreted Color Basic program. This was the initial goal of the CMOC project.

CMOC itself is written in C++ for a Unix-like system. The source code is somewhat complicated because of the anarchic development process that gave

rise to CMOC. Years ago, development was able to take off because having a working product had higher priority than having a clean but theoretical design.

C Language Features

Unsupported C features

- `float` and `double` arithmetic for a target other than the CoCo Disk Basic environment. (CMOC generates calls to the Color Basic floating point routines.)
- Double-precision floating-point arithmetic (the `double` keyword is accepted but is an alias for `float`).
- 64-bit arithmetic (no `long long` type).
- The `volatile` keyword. (It is accepted but ignored, and using it causes a warning to be issued.)
- Bit fields. The *TYPE NAME : BITS* notation is actually accepted by the compiler, since version 0.1.52, but the field is allocated with the given *TYPE*, regardless of the number of bits specified after the colon.
- Arrays of function pointers (and typedefs thereof).
- Typedefs local to a function (global typedefs are supported).
- Redefining a typedef name, event with an identical definition (e.g., `typedef int INT; typedef int INT;`).
- Structs local to a function (global structs are supported).
- Indirection of a pointer to a *struct* used as an r-value (e.g., `*ptr` alone). The l-value case is supported, e.g., `(*ptr).field`.
- The `register` keyword is accepted but ignored.
- K&R function definitions, e.g., f() int a; { ... }
- A `continue` statement in a switch() body.
- An expression of type long, float or double as an argument of a switch().
- Implementing Duff's device in a switch().
- Function-local function prototypes. All prototypes must be declared at global scope.
- Zero-element arrays.

Supported C and C99 features

- Single-precision floating-point arithmetic (since version 0.1.40) under the Disk Basic environment (the `double` keyword is accepted but is an alias for `float`).
- 8-, 16- and 32-bit arithmetic. Type `char` is 8 bits, `short` and `int` are 16 bits and `long` is 32 bits. Each of these types can be `signed` or `unsigned`.
- Pointers, including pointers to pointers, etc.
- Structs, including a struct in a struct. Struct initializers. A struct must be declared at the global level. A reference to an undefined struct is accepted if the declaration or statement only refers to a pointer to such a struct.
- Anonymous structs.
- Assigning a struct value to another (e.g., `struct S a, b; a = b;`). Initializing a struct from another one is also supported (e.g., `struct S a; struct S b = a;`).
- Passing a struct or union by value to a function (since version 0.1.40). (If the size of the struct or union is 1 byte, then it is passed as a 16-bit word whose least significant byte is the struct or union.)
- Returning a struct or union by value from a function (since version 0.1.40).
- Declaring a variable after the function's code has started, as in C99.
- while, do, for, switch, continue, break.
- Declaring a for() loop's control variable in the for() itself as in C99, e.g., for (int i = 0; ...) {}.
- Declaring more than one variable on the same line, e.g., int a = 0, b = 1;
- Variadic functions, e.g., void foo(char *format, ...). There must be at least one named argument before the ellipsis (...), as in ISO C.
- Ending an initializer list with a comma.
- Use of the C preprocessor (the system's cpp is invoked): #include, #define, #ifdef, etc.
- Unions.
- Enumerations.
- Type-safe function pointers.
- The `const` keyword. (Const-correctness issues are diagnosed as warnings, not as errors, to avoid breaking code written before version 0.1.50.)
- Comma expressions (e.g., x = 1, y = 2, z = 3;).

- Binary literals (e.g., `0b101010` for 42). Note that this feature is not part of Standard C.
- Goto and non-case labeled statements.

Installing CMOC

The following instructions assume a Unix-like system.

Requisites

- A C++ compiler, like GNU C++ (g++)
- A C preprocessor (named "cpp"), like the GNU C preprocessor
- GNU Make (build tool)
- GNU Bison (parser generator)
- GNU Flex (lexical analyzer generator)
- LWTOOLS (assembler)

The compiler

Deploy the .tar.gz source archive in a directory, go to this directory, read the generic installation instructions in the INSTALL file, then (typically) give these commands:

The sudo prefix may be needed to do a `make install` to a system directory like /usr/local/bin.

The "check" makefile target runs several unit tests.

The compiler executable is called "cmoc".

To generate the HTML documentation (this document), do `make html`, which will create the file `doc/cmoc-manual.html`.

Running CMOC

Compiling a C program

The following must be in the search path:

- A C preprocessor callable by the name "cpp".
- The LWTOOLS lwasm assembler, and lwlink linker. If libraries are to be created, then the lwar archiver must also be in the search path.

To compile a program written in the file foo.c, run this command:

cmoc foo.c

By default, the resulting .bin file will load at address $2800 on a CoCo (see the `#pragma org` directive elsewhere in this manual). The generated machine language is **position-independent**, so the .bin file can be loaded with an offset, by passing a second argument to Disk Basic's LOADM command.

Pass `--help` to cmoc to see the available options.

CMOC comes with a `writecocofile` command that can copy a .bin file to a 35-track Disk Basic diskette image file. For example:

writecocofile testing.dsk foo.bin

Pass `--help` to writecocofile for the available options.

For more information on running a CoCo emulator on a GNU/Linux system, see http://sarrazip.com/emul-coco.html.

Disabling some optimizations

By default, all peephole optimizations are enabled.

The peephole optimizer can be disabled by passing the `-O0` option to the compiler. This can be useful when one suspects that the compiler may have generated invalid code.

Several peephole optimizations were contributed by Jamie Cho in early 2016 for version 0.1.20. They can be disabled by passing `-O1`.

Option `-O2` is equivalent to using the default (full) optimization level.

Generated files

By default, compiling a C file gives a **.o** object file when option -c is passed, or a .bin executable if -c is not passed and the C file is a complete program.

When the –intermediate (or -i) option is passed, the following intermediate files are also generated:

- **.s**: The assembler file written by CMOC.
- **.lst**: The listing generated by the 6809 assembler (lwasm) from a .s file.

