John's Goodies
Cross-compiling gcc C programs for z/VM CMS
Version 1 Release 4

Document Number jh11-00049-02

June 25, 2016 2:21 p.m.

John P. Hartmann

| **Third Edition (June 2016)**

| This printing applies to Version 1 Release 4 Modification level 0 of the cross compiling package for gcc to CMS.

| Changes since the March 2015 printing are marked by a vertical bar in the margin.

. Changes since the October 2011 printing are marked by a period in the margin.

Contents

Preface

This document describes procedures for generating old-fashioned object modules for CMS or MVS from C programs that are cross-compiled on a UNIX platform.

A public domain C library and supporting macros and EXECs are also included.

Programs can be linked with the C library either in the traditional UNIX way or to the CMS interface:

```
int main(aint argc, char ** argv)
int cmsmain(struct eplist * epl, char (* pl)ff8")
```

The target audience at this stage includes CMS XA and z/CMS. CMS/370 as implemented in O'Hara's six-pack system is a follow-on, if at all possible.

The advantage of this approach over other free CMS C compilers is several fold:

- A current compiler supported by Boeblingen is used without modification.
- Compile performance is improved over running a compiler on an emulated platform, such as z/PDT or Hercules.
- Dual-use of the generated code with CMS and UNIX is automatic unless CMS interfaces are used directly, in which case a certain amount of `#ifdef`-ing will be required.

Overview

This document first describes the general work flow. Following chapters describe implementation details that depend on the platform used:

- z/Linux, where the native compiler is used.
- Other UNIX systems, for example Linux or FreeBSD, typically on Intel or AMD processors. For these, you must build a cross-compiler using crosstool NG `http://crosstool-ng.org/`. This worked for me on FreeBSD and Ubuntu; it was a lot simpler than trying to configure and build a cross compiler from source and it even provides a complete tool chain so you can make a z/Linux executable. Appendix B, "Building a cross compiler with CT-NG" on page 31 shows how I installed a cross compiler using `CT-NG`.

Terminology

In general, commands set in lower case refer to UNIX objects, whereas upper case names refer to CMS commands. Thus, `maclib` runs on UNIX and generates a macro library that you can upload, whereas `MACLIB` refers to the CMS command.

Distribution

The bootstrap code, the library, and utilities are stored in a number of repositories on github below `https://github.com/jphartmann/`:

`cmslib-exec`	Documentation (this publication); `MACLIB` for use in the assembly step; a `TXTLIB` that contains the library; and a VM archive that contains supporting EXECs.
`cmslib`	Source code for the library.
`gas2asm`	The utility to convert a gnu assemble file to HLASM.
`CMS-TXTLIB`	`maclib` and `txtlib` utilities for UNIX.

You can clone the repositories or download a zip of the contents at `https://github.com/jphartmann/`.

Assumptions

I assume you know how to move files from the hosting platform to the target one. Most likely you use TCP/IP or a hot reader.

I also assume you know CMS and how to use it.

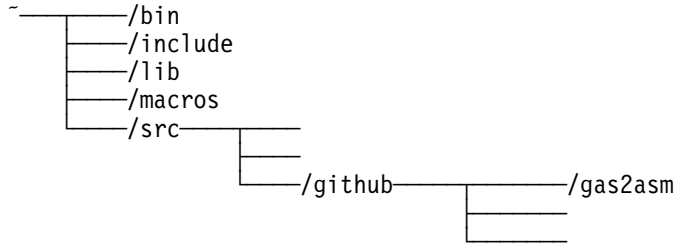
Finally, you or someone you know must be able to compile the `gas2asm` utility using the `make` utility.

You are using a Linux platform. Some basic UNIX skill is required.

Directory structure.

I use the following directory structure and you will likely be forced to adapt it.

Figure 1. Directory tree



The contents of the directory tree are:

`bin`	Linux executables. These are often created by `make install` from directories in the source tree. To be effective, `bin` must be mentioned in your `PATH` environment variable.
`include`	Header files (extension `.h`) for the interface to libraries that are installed in `/lib`. This directory must be specified by the `-I` flag to gcc.
`lib`	Libraries (`.a` or `.so` extensions) that are used to link executables. To be effective, `lib` must be mentioned in your `LD_LIBRARY_PATH` environment variable.
`macros`	`SYS1.MACLIB` if you are able to run HLASM on your UNIX workstation.
`src`	What is needed to build executables and libraries. You might store your own projects here. While not required, you could create a directory `github` to hold projects you have cloned from `github`. `gas2asm` would then be one of such projects.

Prerequisite software

You need a development environment on your workstation that includes a C compiler (gcc or clang to taste), gnu make, and bash. All of this is normally installed on just about any Linux distribution.

With FreeBSD and likely all the other BSD variants, you need to install `gmake` and create an alias that points `make` to `gmake`.

If you wish to enhance or fix `gas2asm`, you will need `flex`.

If your Linux is not z/Linux, you must also install a cross compiler as described in Appendix B, “Building a cross compiler with CT-NG” on page 31.

You need `git` if you wish to work with my repositories.

If it does not work

Let me know if you have done as described in this document. However, I do not wish to know about trouble with your own shell scripts.

If it should use autoxxx and packages and whatnot

Feel free to make a pull request for `gas2asm` when you have converted it to `CMake`. No point in trying `automake`.

In short

If your desire is to create an OS/360 object module from a C program using gcc on UNIX, all you need is:

1. A compiler targeted for s390 or s390x. You already have one if your Linux is for z.
2. `gas2asm` to convert gas to assemble.

With those two, you can compile metal C (that is, C programs that run on almost the bare metal), but you will need something to set up at least the stack before calling `main()`.

3. You might consider using `FPLGCC TXTLIB` to obtain `run390.assemble` to allocate the stack frame.
4. Link with `FPLLINKM` to avoid the C library.

All the rest described herein is optional. Use it if you like it; substitute your own if you do not.

Document contents

Chapter 1, “Restrictions” on page 1 tries to dampen your enthusiasm and to set realistic expectations.

Chapter 2, “Cookbook” on page 3 shows how, but not why. Refer to the following chapters for design considerations.

Chapter 3, “General work flow of compiling and generating a module” on page 6 contains an overview of the compilation and build process.

Chapter 4, “Writing *CMS/TSO Pipelines* stages in C” on page 11 describes now to use C programs with *CMS/TSO Pipelines*.

Chapter 5, “Running a gcc390 module” on page 12 describes the invocation parameters for a C program that uses the C library.

Chapter 6, “CMS specifics” on page 15 describes header files and functions in support of `CMS`.

Chapter 7, “Implementation” on page 21 describes the implementation of the CMS functions.

Chapter 8, “Reference information” on page 25 contains reference information, such as command formats.

The appendices contain reference information, in particular:

- Appendix A, “Distribution and installation” on page 29 describes the development environment.
- Appendix B, “Building a cross compiler with CT-NG” on page 31 shows how to install `CT-NG` and use it to build a cross compiler.
- Appendix C, “Notes” on page 33 contains various notes and explanation of attempts that did not pan out, most notable an attempt to convert a Linux executable to a CMS command.

Document status

This document is itself very much work in progress; and what it describes is to some extent also work in progress. That said, a REXX interpreter for Linux was ported to CMS with reasonable limited effort: a few CMS built-in functions were added as was CMS command processing. The Public Domain C library for MVS and CMS was more effort: a number of bugs and omissions were fixed and native CMS/XA support replaced OS simulation for the I/O library.

Ebbeløkke,

June 25, 2016 2:21 p.m.

j.

Chapter 1. Restrictions

What gcc390 can do

While the compiler and the `gas2asm` conversion utility, of which you will get to know a lot more, are target agnostic, at least as long as the target supports the OS/360 object format, you will be building your program for one of three environments, as summarised in Figure 2.

Figure 2. Execution environments for cross compiled programs

Header files	Entry point	Link method	Sample program	Description
`stdio.h` `stdlib.h`	`main`	`FPLLINK`	`hello.c`	A program that is ported from a `UNIX` environment with minimal fuss. Your program starts at the `main()` function with the usual UNIX arguments, `argc` and `argv`
`cmsbase.h`	`cmsmain`	`FPLLINKM`	`minimal.c`	A program that is written specifically for `CMS`. You receive `CMS` parameter lists and deal directly with `CMS`.
`cmspipe.h`		`LOAD`	`fplhello.c`	A program specifically designed to run in the *CMS/TSO Pipelines* environment. Your module is linked into a filter package or into the main pipeline module. The entry point is identified in an entry point table just as if it had been written in PL/j. Each C language stage is provided its own stack frame. We refer to each invocation as a thread, but this has no relation to `CMS` Application Multitasking, Language Environment, or anything else.
				Roll your own. You might call C code from your assembler modules.

Thread local data

The following data are thread local, that is, they are not allocated in static storage as is customary for simplistic library implementations:

- `errno`
- file descriptors
- the list of `atexit()` routines.

What gcc390 cannot do

Apart from **CMS Application Multitasking** (CMS MT), which we refuse to touch even with a ten foot pole, we are not aware of anything that "simply does not work"; but there are many differences between UNIX and CMS; and they do matter.

- UNIX modules are not reusable; they depend on a fresh load of their writable data for each invocation. Thus, in general, you cannot `NUXCLOAD` such a module and

expect that it behaves in a reproducible fashion; nor can you *à priori* expect that *CMS/TSO Pipelines* filter packages containing UNIX modules will work.

The requirement from a CMS perspective boils down to this: The C program must initialise all storage not allocated on the stack, or it must not depend on such storage having been initialised. Essentially, initialisation must be by executable instructions rather than by preassembled values for data items whose value change during execution.

You can inspect the load map to get an indication of potential trouble. Preassembled values will generate a section of private code while uninitialised variables are stored in a named common.

If you write the program, you can take care of your own code, but you cannot control what the library does.

To understand better what is going on, you will need to know a few points about UNIX design:

- A UNIX executable is mapped into memory by the paging system; it is not, as on CMS, read into storage and left on its own.
- A UNIX executable (the underlying file) contains a number of segments (nothing like VM segments), which are interesting when trying to run the code.

  The module is also described by a number of sections (the linker's view).
- Three types of segments (or types of storage if you like) are used during execution:
  - Text segment(s) contain executable code and read only constants. Text segments are reentrant and are shared even between users running the same underlying executable file. Such a section is mapped to an RSECT
  - Data segment(s) contain initialised writable storage. These segments are loaded private to the process that execute the module. They are reloaded if the module is executed a second time. Such a section is mapped to a CSECT
  - Uninitialised storage (sometimes called bbs) is allocated dynamically by the loader and initialised to zero. Such a section is mapped to a named COM.

Even though the gas2asm utility knows of these attributes and forwards them to CMS as macros, the CMS module format and the CMS loader are clearly not capable of doing anything with it. (However, IBM's CMS C and Language environment does because it uses program objects.)

On CMS virtual machines

CMS virtual machines are IPLed in ESA/390 mode and remain in that mode, while z/CMS switches to z/Architecture mode.

This means that the CMS virtual machine is limited to the architecture described in SA22-7201-08, specifically:

- Only the "N3" instructions from the original z/Architecture are available.
- Long displacements are not supported; they are quietly ignored. That is, those instructions are RXE form, *not* RXY as you might expect. Be sure to tell HLASM that you wish to use MACHINE(Z) rather than take the default.

Chapter 2. Cookbook

This chapter (with its references) contains everything you need to do to cross compile for CMS if you do not have HLASM or equivalent available on your workstation and you are happy with the default installation locations. Subsequent chapters amplify and explain.

The examples in the following sections show the commands you must issue.