When linking, these intermediate files are also generated:

- **.map**: The linking map generated by lwlink. It lists the sections and symbols that appear in the final executable.
- **.link**: The linking script for lwlink. It lists the sections that must be gathered by the linker to produce the final executable.

Sine CMOC 0.1.62, the `--intdir=`*D* option can be used to have these intermediate files generated in directory *D*.

The executable in Disk Basic **.bin** format can be transfered to a CoCo or emulator and loaded with the LOADM command. (If compiling for OS-9—with the `--os9` command-line switch—the executable has the name of the C file without the extension.)

When distributing the .bin file to CoCo users, it is not necessary or useful to accompany it with the other generated files.

The Motorola S-record (SREC) format

Pass the `--srec` option to cmoc to make it generate an executable in Motorola S-record format. The executable file will have the .srec extension.

Since version 0.1.43 of the compiler, this format is the default one when targetting the USim 6809 simulator (with the `--usim` option).

The Vectrex video game console

The doc subdirectory of the source distribution contains cmoc-vectrex.markdown, which gives Vectrex-specific instructions. Giving the `make html` command will generate cmoc-vectrex.html, which can be viewed in a browser.

Note that questions regarding Vectrex-specific issues should be addressed to **Johan Van den Brande** at `johan@vandenbrande.com`.

The Dragon computer

The Dragon was a clone of the Color Computer. It ran a Basic interpreter and a disk operating system that were simitar to the CoCo's. The .bin file format was different however.

CMOC will generate a Dragon .bin file when it is given the `--dragon` command-line switch.

If a Dragon program is separated in several C files, each of these files must be compiled with the `--dragon` switch, and the linking invocation must also be given `--dragon`.

As of CMOC 0.1.56, floating point types are not supported when targeting the Dragon. The `<disk.h>` library is also not usable on the Dragon. These missing features might become available in future versions of the compiler.

Modular compilation and linking

Since version 0.1.43, CMOC has supported modular (separate) compilation as well as linking several modules (object files) together.

In other words, the user can now separate a large program into several .c files, compile each of these files to a .o object file, then have CMOC link them together into a final executable.

LWTOOLS' lwasm and lwlink commands must be in the search path, otherwise, the user can specify the `--lwasm=` and `--lwlink=` command-line options to specify the full path of these commands.

Target specification

When a program is separated in multiple C files, each of these files must be compiled with the same target command-line switch (e.g., `--coco`, `--dragon`, `--os9`, `--vectrex`). The linking invocation — the one that generates the final executable — must also be given the same switch. This ensures that all parts of the program are compiled consistently, i.e., they all assume the same underlying environment.

Creating libraries

Object files (.o) can be grouped into a library by using LWTOOLS' lwar command:

```
lwar -c libstuff.a foo.o bar.o baz.o
```

Library filenames should have the `lib` prefix and be given the `.a` extension, so that they will be compatible with the `-l` option that is used to specify a library to link with, e.g.:

```
cmoc -o prog.bin prog.c -lstuff
```

In this case, only the .o files of libstuff.a that are actually needed will be copied to the executable.

It is possible to specify `libstuff.a` on the CMOC command line, but then all of the library's object files get copied to the executable.

The `-l` option must be specified after the source or object file(s).

The `-L` option can be used to specify a directory in which to search for libraries specified by `-l`. The `-L` option can be specified before or after the source or object file(s).

An object file is indivisible as far as the linker is concerned, and therefore, so is a C file. This means that either all of the functions and globals in a C file will

end up in the executable, or none of them will. When designing a library, it may be desirable to put each function in its own C file, so that only the functions used by the parent program will be copied into the executable.

User library constructors and destructors

Defining the constructor and/or destructor

A library that does not come with CMOC might need to run some code before and/or after the main() function. Some of the uses for this include initializing some global variables or releasing some resources when the program quits.

To have code executed before main(), the library author must create a .asm file that defines a `constructors` section. For example:

```
IMPORT  _toolkit_constructor
SECTION constructors
lbsr    _toolkit_constructor
ENDSECTION
```

(Each line in the previous file must be indented.)

The code above calls a C function called toolkit_constructor() (without an initial underscore). Some .c file in the library must define that function, which must execute whatever initialization is needed by the library.

In the example above, the LBSR instruction is used instead of JSR to maintain the relocatability of the library.

To have code executed after main(), the same .asm file can be used to create a `destructors` section.

```
IMPORT  _toolkit_destructor
SECTION destructors
lbsr    _toolkit_destructor
ENDSECTION
```

Here, a C function called toolkit_destructor() is called.

The .asm file may define only one of these two sections. It does not have to define both.

Compiling the library

If the .asm file is called prepostmain.asm for example, then it must be assembled into an object file with a command like the following:

```
lwasm -fobj -o prepostmain.o prepostmain.asm
```

This object file must *not* be included in the .a file that comprises the other object files of the library. (A .a file can be created with a `lwar -c` command. See the "Creating libraries" section elsewhere in this manual.)

Installing the library

When the library is installed somewhere, the .o file that contains the constructor and destructor (e.g., prepostmain.o) must be installed along with the library's .a file.

Using the library

Once the library has been installed in a directory, a program that uses this library must add some arguments to the CMOC invocation that links the program. If for example the library is named "toolkit" and it was installed in /usr/local/tools, then this directory contains both prepostmain.o and libtoolkit.a, and the following arguments must be passed to CMOC when linking the program:

```
/usr/local/tools/prepostmain.o -L /usr/local/tools -ltoolkit
```

The prepostmain.o file must be specified explicitly to force the linker to include all of its contents in the executable. If this .o file were only included in the .a file, then it would not be used by the linker because no code refers explicitly to the contents of prepostmain.asm.

Multiple definitions

Because of the way the linker works, there is no error message when the same function or global variable is defined with external linkage by two modules. Only a warning is issued. Such a warning should be viewed as an error and the duplication should be resolved.