If your UNIX is not z/Linux

You must install a cross compiler. This procedure has been performed on `FreeBSD` and `Ubuntu`. Other Linux distributions are likely to work similarly. For Apple, Arm, and all the rest, please report success or failure, as appropriate so the this documentation can be updated.

Refer to Appendix B, "Building a cross compiler with CT-NG" on page 31 for a sample session. Be sure to configure the tool to install in the root `$HOME`.

Download the tool to a convenient directory that is out of the way (`/tmp` works). Crosstool generates a lot of chaff; you definitely do not want to have it in your home directory or your source directory. Perhaps you have `/usr/junk`; that would also be a candidate.

Once you have downloaded `ct-ng`, position yourself in the directory where you downloaded the tool and follow the example in Appendix B, "Building a cross compiler with CT-NG" on page 31. The commands you will be issuing are:

```
tar -xf crosstool-ng-1.20.0.tar.bz2
cd crosstool-ng-1.20.0
./configure --prefix=${HOME}
make
make install
ct-ng s390x-ibm-linux-gnu
mkdir .backtrace
unset LD_LIBRARY_PATH
ct-ng build
```

Building the cross compiler is likely to take half an hour or more. Perform the following steps in the meanwhile. (You clearly cannot do a sample command until the cross compiler is installed.)

Customise your Linux ID

Environment variables

You need to define some environment variables.

PATH	You need to add &tilde/bin to your path, as this is where gas2asm &c will be installed.
VMHOST	To upload ASSEMBLE files automatically this environment variable must be set to the host name or IP address of your VM system. You must also customise your .netrc file as described below.

.netrc

Leaving aside any discussion on the security issues, you must configure the ftp connexion to your VM system. Mine is:

```
machine cphart login john password xxxx macdef init
cd sfs:john.stage

```

Note that a blank line terminates the init macro.

Clone and build gas2asm

```
mkdir ~/src
mkdir ~/src/github
cd ~/src/github
git clone https://github.com/jphartmann/gas2asm.git
cd gas2asm
make
make check
make install
```

This installs the utilities and copies macros that are required for assembly to ~/macros. Refer to the README in the package and “gas2asm—Convert .s to HLASM” on page 26.

Clone cmslib-exec and upload VM objects

```
cd ~/src/github
git clone https://github.com/jphartmann/cmslib-exec
cd cmslib-exec
```

The files you just clones are:

fplg2a.maclib	A CMS MACLIB. Upload binary F(80).
fplgcc.txtlib	A CMS TXTLIB. Upload binary F(80).
fplgcc.vma	A VM archive containing EXECs to link the compiled code. Upload binary F(80).
fplgcc.tar	C header files, a make file, and a sample (minimal). The archive should be installed in the directory above the directories that contain your C projects, for example &tilde/src. `tar -xf fplgcc.tar -C ../..`

Build one of the samples

The reason for building a sample is really not to test that the sample works, but to test that the cross compiler is installed correctly.

On your workstation

1. Navigate to the appropriate sample directory. We shall take the simplest.
2. Build the sample and transmit the object module to CMS

   ```
   cd
   cd src/gcccms/samples/minimal
   ```

   The output is rather a lot here it is, but truncated in the right margin:

   ```
   [/home/john/src/github/cmslib/src/samples/minimal] make
   PATH=/home/john/x-tools/s390x-ibm-linux-gnu/bin s390x-ibm-linux-gnu-cc
   PATH=/home/john/x-tools/s390x-ibm-linux-gnu/bin s390x-ibm-linux-gnu-cc
   gas2asm < z/minimal.s | asmxpnd > z/minimal.assemble
   (cd z && cat minimal.assemble >@minimal.assemble)
   rm z/minimal.s
   [/home/john/src/github/cmslib/src/samples/minimal] make up
   (cd z && cat minimal.assemble >@minimal.assemble)
   /bin/echo -e "site fix 80\nlcd z\n put @minimal.assemble minimal.assem
           |ftp cphart
   Local directory now /home/john/src/github/cmslib/src/samples/minimal/z
   [/home/john/src/github/cmslib/src/samples/minimal]
   ```

   The output lines show:

   a. The dry run of the compiler to test `printf` arguments.
   b. The actual compilation.
   c. Conversion from `gas` to `assemble`.
   d. Gather all `assemble` files into a stacked one. Here we have but a single file.
   e. Upload to CMS.
   f. `make` deletes a temporary file. To retain it, issue `make z/minimal.s`.

The procedure on CMS is equally painless. Just assemble and link the module:

```
global maclib fplg2a
Ready; T=0.01/0.01 18:33:44
hlasm minimal (term
 Assembler Done No Statements Flagged
Ready; T=0.05/0.08 18:33:53
fpllinkm minimal
RC=0;
Ready; T=0.03/0.04 18:34:03
minimal
Ready(00042); T=0.01/0.01 18:34:10
```

As you see, we have the answer to the universe and everything.

Chapter 3. General work flow of compiling and generating a module

This chapter describes how a program is cross compiled for CMS and then linked. While the tasks performed on the workstation have been automated by make files, this chapter describes each individual step, as if you were to perform it by hand.