For similar reasons, warnings will be issued when two modules define static functions or globals with the same name, e.g., `static int n;`. Such warnings may be safely ignored, because the symbols are static, thus not exported, and will not actually clash.

These ambiguous diagnostic messages may be fixed by future versions of CMOC and LWTOOLS.

Specifying code and data addresses

In a modular program, the address at which code and data must be positioned must *not* be specify with the `#pragma org` and `#pragma data` directives. Those addresses must be specified with the `--org` and `--data` command-line options when invoking CMOC to perform linking:

The same goes for `#pragma limit`: the `--limit` option must be used instead.

Starting with version 0.1.43, these pragmas can only be used when compiling directly from a C file to an executable, i.e., the `-c` option is not used.

Assembly language modules

An object file typically comes from a C file, but an assembly language file can also be passed to CMOC, which will invoke the assembler on it. That file's extension must be `.s` or `.asm`.

There are conventions to be observed however:

- Most of the code must be in an assembler section named `code`:
- The exception is code that initializes global variables. That code must be in a section called initgl. Such a section must **not** end with an RTS instruction:
- Functions and global variables that are to be made available to other modules must be exported with an EXPORT directive, e.g., `_func EXPORT`.
- Functions and global variables that are expected to be provided by *other* modules must be imported with an IMPORT directive, e.g., `_printf IMPORT`.
- The code should preferrably be position-independent, but that is not a requirement if the executable will always be loaded at the address it was compiled for.
- See the *Calling convention* section elsewhere in this manual for the rules to observe.
- Read-only global variables and values must be in a section named `rodata`. This includes string literals. String literals must end with a null byte. If the `\n` sequence is used in the literal, it must be encoded as byte 10 ($0A).
- Writable global variables that have static initializers (typically arrays) must be in a section named `rwdata`.
- Writable global variables that do not have initializers, or that are initialized by executing code in an `initgl` section, must be in a section named `bss`. This section must only give `RMB` directives, and no `FCC`, `FDB` or `FCB` directives. The `bss` section follows that rule so that it does not take any space in the executable file (at least with the Disk Basic BIN format).
- Function names and global variable names must start with an underscore. In other words, if the C name is `foo`, then the assembly language name must be `_foo`.

Merging a binary file with the executable

To include an arbitrary file in the final executable, LWASM's `INCLUDEBIN` directive can be used.

As an example, assume the binary file is called `blob.dat` on the filesystem of the computer the program is being developed on. Create a `blob.asm` file that contains this:

```
        SECTION rodata
        EXPORT _blob
        EXPORT _blob_end
_blob
        INCLUDEBIN blob.dat
_blob_end
        ENDSECTION
```

Specifying `rodata` will put the data in the read-only section of the program. To put it in the read-write section, specify `rwdata` instead.

Prefixing the symbols with an underscore is necessary to comply with CMOC's naming convention. This avoids conflicts with symbols generated by CMOC.

Invoke LWASM to have it generate an object file:

```
lwasm -fobj --output=blob.o blob.asm
```

In the C program, access the data this way:

```
    unsigned char *start, *end;
    asm
    {
_blob     IMPORT
_blob_end IMPORT
        leax    _blob,pcr
        stx     :start
        leax    _blob_end,pcr
        stx     :end
    }
```

After this code, the `start` and `end` pointers delimit the data as loaded in memory on the target machine. (Note that there is no way to force the data to be loaded at a specific address. The position of the data is decided by the linker, the origin address set by `--org` and the offset passed to `LOADM` on a CoCo.)

When invoking CMOC to perform the linking phase, add `blob.o` to the command line.

See the LWTOOLS site for details on assembler directives.

Importing symbols used by inline assembly

If an inline assembly block uses a global variable that is provided by another module, an import directive must be included:

Generating Prerequisites Automatically

A multi-module project typically uses a makefile to manage the build process.

CMOC can automatically generate a dependencies file, with a `.d` extension, that lists the source files that were encountered during the compilation of a C file. The dependency file will be compatible with GNU Make.

This is done by passing either the `--deps-only` or `--deps` option to the compiler. A `.d` file will be generated with a makefile rule like this:

Option `--deps-only` stops right after generating this dependency file. Option `--deps` can be used with `-c` to have the compiler both compile the C file and generate the dependency file.

The Generating Prerequisites Automatically section of the GNU Make manual should be consulted for details on how to write a makefile that uses this mechanism. GNU Make's `-M` is similar to `--deps-only`, while `-MMD` is similar to `--deps`. Note that the `sed` command mentioned in that manual is not necessary with CMOC, which automatically includes the `.d` filename in the generated rule.

The name of the `.d` file is formed from the name of the `.o` file to be generated.

This option has no effect when also specifying the `-E` option (which prints the preprocessor output and stops).

Programming for CMOC

Binary operations on bytes

Binary operations on bytes give a byte under CMOC, whereas they give an integer under Standard C. To get a warning for such operations, pass the `-Wgives-byte` command-line option. This can be useful when porting an existing C program to CMOC.

Signedness of integers

The compiler generally produces more efficient code for unsigned arithmetic. Therefore, types `unsigned` and `unsigned char` should be preferred to `int` and `char` where possible.

CMOC considers that the signedness of an additive or multiplicative operation is the signedness of the left operand.

CMOC can issue a warning when comparison operators `<`, `<=`, `>` or `>=` are used with operands of different signedness. This is obtained by passing the `-Wsign-compare` command-line option.

Pre-increment vs. post-increment

The pre-increment (++i) typically generates one instruction less than the post-increment (i++). This is because the post-increment must yield the *initial* value of the variable. When this initial value is not needed, it is advisable to use the pre-increment.

The same situation holds for the decrement operators.

Origin address

To specify the origin address of the program, start your program with a `#pragma org` directive. For example:

```
#pragma org 0x5C00
```

By default, the origin address is 0x2800. (Under Disk Basic, the Basic program normally starts at 0x2601, because four PMODE pages are reserved by default.)

End address and length of the executable

To determine the effective start and end address of the executable (assuming no relocation by LOADM), one can look up the `program_start` and `program_end` symbols in the .lst listing file generated by the assembler.

`program_end` is useful to check that the executable fits in the available RAM. On a 32k CoCo, the RAM ends at $8000. By defaut, Color Basic reserves 200 bytes for Basic strings, and before that, the system stack should be given 512 bytes for a typical C program. These assumptions mean that an executable should not go beyond $7D38.

If a program crashes the CoCo just by getting loaded, it could be because it is too long and overwrites the system stack. The `parse-coco-bin` Perl script, available on the CMOC home page, can be useful to confirm that.

The following example program prints the start and end of the memory it uses initially:

```
#include <cmoc.h>
int main()
```

```
{
    char *s, *e;
    asm
    {
        leax    program_start,pcr
        stx     :s
        leax    program_end,pcr
        stx     :e
    }
    printf("START AT %p, END AT %p.\n", s, e);
    return 0;
}
```

Note that the label addresses are taken relatively to the program counter (`,pcr`), so that this program will report the correct addresses even if the program is loaded at an address other than the one specified in the .bin file. (This can be done by passing a second argument to Disk Basic's LOADM command.)

This section assumes that the code and data sections form a single block. When using the `#pragma data` directive (see elsewhere in this manual), the *writable* globals will not be between `program_start` and `program_end`.

Enforcing a limit address on the end of the program

As of version 0.1.20, CMOC accepts the `--limit=X` option. When it is passed, CMOC will fail if the end of the program, as indicated by the `program_end` listing symbol, exceeds address X (specified in hexadecimal).

For example, `--limit=7800` will keep the program from finishing too close to the system stack under Disk Basic, which is not far below $8000. A limit at $7800 leaves two kilobytes for the stack and for Basic's string space.

It is not necessary to pass the `--verbose` option to use the `--limit` option.

A `#pragma limit 0xNNNN` directive in the source code achieves the same purpose as `--limit`.

Position-independent code

All 6809 code generated by CMOC is position-independent, i.e., it can be loaded at any address and will still work.

Determining that CMOC is the compiler

Since version 0.1.48, CMOC automatically defines `_CMOC_VERSION_` to be a 32-bit number of the form XXYYZZZ when the version is X.Y.Z. For example,

version 0.1.48 defines `_CMOC_VERSION_` as 1048. This can be useful when some C code needs to do something under CMOC but something else under another compiler.

Specifying the target platform

By default, the compiler defines the `_COCO_BASIC_` preprocessor identifier. This identifier can be used to adapt a program to make it use alternative code depending on whether it will run under Disk Extended Color Basic or not.

To target OS-9, pass the `--os9` option. The compiler will define `OS9`.

To target the Vectrex, pass `--vectrex`. The compiler will define `VECTREX`.

To target the Dragon, pass `--dragon`. The compiler will define `DRAGON`.

When passing `--usim`, the compiler targets the USim 6809 simulator, which comes with CMOC. The `USIM` identifier will be defined. No .bin file is produced in this case. The .srec file can be executed by passing its path to `src/usim-0.91-cmoc/usim`.

The standard library

CMOC's standard library is small. The program must `#include <cmoc.h>` to use functions like printf(). See that file for a list of implemented C functions. Many are C functions while others are CMOC extensions. ("Standard" here means that those functions come with CMOC, not that CMOC aims to provide a complete C standard library.)

readline() acts like Basic's LINE INPUT command and returns the address of the (NUL-terminated) string entered. This address is a global buffer. The next call to readline() will overwrite that buffer.

printf()

CMOC's printf() function supports %u, %d, %x, %X, %p, %s, %c, %f and %%. Specifying a field width is allowed, except for %f. A left justification is only supported for strings, e.g., `%-15s` will work, but `%-6u` will not. Zero padding for an integer is supported (e.g., `%04x`).

The `l` modifier is supported for 32-bit longs (e.g., "%012ld");

%p always precedes the hexadecimal digits with `$`, as per the CoCo assembly language notation. %x and %X do not generate such a prefix. %p, %x and %X always print letter digits as capital letters (`A` to `F`, not `a` to `f`).

printf(), like putchar() and putstr(), sends its output one character at a time to Color Basic's PUTCHR routine, whose address is taken from $A002.

Redirecting the output of printf()

The standard library's printf() writes the characters of the formatted string by calling the routine whose address is stored in the library's CHROUT global variable (not to be confused with Color Basic's CHROUT vector at $A002). The same applies to functions sprintf(), putchar() and putstr().

By default, under Color Basic, that routine is the one found in that $A002 vector. To designate a C function as the new character output routine, first define the new routine:

```
void newOutputRoutine()
{
    char ch;
    asm
    {
        pshs    x,b  // preserve registers used by this routine
        sta     :ch
    }

    // Insert C statements that print or otherwise process 'ch'.

    asm
    {
        puls    b,x
    }
}
```

This routine will receive the character to be printed in register A. It **must** preserve registers B, X, Y and U. It does not have to preserve A.

Install it at the appropriate time with this call:

```
void *oldCHROOT;

oldCHROOT = setConsoleOutHook(newOutputRoutine).
```

To restore the original output routine, do this:

```
setConsoleOutHook(oldCHROOT);
```

sprintf()

This function is like printf(), but it writes into a memory buffer whose address is passed as the first argument, before the format string. For example:

```
char greeting[32];
void f(char *name)
{
    sprintf(greeting, "Hello, %s.", name);
```

```
}
```

Calling f("Lonnie") will write "Hello, Lonnie." in the greeting[] array, including a terminating '\0' character. A total of 15 bytes get written to the start of that array.