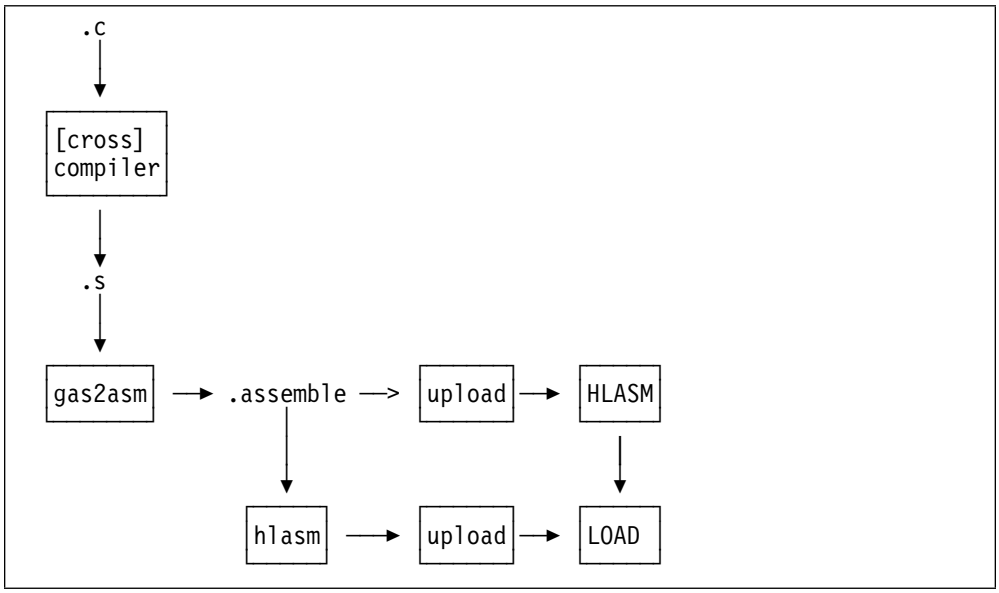
Compilation of a simple C program on the workstation feels as if it is a single step operation, but in fact several steps are performed by the GNU compiler collection to produce an executable module:

1. The C preprocessor embeds files that are specified by `#include` and performs substitution of macros (`#defines`).
2. The main compiler produces a symbolic assembler program in a format suitable for the GNU assembler.
3. The GNU assembler (`as`) produces an object deck.
4. The loader (`ld`) resolves library references and produces the final command module in what is called Executable and Linkable Format (ELF).

As CMS does not know of ELF objects, we break this procedure after step 2. Then we perform these steps:

- Convert the `gas` input file to a file usable to HLASM (.assemble).
- Generate the object module by HLASM (or equivalent). This step can be performed either on the work station (if you have a suitable assembler), or with HLASM on CMS.
- Use the `FPLLINK EXEC` to produce the executable CMS `MODULE` file.

The flow is depicted below.



If you are too busy to read the following, do take careful note that the compiler options

```
-m31 -march=g5 -fexec-charset=IBM-1047
```

are all *sine qua non*. Also take note that it is all embedded in a makefile if you use the recommended setup.

Compile step

The purpose of the compile step is to obtain an assembler file. That is, you do not run the final pass to generate an elf object file, as CMS has no idea how to deal with such a beast.

The output file has the file type `.s`; it is in GNU Assembler form.

Required compiler options

Required flags are:

- -S

  (Upper case.) To specify that the compiler stops after the assembler program is generated. *Sine qua non.*

- -nostdinc

  Do not search the standard include path. If you did, you might get stuff resolved for the compile platform rather than the target one. There will likely be conflicts, even with the cross compiler's standard files, because the public domain C library has a different, *e.g.,* `stddef.h` than the one supplied with gcc.

- -I

  Probably many of them, to get to the library header files. You cannot use the header files for the compile platform as they are unlikely to have that same naming conventions as the CMS library uses.

- -fexec-charset=IBM-1047

Specify that you want EBCDIC conversion of character strings and constants. This is crucial as it converts character literals to the corresponding EBCDIC value. You can specify anything `iconv` supports; you need to install further code pages if IBM1047 does not work either. Or you may desire a different code page, perhaps the one for your native language.

- -Wno-format

  Because the format checker assumes that the output is in ASCII. A slew of confusing warnings would ensue if this option were to be enabled. But see "Enabling warnings" below.

- -m31

  Generate 31-bit code. This is crucial unless, of course, you are generating code for z/CMS in 64-bit mode (and then you are even more on your own).

- -march=g5

  Contrary to gcc documentation this option is required to prevent the compiler generating long relative calls to library routines. These instructions are not available in a virtual machine running ESA mode, even if CP runs z/Architecture one the latest IBM processor. (And if you try z/CMS you will find that the loader does not understand the `ESD` entries for such external references.)

- -fno-use-linker-plugin

  Disable a check for something that is never going to be used. This option was added around gcc 4.6.0; omit it if gcc complains about it.

A few other options you might consider:

- -fverbose-asm -g

  The compiler produces more information in the output file. `gas2asm` uses this to include the source C code with the generated code. This may be useful for debugging.

- -save-temps

  Leave the output from the C preprocessor intact. This option may be helpful to debug your macros and may help to spot unmatched parentheses, particularly if you have an editor that colours the syntax.

In addition you must define some macros that specify that you are compiling for z/VM. Consider this boilerplate.

```
-D__ZVM__ -D__CMS__ -U__gnu_linux__
```

Enabling warnings

Wise girls use `-Wall -Werror` to ensure that their programs do not have obvious deficiencies.

However, `-fexec-charset=IBM-1047` gets in the way of verifying `printf` parameter lists.

Thus, you may consider doing a dry run by compiling for an ASCII target using `-Wall -Werror` before compiling for EBCDIC with `-Wno-format` and `-fexec-charset=IBM-1047`. I would not dream of anything else.

Convert GNU Assembler to High Level Assembler

For this step you need a couple of utilities that I provide in source form.

```
https://github.com/jphartmann/gas2asm
```

The procedure is straightforward (here written as a REXX statement):

```
'gas2asm <'fn'.s | asmxpnd >'fn'.assemble'
```

Or as a `make` rule:

```
%.assemble: %.s
      gas2asm < $< | asmxpnd > $@
```

Refer to "`gas2asm`—Convert .s to HLASM" on page 26 for details on command flags.

Assemble step

If you have a cross-assembler you will likely need to modify the flags in `fplgcc.makefile` to suit your tastes. Otherwise upload the source and run it through HLASM.

In either case, the resulting TEXT deck should wind up on CMS.

You need a small set of macros (`FPLG2A MACLIB`).

Link step

This is always done on CMS.