The **caller is responsible for providing a buffer long enough** to receive all the text resulting from the format string and its arguments, *including the terminating '\0' character.*

In this example, the longest "name" that can be safely passed to f() would be a 23-character name.

The standard C language offers snprintf(), which is safer because it requires passing the length of the destination buffer. But checking for this limit would have a performance hit that is not necessarily acceptable on a CoCo. If such a function is needed, it can be implemented using the technique described in the previous section.

Redefining a standard library function

Defining a function using the name of a standard library function is allowed. For example, a program could redefine `rand()` and `srand()` to implement an alternative random number generator. In the final assembly program, such functions replace the ones provided by the standard library.

Dynamic memory allocation with sbrk()

sbrk() is a function that dynamically allocates a region of memory of the size (in bytes) passed to it as an argument. It returns a void pointer. If the quantity of memory requested is not available, sbrk() returns `(void *) -1`. For example:

```
void *p = sbrk(100);
if (p != (void *) -1)
    memset(p, 'X', 100);
```

The name of the function comes from the notion of a "program break", which is the current end of the memory allocated to the program. The memory after that is presumed to be free for dynamic allocation.

In the case of the CoCo, the assumption is that the program is loaded after the Basic program and variables. This means the space that `sbrk()` can allocate from goes from there to the top of the stack, which is around $3F00 on a 16K CoCo and $7F00 on a 32K-or-more CoCo. Do not use `sbrk()` if these assumptions do not apply, e.g., when using `--data` to position the writable globals elsewhere than right after the code and read-only data.

Use the `--stack-space` option or the `#pragma stack_space` directive (documented elsewhere in this manual) when the program needs more than 1024 bytes

of stack space.

To determine how much of that memory is available for sbrk(), call sbrkmax(), which returns the number of bytes as a size_t (unsigned). CMOC ends the region available to sbrk() about 1024 bytes before the stack pointer, leaving those bytes to program calls and local variables.

sbrkmax() returns 0 if the program is loaded after the stack space.

Inline assembly

Inline assembly text can be specified by surrouding it with `asm {` and `}`. In the text, one can refer to C variables (global, local and parameters) as well as functions. Labels can be used for branch instructions, but a label must either be unique to the whole program or comply with what lwasm considers a "local" label. Prefixing a global label with the name of the current C function is a good way to help prevent clashes. A label must appear at the very beginning of the line, without spaces or tabs in front of it.

One way of using lwasm **local labels** is to prefix the label name with the @ character. Such a label will be local to the current block, which will begin at the previous blank line (or start of the `asm` block) and end at the next blank line (or end of the `asm` block). Refer to the LWASM manual about its symbols for details on using local labels.

The assembler may also support `$` as a local label marker, but it is not recommended to use it that way in inline assembly because it may hinder portability to OS-9, where `$` is not used as a local label marker.

The following example fills array `out` with `n` copies of character `ch`, then returns the address that follows the region written to:

```
#include <cmoc.h>

char *f(char *out, char ch, unsigned char n)
{
    char *end;
    asm
    {
        ldx     out         /* comments must be C style */
        lda     ch          // or C++ style
        ldb     n           ; or semi-colon comment (passed to assembler)
f_loop:
        sta     ,x+
        decb
        bne     f_loop
        stx     end
    }
```

```
    return end;
}

int main()
{
    char a[10];
    a[9] = '\0';
    char *p = f(a, 'X', (unsigned char) sizeof(a) - 1);
    printf("a='%s', %p, %p\n", a, a, p);
    return 0;
}
```

Since version 0.1.21, when referring to a C function, the function name is replaced with its assembly label, possibly followed by the `,pcr` suffix. This suffix is omitted if the instruction is BSR, LBSR or JSR, because these instructions do not support the `,pcr` suffix and they do not need it anyway. The following example calls the same C function three different ways:

```
asm
{
    jsr     someCFunction
    lbsr    someCFunction
    leax    someCFunction
    jsr     ,x
}
```

Note that CMOC always generates position independent code. This rule should be maintained in inline assembly if the resulting program is to be relocatable.

The BSR instruction is not recommended because it is a short branch and if the called function is too far from the call, the assembly step will fail.

Since 0.1.39, semi-colon comments are supported. They are passed verbatim to the assembler.

Note that using inline assembly is likely to make the program non portable to other C compilers.

See the *Calling convention* section elsewhere in this manual for the rules to observe. Namely, inline assembly must not modify U or Y. It is allowed to modify D, X and CC.

Preprocessor identifiers in inline assembly

The GNU C preprocessor can add spaces in surprising ways, which makes its use slightly problematic in inline assembly. For example:

```
#define PIA0 0xFF00
```

```
asm
{
    stb     PIA0+2
}
```

One would expect this code to generate an `stb 0xFF02` instruction, but cpp will actually expand this to `stb 0xFF00 +2`, because it apparently adds a space after the expansion of the PIA0 identifier.

The assembler takes this space as the start of the comment, so it ignores the +2 and assembles `stb $FF00`.

A workaround appears to be to reverse the addition and write `stb 2+PIA0`. No space gets added before the identifier.

Therefore, preprocessor identifiers should be used with caution in inline assembly.

Referring to variables whose name is that of a register

Before version 0.1.31, bad assembly language text could be emitted if inline assembly referred to a C variable that has the name of a register. To help resolve this ambiguity, version 0.1.31 introduced a *C variable escape character*, which is a colon that is put in front of the C variable name.

For example:

```
char b;
asm
{
    inc     :b
    ldb     :b
    leax    b,x
}
```

Here, `:b` refers to variable *b* of the C program, while the *b* in *b,x* refers to the register.

This change may break inline assembly code in programs that were written for versions of CMOC preceding 0.1.31. Adding a colon at the right places will resolve the issue.

Note that the escape character is not necessary on variable names that are not also register names.

Assembly-only functions

When a function is written entirely in assembly language for performance reasons, the stack frame may not be necessary. CMOC will not generate a stack frame for a function defined with the `asm` modifier, as in this example:

```
asm int f(int m, int n)
{
    // U not pushed, so 1st argument is at 2,s
    asm
    {
        ldd     2,s     // load m
        addd    4,s     // add n, leave return value in D
    }
}
```

Only `asm { ... }` statements are allowed in such a function. Typically, only one is needed. Local variables cannot be defined in that function and the function's parameters cannot be accessed by name. The assembly code is allowed to refer to global variables however.

By default, the compiler ends the function with the RTS instruction (or RTI if the function is declared with the `interrupt` modifier). To keep the compiler from emitting that instruction (RTI as well as RTS), add the `__norts__` keyword:

```
asm __norts__ int f(int m, int n) { ... }
```

See the *Calling convention* section elsewhere in this manual for the rules to observe. In particular, note that byte arguments are promoted to words, which are pushed onto the stack in the big endian byte ordering.

Interrupt Service Routines

CMOC supports the `interrupt` function modifier, which tells the compiler to end the function with an RTI instruction instead of an RTS. For example, the following function handles the VSYNC 60 hertz interrupt:

```
interrupt void newCoCoIRQRoutine()
{
    asm
    {
        lda     $FF03           // check for 60 Hz interrupt
        lbpl    irqISR_end      // return if 63.5 us interrupt
        lda     $FF02           // reset PIA0, port B interrupt flag
    }

    // Do something in C.

    asm
    {
irqISR_end:
    }
    // Nothing here, so that next instruction is RTI.
```

```
}
```

This routine could be hooked to the IRQ vector this way on a CoCo:

```
disableInterrupts();
unsigned char *irqVector = * (unsigned char **) 0xFFF8;
*irqVector = 0x7E;  // extended JMP extension
* (void **) (irqVector + 1) = (void *) newCoCoIRQRoutine;
enableInterrupts();
```

Header `<coco.h>` defines macros disableInterrupts() and enableInterrupts(). They set and reset the F and I condition codes.

The FIRQ vector is at $FFF6.

Note that using the `interrupt` keyword is likely to make the program non portable to other C compilers.

Function pointers

The address of a function can be taken and stored in order to be called through that pointer later.

The following example shows that the two syntaxes used in C to call a function through a pointer are supported by CMOC:

```
unsigned char f(int a, char b) { ... }
int main()
{
    unsigned char (*pf)(int a, char b) = f;
    unsigned char c0 = (*pf)(1000, 'x');
    unsigned char c1 = pf(1001, 'y');
    return 0;
}
```

A member of a struct can point to a function. For example:

```
struct S
{
    void (*fp)();
};
void g() { ... }
int main()
{
    struct S s = { g };  // member 'fp' points to g()
    s.fp();  // call g()
    (*s.fp)();  // call g()
    return 0;
}
```

Array initializers

Local vs. global array

When only a single function needs to use a read-only array of integers, this array could be defined locally to that function, but it is more efficient, as of CMOC 0.1.10 to define the array as global. This is because the global array will be emitted as a sequence of FCB or FDB directives, while a local array will be initialized with a series of load and store instructions.

In the following example, array `g` will be initialized more efficiently than array `l`:

```
int g[] = { 55, 66, 77 };
void f()
{
    int l[] = { 22, 33, 44 };
    /* ... */
}
```

Execution of global array initializers

A global array initializer containing only integer values (not string literals) is treated specially. It is initialized at compile-time, not at run-time. This means that if the program modifies the array, quits, and then is reexecuted, the modified contents will still appear as is. The array will not be reinitialized to the values in the initializer.

For example:

```
#include <cmoc.h>
int a[2] = { 13, 17 };
int main() { a[0]++; printf("%d\n", a[0]); return 0; }
```

The first time this program is executed, a[0] starts at 13, then is incremented to 14, which is the number that is printed.

The second time this program is executed, a[0] starts at 14 because array `a` is *not* reinitialized upon entry.

This is meant to save memory by not including a second copy of the initializer just for run-time initialization purposes.

Constant initializers for globals

CMOC considers a global variable's initializer to be constant if it is made of:

- numerical literals;
- constant numerical expressions;
- string literals;

- the address of a variable;
- a function name;
- an array name
- an arithmetic expression (using +, -, *, / or %) involving only one variable and one or more constant numerical expressions;
- a sequence of constant initializers between braces.

Array sizes

One must be careful when specifying an array size using an arithmetic expression, like this:

```
char a[256 * 192 / 8];
```

This will generate an "invalid dimensions" error message. This is because the three numerical constants are of type `int`, which means they are signed integers. In 16-bit signed arithmetic, 256 * 192 is -16384, which gets divided by 8, which is -2048. This size is rejected by the compiler, which only accepts array sizes between 1 and 32767 bytes inclusively.

The fix is to force the expression to be evaluated with unsigned integers:

```
char a[256U * 192U / 8U];
```

This will be 6144 bytes, as intended.

Compiling a ROM image for a cartridge

To help support the ROM cartridge format, CMOC supports directives that allow telling it to assemble the code, the string, long and real literals, and the read-only global variables at the typical CoCo cartridge ROM address of $C000, while mapping the writable global variables at a RAM address like $3800.

This is achieved by using the four `#pragma` directives that appear in this example:

```
#pragma org $C000
#pragma data $3800

int f() { return 42; }

const int g = 100;
const unsigned char byteArray[3] = { 11, 22, 33 };
const char text[] = "hello";

int anotherWritableGlobal;

int main()
{
```

```
    anotherWritableGlobal = 99;
    return 0;
}
```

`g` is read-only because it is of a constant type. `byteArray` and `text` are read-only because they are arrays whose elements are of a constant type.

These three variables are thus automatically put in the read-only section, next to the code. This means they will be part of the cartridge ROM, instead of using up RAM space.

In the case of `text`, the use of the empty brackets is necessary for that variable to be seen as read-only. Declaring this variable as `const char *text` would lead the compiler to see it as writable: the `text` pointer itself can be modified, although the characters it points to cannot.