Use `FPLLINK` to generate a module that uses that UNIX `main()` entry conventions; and use `FPLLINKM` for a program that uses the CMS `cmsmain()` entry conventions. Refer to "FPLLINK—Link a CMS module" on page 26. Undefined symbols invariably lead to a branch to location 0.

Dealing with long external symbols

C symbols are mixed case and can be of arbitrary length. HLASM symbols are caseless, normally folded to uppercase, and at most 64 characters (local symbols) or eight characters for external symbols when the OBJ object module format is used. (The GOFF object module format does not impose the latter restriction).

`gas2asm` changes periods (.) in symbols to number signs (#), so at least you do not have to worry about that.

You have two ways to deal with symbols that do not comply with these restrictions:

- Use the C preprocessor to redefine a long symbol to a unique shorter one. If you are importing a library, you may wish to write C wrappers for the library routine to include a set of `#define`s before including the distributed C code, for example:

  ```
  #define deflateInit2 dflini2
  #include "deflate.c"
  ```

Since it is performed by the C preprocessor, this approach can deal with symbols that are longer than 64 characters and symbols that fold to the same uppercase. (It can deal with anything.)

- Note the hashing performed by `gas2asm` and use the hashed symbol instead. This is, to say the least, tedious. However, if the symbols are used only between C programs, hashing cancels out because `gas2asm` produces a reproducible hash for any particular symbol.
- Use an assembler that supports long and case sensitive symbols. He says, somewhat tongue-in-cheek. With that, you will need to run the preloader on CMS for it to perform its own mangling of long external symbols.

Chapter 4. Writing *CMS/TSO Pipelines* stages in C

CMS/TSO Pipelines 1.1.12/000A supports cross compiled C code as a first class citizen. The program may be linked into the main pipeline module or it may be supplied in a filter package, just as any other program. That is, *CMS/TSO Pipelines* code allocates stack space and manages a stack for each stage.

There are at least two distinct motivations for writing *CMS/TSO Pipelines* stages in C:

- To implement code of your own design. Typically, such a program will not require any library, as the entire *CMS/TSO Pipelines* environment is available to such a program.
- To interface to a C library, for example for compression or pattern matching. Apart from the particular library being ported, it may require routines from the C library; such routines must be linked statically with the library.

Entry conditions for a C language stage

The structures and entry points are declared in `cmspipe.h`

CMS/TSO Pipelines passes two parameters to the C program:

1. An opaque token representing the environment. This token must be supplied on all service calls to *CMS/TSO Pipelines*.
2. A pipeline parameters structure. This contains the information passed in registers 0 through 5 in the assembler interface.

Chapter 5. Running a gcc390 module

This chapter assumes that you have generated your module by FPLLINK.

Argument string in general

The executable is invoked just as any other CMS command, but the argument list is definitely unixy:

- Flags to control the library are specified by words of the form --fpl-<name>. These are recognised only at the beginning of the parameter list.
- Plain words will become separate entries in argv.
- Quoted strings, either single or double, become a single entry in argv.
- Redirection of the standard I/O files is specified by a subset of the shell features.

Flags to control the library

The flags can be specified at the very beginning of the parameter list:

--fpl-csl-error	Display callable services return codes and parameter lists after a call to a service has completed with a nonzero return code. Refer also to "tocsl—Interface to CMS callable service" on page 19.
--fpl-csl-verbose	Display callable services return codes and parameter lists after a call to a service has completed. While designed to debug the library, this option may prove useful when getting to grips with CSL and also if I/O redirection does not perform as you desire. Refer also to "tocsl—Interface to CMS callable service" on page 19.
--fpl-debug	Print more unspecified debugging information.

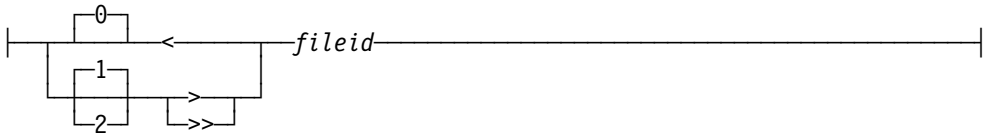
As these flags are processed after any I/O redirection on the command line, you cannot debug trouble with I/O redirection with these flags.

I/O redirection

The first three file descriptors are initially opened to the terminal, as on UNIX.

Use <, >, or >> to redirect the corresponding file.

redirect:

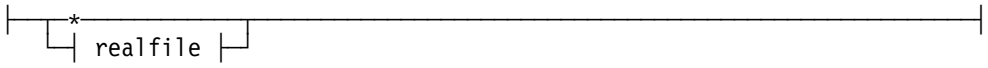


go390.c calls the standard library open() using the specified file name string.

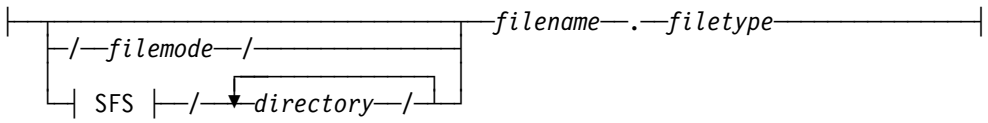
File identifiers

File identifiers specify the path to the file to be opened in I/O redirection, `open()`, `fopen()`, and `freopen()`. The syntax is borrowed from the CMS C library.

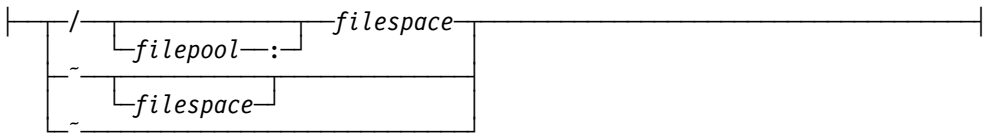
fileid:



realfile:



SFS:

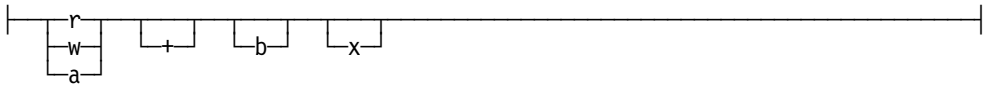


The file identifier `*` refers to the console of the virtual machine. It can be opened for read or append. If opened for write, it is appended to just the same.