Using `sbrk()` can be dangerous when the writable data section is not in the default position.

After compiling the program, the .bin file normally contains a single contiguous block of code. This block must be extracted from the .bin file and, for a test with the XRoar emulator, it must then be padded at the end with enough bytes so that the total file length is a multiple of 256. The following Perl script does this:

```
#!/usr/bin/perl
sysread(STDIN, $h, 5) == 5 or die;
sysread(STDIN, $rom, 0xFFFF) > 0 or die;
my $romLen = length($rom) - 5;
binmode STDOUT or die;
print substr($rom, 0, $romLen);
my $extra = $romLen % 256;
print chr(0) x (256 - $extra) if $extra;
```

The script, in a file called bin2cart.pl, can be used this way:

```
perl bin2cart.pl < foo.bin > foo.rom
```

This ROM image can be tested in the XRoar emulator this way:

```
xroar -machine cocous -cart-autorun -cart foo.rom
```

Note that XRoar requires the image file to have the .rom extension.

In a cartridge-based program written as above, the CoCo's 60 Hz IRQ interrupt is not enabled, so Basic's TIMER counter (at $0112) does not get incremented. To enable the IRQ in such a program, put this at the beginning of the `main()` function:

```
asm
{
    // We come here from a JMP $C000 in Color Basic (normally at
```

```
    // $A10A in v1.2). At this point, the 60 Hz interrupt has
    // not been enabled yet, so enable it.
    lda     $FF03   // get control register of PIA0, port B
    ora     #1
    sta     $FF03   // enable 60 Hz interrupt

    // Unmask interrupts to allow the timer IRQ to be processed.
    andcc   #$AF
}
```

Finally, it is preferable to use the command-line options `--org` and `--data`, instead of `#pragma org` and `#pragma data`, when developing a program made of multiple C files. The two command-line options should only be passed to the compiler invocation that links the executable from the object files.

Enumerations (enum)

Enumerations are supported, with the restriction that an enum must be declared at the global scope. In particular, an enum with an enumerator list (e.g., `enum { A, B }`) cannot be used in the formal parameter of a function, nor as its return type.

An enum can be anonymous (e.g., `enum { A, B }`) or be given a name (e.g., `enum E { A, B }`).

Each enumerated name can have a specified value, e.g., `enum { SCREEN = 0x0400, BIT5 = 1 << 5 }`. Such a value must be a constant expression.

Floating-point arithmetic

The `float` and `double` keywords and floating-point numeric literals (e.g., `1.2f` or `-3.5e-17`) have been supported since version 0.1.40, but **only under the Color Computer's Disk Basic environment**.

A warning is issued when the `double` keyword is encountered, so the user knows not to expect double-precision. There is also a warning when a numeric literal does not use the `f` suffix, which specifies that the literal is single-precision. There too, double-precision must not be expected. It is recommended to code programs using the `float` type and the `f` suffix.

The compiler will fail to compile a program that uses floating-point arithmetic when a platform other than Disk Basic is targeted.

CMOC's printf() function supports the %f placeholder, but as of version 0.1.40, it does not support the width or precision parameteres of %f (e.g., "%7.3f").

The `<cmoc.h>` header file provides functions `strtof()` and `atoff()` to convert an ASCII decimal representation of a floating-pointer number into a float value, as well as `ftoa()`, to convert a float value into an ASCII decimal representation.

Function Names as Strings

The constants `__FUNCTION__` and `__func__` can be used to refer to the current function's name:

```
printf("Now executing %s().\n", __func__);
```

In the global namespace, these identifiers give the empty string.

Detecting null pointer accesses at run time

Accessing a value through a null pointer is a common bug. To help detect such accesses, CMOC has the –check-null command-line option, which adds run-time checks before every use of a pointer and every array element access.

By default, the handler that is invoked when a null pointer access is detected is a no-op. The program should start by calling the `set_null_ptr_handler()` function to set up a handler that will receive the address of the failed check as an argument. For example:

```
#include <cmoc.h>

struct S { int n; };

void nullPointerHandler(void *addressOfFailedCheck)
{
    printf("[FAIL: %p]\n", addressOfFailedCheck);
    exit(1);
}

int main()
{
    set_null_ptr_handler(nullPointerHandler);
    struct S *p = 0;
    p->n = 42;
    return 0;
}
```

This program will fail and display an address. One can then look up this address in the .lst listing file generated by CMOC to determine in which function that null pointer was detected.

Using this option incurs a performance cost, so it is only recommended during debugging. An alternative is to define an `assert()` macro that expands to nothing when `NDEBUG` is defined.

Detecting stack overflows at run time

Runaway recursion or excessively long local arrays can cause corruption that is difficult to investigate. To help detect stack overflows, CMOC has the –check-stack command-line option, which adds run-time checks at the beginning of each function body.

When passing –check-stack to CMOC, the program should start with a call to `set_stack_overflow_handler()` that installs a handler. This handler receives two arguments: the address of the failed check and the out-of-range stack pointer. The handler must not return. For example:

```
#ifdef _CMOC_CHECK_STACK_OVERFLOW_
void stackOverflowHandler(void *addressOfFailedCheck, void *stackRegister)
{
    printf("[FAIL: %p, %p]\n", addressOfFailedCheck, stackRegister);
    exit(1);
}
#endif
void recurse() { recurse(); }
int main()
{
    #ifdef _CMOC_CHECK_STACK_OVERFLOW_
    set_stack_overflow_handler(stackOverflowHandler);
    #endif
    recurse();
    return 0;
}
```

This program will fail and display two addresses. One can look up the first one in the .lst listing file generated by CMOC to determine in which function that stack overflow was detected.

Using this option incurs a performance cost, so it is only recommended during debugging.

The preprocessor identifier `_CMOC_CHECK_STACK_OVERFLOW_` is defined by CMOC when –check-stack is used. This identifier can be used to exclude stack check code when not needed.