Access modes

The second parameter to the f-style open functions specify the access modes as a string. The standard mandates this form:

mode:



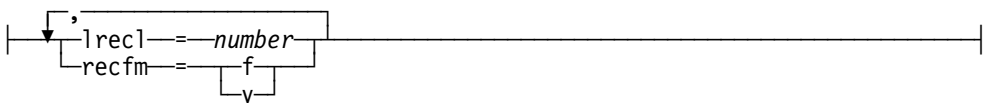
The mode flags and the associated device open flags are shown below.

r	O_RDONLY	Read file from beginning.
w	O_CREAT O_WRONLY O_TRUNC	Replace file from beginning.
a	O_CREAT O_WRONLY O_APPEND	Append to file.
b	O_BINARY	Binary access. Ignore record breaks in file.
+	O_RDWR	Open both for read and write.

The exclusive flag is not supported, as this function must be handled by `SFS` work units.

The mode string may be followed by a comma and CMS specific information:

cmsMode:



The CMS mode flags are specified as the third parameter to `open()`.

Examples: To count the number of bytes, words, and lines in the load map on mode a:

```
wc < /a/LOAD.MAP
```

Handling and reporting execution errors

This section deals with errors you discover.

Traceback

When you (or rather your program) diagnoses an error during execution, you might ask the question, how did I get here?

Call `__traceb` to obtain a stack trace. It prints a line for each active stack frame. If the programs have been compiled with the recommended options this includes source file and line identification.

Debugging

Well, yes, ahem, no; there is no debugger.

This may make it easier to pin down something that is not immediately obvious or to set breakpoints:

- If you compiled with optimisation, do without. The chance that this clears the problem is remote, but it gets you ready for the next item.
- Link the module absolute by specifying the option `NORLD` on `FPLLINK`:

  ```
  fpllink hello ( norld
  ```

  Now you can use the load map to determine the failing module easily. The assembler listing should then get you the failing instruction and if you compiled with `-fverbose-asm`, even the failing source code.
- The object code contains eyecatchers before each function. The function name is enclosed in “noses”, for example, `>>main<<`. Register 13 usually points to an address not far beyond this constant.
- Use CP TRACE to set breakpoints. With an absolute module, you can set them before running the command.
- A debugger is not entirely out of the question, but as such a beast would run in an adjunct virtual machine, it would be out of bounds for most. But decidedly fun to make.

Chapter 6. CMS specifics

This chapter lists CMS facilities you can access from C.

If you find no interface to a particular CMS function, you can always reach it through `__svc204()` once you have discovered the format of the parameter list.

CMS Data areas

So far we have structure definitions for the following CMS control blocks based on CMS 22 (z/VM 5.2). There is one `h` file for each data area.

In general, the structure has one member for each `DC/DS` instruction. The member name is the assembler label translated to lowercase. `EQUs` are gathered in an `enum` after the structure.

The h files are generated on CMS by an ever expanding pipeline refinery.

Figure 3. CMS data areas

`adt.h`	Active disk table entry.
`cmsstack.h`	Parameter list to put a line on the CMS console queue.
`eplist.h`	Extended parameter list. What a CMS command may find pointed to in general register 0.
`exsbuff.h`	Information returned by the DMSEXIST callable service.
`linerd.h`	The `LINERD` macro parameter list.
`nucon.h`	Nucleus constant area. The contents of the first page in storage. You get a pointer to this block by: `struct nucon * pnuc = (struct nucon *) 0;`
`scblock.h`	Subcommand control block.
`svcsave.h`	System SVC save area. There is one such for each CMSCALL active. The most current, that is, the running C program, is found in `currsave` in `nucon`.
`usersave.h`	The user save area. A CMS command is entered with general register 13 pointing to a user save area.

Other `#include` files

Figure 4. Other include files	
cmsbase.h	Declares low level assembler routines for storage management, CMS SVC 204 entry, and selected CP diagnose instructions.
cmspipe.h	Specifies the interface to *CMS/TSO Pipelines*.
cmssubcom.h	Declares functions to manage SUBCOM callbacks, *e.g.*, from a macro.
csl.h	Declares functions to call a function using type 1 calling conventions, in particular the CMS interface to callable services (DMSCSL).

Functions

This section describes functions that are specific to this CMS implementation and are for general use. Other functions are for the exclusive use by the C library.

__adtlkp—Return an active disk table entry

The __adtlkp function returns a pointer to the active disk table entry for the specified mode letter.

```
►►—struct adt *—__adtlkp—(—unsigned char *—mode—)————————————►◄
```

mode A pointer to the mode letter.

Declared in: cmsbase.h.

Return value: The return value from __adtlkp is a pointer to an active disk table entry, or NULL, if none exists for the specified mode.

Example:

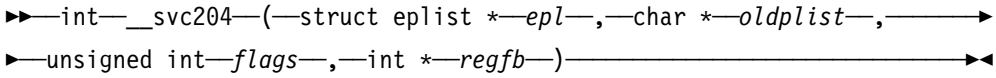
```
#include <cmsbase.h>
#include <adt.h>

struct adt * pa = __adtlkp("A");

if (pa) f->blksize = pa -> adtdbsiz;       /* Get disk block size */
```

Notes:

1. See PRINTBLK SAMPASM on the CMS source disk.

__svc204—Call a CMS service

Use __svc204 to issue calls to low level CMS functions that you cannot reach through library functions.



Declared in: cmsbase.h.

epl — A pointer to the extended parameter list, which contains the untokenised command string. Specify NULL for calls where no such parameter list is present.

oldplist — A pointer to a list of CMS parameter tokens. These are usually doublewords, but each function has its own idiosyncrasies. The parameter list is usually terminated in a "fence", which is eight bytes of binary ones. The macro FENCE is available to initialise a fence.

flags — The flags passed in register 15. in particular the rightmost byte contains the call type. Be sure to specify the COPY flag if the CMS function requires it.

regfb — A pointer to an array of two integers into which are store the contents of registers 0 and 1 upon return. Specify NULL if the values have no interest to you.

Return value: The return value from __svc204 is the contents of general register 15 upon return from CMS.

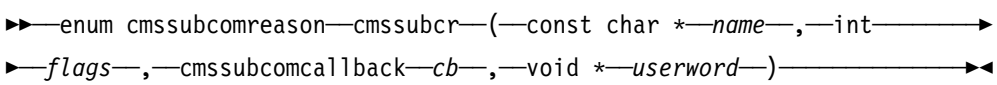
Example: To issue a string to a specified subcommand environment.

```
#include <cmsbase.h>
#include <eplist.h>

static int
subcom(struct evalblok * addr, struct evalblok * ev)
{
   unsigned char * string=(unsigned char *) ev->data;
   struct eplist epl={string, string, string+ev->length};
   int i;
   char pl[2][8]=
      {
         "        ", FENCE,
      };

   for (i = 0; 8>i && addr->length > i; i++)
      pl[0][i] = addr->data[i];

   return __svc204(&epl, pl[0], 0x2<<24, NULL);       /* 2 is SUBCOM */
}
```

cmssubcr—Enrol subcommand callback routine

Use cmssubcr to enrol a callback for a CMS subcommand. The specified function is called when the subcommand is issued an the runtime environment will accept a recursion.

```
typedef int (* cmssubcomcallback) (struct usersave * usave,
   struct eplist * epl, void * userword);
```



Declared in: cmssubcom.h.

name The name of the subcommand to be registered. Specify a character string that is eight bytes, padded if necessary.

flags Currently unused. Specify 0.

cb The routine to be called when the subcommand is issued.

userword Where you can pass your private pointer to the handler.

Return value:

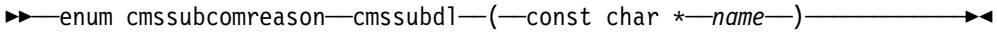
0 Done OK.
1 Name string is null.
2 Name string longer than 8 characters.
3 Handle pointer is null.
4 Callback pointer is null.
5 A handler for the name specified is already active.
6 Unable to allocate storage for control blocks.
7 Name contains bad character (SUBCOM return code 20).
8 CMS is unable to allocate storage (SUBCOM return code 25).
9 Undocumented CMS return code.
10 No subcommand registered by name specified.

Example:

```
enum cmssubcomreason rsn;
int varacc(struct usersave * usv, struct eplist * epl);

rsn = cmssubcr("EXECCOMM", 0, &varacc, gbl);
```

cmssubdl—Retract subcommand callback routine

Use cmssubdl to retract a handler set up by cmssubcr().



Declared in: cmssubcom.h.

sbx The handle returned by `cmssubcr()`.

Return value:

0 Done OK.
1 Name string is null.
2 Name string longer than 8 characters.
3 Handle pointer is null.
4 Callback pointer is null.
5 A handler for the name specified is already active.
6 Unable to allocate storage for control blocks.
7 Name contains bad character (`SUBCOM` return code 20).
8 CMS is unable to allocate storage (`SUBCOM` return code 25).
9 Undocumented CMS return code.
10 No subcommand registered by name specified.

Example:

```
cmssubdl("EXECCOMM");
```

oscall—Call function expecting OS linkage

Use `oscall` to switch to OS type 1 calling conventions for a function that accepts a variable length parameter list.

Declared in: `csl.h`.

f The address of the function to call.

count The count of arguments to the function.

`...` Parameters to the function. These may be integers or pointers, as required by the function.

Return value: The return value from `oscall` is the contents of general register 15 upon return from the routine.

Example:

Caveat Emptor: Do not pass structures to `oscall`; as the structure is effectively turned into a list of pointers by `gcc`. Pointers to structures are, of course, OK.

tocsl—Interface to CMS callable service

Use `tocsl` to invoke a CMS callable service. Specify the callable service and its parameters. `tocsl` returns the contents of general register 15; it is up to the individual routine whether this is a return code or not.

Some common `CSL` error code are diagnosed on standard output.

Declared in: csl.h.

count	An integer variable specifying the number of parameters following the routine name.
name	A pointer to an array of eight characters that specify the routine to call. The name must be padded with blanks on the right, if required.
...	Parameters. Each parameter must be a pointer. Thus, numeric arguments must be specified as the address of an int variable.

Return value: The return value from tocsl is the contents of general register 15 upon return from the routine.

Example:

```
static int
devclose(FILE * f)
{
   int rv, rsn;
   static const char commit[]="COMMIT";
   int clen=strlen(commit);

   tocsl(5, "DMSCLOSE", &rv, &rsn, f->token, commit, &clen);
   return rv ? -1 : 0;
}
```

Related information: You can log CSL activity to the terminal (this output cannot be redirected at the library level):

```
#include <csl.h>

__csldbg = __csl_verbose;             /* To show all CSL calls       */
__csldbg = __csl_quiet;               /* To turn it off              */
__csldbg = __csl_error;           /* To log details about CSL errors */
```

Refer also to “Flags to control the library” on page 12.

Chapter 7. Implementation

This chapter describes how the runtime library and the compiler executables fit into CMS.

Cross compiled C code can run in three distinct environments:

- As a CMS command. The entry point for a command is a library routine, `run390`. It allocates the runtime environment, including the runtime stack, sets up the arguments for `main` and calls the compiled code.
- As a subroutine of a CMS command that is not cross compiled. The caller is responsible for building the execution environment. The subroutine is likely to use a private calling convention.
- As a stage of a pipeline. *CMS/TSO Pipelines* supports cross compiled programs directly. Such programs are embodied in filter packages. In this scenario, we support multiple concurrent stages that are cross compiled and thus multiple concurrent runtime stacks.