By default, CMOC allocates 1024 bytes for the stack on a CoCo (256 bytes on a Vectrex). In a program that is compiled with the default layout, the stack is preceded by the memory managed by sbrk() (see elsewhere in this manual).

Note that this feature is not usable under **OS-9**, where stack checking is automatic and uses a different mechanism.

Specifying the space allocated to the system stack

Option `--stack-space=`*N* (with $N > 0$) can be used to specify the number of bytes (in decimal) to be reserved to the stack. This affects the quantity of memory available to sbrk() and the stack checking done by `--check-stack`. Use this option when the program uses lots of stack space. Note that this option is not permitted when targeting the Vectrex.

The stack space can also be specified with `#pragma stack_space` *N*. The command-line option has precedence over this pragma.

Specifying the stack space must be done when compiling the C file that defines main(). It has no effect when compiling a file that does not define main(), nor when calling CMOC to perform linking on a list of object files.

Note that this feature is not usable under **OS-9**, where stack checking is automatic and uses a different mechanism.

OS-9 stack checking

When targeting OS-9, by default, a stack check is performed upon entering each function, unless that function is assembly-only.

The stack check verifies that there will be at least 64 bytes of free stack space once the function will have allocated stack space for its local variables. If this check fails, a "stack overflow" error message is printed in the standard output and the program exits.

To change this 64 to another (non-zero) value, use the command-line `--function-stack` option. Pass 0 to this option to disable the stack check.

Single-execution programs

If a program to be run under Disk Basic is intended to be executable only once, i.e., that reloading it or rebooting the CoCo is required to run it again, then the code that initializes the writable global variables is not needed upon entry to function main().

This routine is assembled at the very end of the code section. When the program specifies `#pragma exec_once`, then the "program break" used by sbrk() (see the previous section) is placed at the start of the routine. This makes the memory of that routine available to be allocated by sbrk().

Calling convention

CMOC follows the C convention of passing the parameters in the stack in the reverse order.

The caller pops them off the stack after the call.

An argument of type `char`, which is signed, is promoted to `int`.

An argument of type `unsigned char` is promoted to `unsigned int`.

The return value must be left in B if it is byte-sized or in D if it is 16 bits. If the return value is a struct, a long, a float or a double, then the return value must be stored at a location whose address is received by the function as its first (hidden) parameter.

The body of a CMOC-generated function preserves registers U, S and DP. It is allowed to modify A, B, X and CC.

Under OS-9, CMOC uses Y to refer to the data section of the current process. Any code that needs to use Y must preserve its value and restore it when finished. For portability, this rule should also be observed on platforms other than OS-9.

The compiler's low-level optimizer may emit code that uses Y when targeting a platform other than OS-9.

The called function does not have to return any particular condition codes.

Register DP is not used or modified by CMOC-generated code.

A CMOC-generated function uses a stack frame if the function receives one or more parameters or has local variables, and is not defined with the `asm` keyword (see the *Assembly-only functions* section elsewhere in this manual). Register U is used as the stack frame pointer. Such a function starts with these instructions:

```
PSHS    U
LEAU    ,S
```

an ends with these:

```
LEAS    ,U
PULS    U,PC
```

Calling a program as a DEF USR routine

Typically, such a program gets loaded at the end of Basic's RAM area, i.e., `&H7A00`. Pass –org to the compiler to position the program at such an address. In the Basic program, reserve some high RAM to your CMOC program, i.e., `CLEAR 200,&H79FF`. Then use `LOADM`, then `DEF USR` to define a user routine that points to your CMOC program, e.g., `DEF USR5=&H7A00`. (The 5 in this example

can be a number from 0 to 9.) Finally, call the routine using the routine number and an argument, e.g., `R=USR5(1000)`.

Value returned by main()

Since CMOC 0.1.58, the main() return value, and the exit() argument, are guaranteed to be returned in D upon exiting the program, when compiled for the CoCo Basic environment or for the Dragon.

This can be useful to return a 16-bit address from one program for use by another program. Note that registers U and Y must be preserved (typically with PSHS and PULS) when calling another program.

Compiling a program executable by the DOS command

Disk Basic 1.1 has a DOS command that loads track 34 at $2600 and if the contents start with the bytes 'O' and 'S', the interpreter jumps at $2602.

CMOC can accommodate this with its –dos option. It implies –coco, it uses $2600 as the code origin, and it generates the "OS" marker at that address.

The output is still a .bin file. CMOC comes with a command-line tool called `install-coco-boot-loader` that takes this file and copies it to track 34 of a DECB disk image. For example:

```
cmoc --dos prog.c
install-coco-boot-loader foo.dsk prog.bin
```

This tool will fail if track 34 is already occupied. It has a `--force` option that will install the boot leader despite that.

To uninstall the boot loader, use the `--uninstall` option:

```
install-coco-boot-loader --uninstall foo.dsk
```

C Preprocessor portability issues

For maximum portability, a CMOC program should not use the C Preprocessor's "stringification" operator (#), e.g.:

```
#define f(x) #x
```

When this feature is supported by cpp, the expression `f(foobar)` yields `"foobar"`, including the double quotes.

When this feature is not supported, the pound sign is left as is, so CMOC will see `#foobar`, which will typically give a syntax error. This has been observed on the Macintosh (Darwin) as of May 2017.

Also to be avoided is the use of C++-style comments on a line to be processed by cpp, e.g.:

```
#define foo 42  // blargh
```

Some preprocessors may leave the comment instead of stripping it away, so the following should be used instead:

```
#define foo 42  /* blargh */
```

License

EXCEPT FOR THE FILES UNDER src/usim-0.91-cmoc in the CMOC source archive, CMOC is distributed under the following license:

This program is free software: you can redistribute it and/or modify it under the terms of the **GNU General Public License** as published by the Free Software Foundation, **either version 3 of the License, or (at your option) any later version.**

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.