Runtime environment

The runtime environment mimics in some respect the UNIX process image, but it is allocated within a piece of storage in the virtual machine. From the top (highest address) down, this area contain distinct sections, though not all sections are always present:

1. A thread global area. This area is always present; it is mapped by `cmsthreadglobal.h`.
2. The environment, unless it was suppressed at link time or as described here.

   The initial contents of the environment are obtained from the variables in the global variable group `ENVIRON`. It is stored in with the variables at high addresses and the vector of address below.

   A program that does not use the environment may expand the macro `GCC390_NO_ENVIRON()` in file scope (that is, not within a function):

   ```
   #include <cmsbase.h>

   GCC390_NO_ENVIRON();
   ```
3. A global area for the library. This is not for the C library in general, but a feeble attempt at the base CMS level being reentrant. This global area is anchored in the `USERSAVE` area.
4. The runtime stack. It grows downwards.
5. A guard page that has storage key zero. A program check will result as a result of stack underrun unless a stack frame or `alloca()` request is larger than 4K. Dynamic arrays allocated on the stack also add to the frame size.

Caveat emptor: CMS storage is likely to be overwritten if the program manages to allocate stack space below the guard page.

Storage management

All storage obtained by `malloc` and friends is allocated in a private subpool, as is the stack. The exact name of the subpool is unspecified; it may contain binary data to support *CMS/TSO Pipelines* stages written in C.

All storage is released automatically by CMS when the program exits, by nature of it being allocated in a private subpool.

There is no attempt to support other types of storage allocation; you will know what to do if you need it.

Controlling the library

Several switches are available to control the library.

- Logging calls to Callable Service Library routines is controlled as described in "`tocsl`—Interface to CMS callable service" on page 19.

The execution stack

The runtime allocates a storage area to contain

`run390.assemble` allocates the execution stack. The default size is 32 pages, equivalent to 128K.

The page at lowest address is a "guard page"; it has storage key 0; thus it not usable for stack space.

You can change the stack size in two ways, in order of priority:

1. Expand the macro `GCC390_STACK_SIZE` specifying the number of 4K pages desired:

   ```
   #include <cmsbase.h>

   GCC390_STACK_SIZE(256);
   ```

   The macro should be expanded in file scope and at most once in a program.

2. Set the global variable `GCC390_STACK_SIZE` in the group `ENVIRON` to the desired number of pages.

   ```
   address command 'GLOBALV SELECT ENVIRON SET GCC390_STACK_SIZE 256'
   ```

Low level stack management

These subroutines maintain a pointer to the C stack in the CMS `USERSAVE` area. They support the PDP library.

__gstack—Allocate a stack frame to the current SVC level

`__gstack` allocates a stack frame if no frame is associated with the current CMS SVC level; if one exists, it is returned for use.

`__gstack` requires no parameters. It returns updated values in these registers:

0 Work register.

1 The C stack. Transfer to register 15 and call the C language routine, for example:

```
call __gstack
lr 15,1
l 14,=v(function)
basr 14,14
lr 15,2
```

4 The current USERSAVE.
5 The C global area.
11 Used as a work area. You must save its contents if it must be preserved over the call.
15 Return code. Nonzero means that the stack frame was not allocated.

__rstack—Release current SVC level's stack frame

__rstack requires no parameters. It returns updated values in these registers:

0 Work register.
1 Work register.
4 The current USERSAVE.
5 The C global area if the stack frame is still in use; otherwise 0.

__cstack—Clear the current SVC level's stack frame address

This will cause the next call to __gstack to allocate a new frame. Useful with *CMS/TSO Pipelines*.

__cstack requires no parameters. It returns updated values in these registers:

4 The current USERSAVE.
5 The C global area of the stack frame, if any. otherwise 0.

__pstack—Set the current SVC level's stack frame address

Useful with *CMS/TSO Pipelines*.

__pstack requires these parameters:

1 Replacement global area address.

It returns updated values in these registers:

4 The current USERSAVE.
5 The C global area of the previous stack frame, if any. otherwise 0.

Interlanguage calls

Calling C subroutine from assembler main

The compiled C subroutines adhere to the Linux ABI. Specifically the register conventions described in "Calling conventions (Linux ABI)" on page 33.

You must allocate a stack frame (you can reuse the same stack frame for multiple calls) and convert the assembler format parameter list to C parameter conventions, or more likely, write C code to extract the parameters from a structure the address of which you supply as a parameter.

Likewise, the return code, which will be in general register 2, must be transferred to register 15 and the standard save area reestablished.

Calling assembler from C

The assembler routine will be invoked as per the ABI. You can allocate a PL/j save area stack on the C stack:

```
c2asm csect
 stm 6,15,24(15)
 lr 13,15
 ahi 13,-sasize
 ...
 lr 2,15
 ahi 13,sasize
 lm 6,14,25(13)
 br 14
```

You can call back into C code by setting register 15 to 96 less than your save area stack.

Chapter 8. Reference information

Source for the utilities and the required macros is available from

`http://vm.marist.edu/~pipeline/gas2asm.html`

The conversion process is three steps:

1. `gas2asm` generates an `ASSEMBLE` file with possibly very long lines.
2. `asmxpnd` produces the proper F(80) record layout for `HLASM`.
3. The macros in `FPLG2A MACLIB` processes some `gas` instructions.

Editing makefiles

The recipes in a makefile (the indented lines that specify an action) must start with a tab. Be careful if you use an editor that does not preserve tabs, such as `KEDIT` unless you specify `TABSOUT`.

```
h390: ${VMOUTDIR}/hello.text
   echo -e 'site fix 80\nbin\nput' $< |ftp cphart
```

In the example above, the second line is a recipe; `KEDIT` used `TABSOUT ON 3` to create the tab.

On the other hand, variable assignment and everything else must not have leading tabs, which means that you have to curtail your penchant for indenting:

```
ifndef UPLOAD
 UPLOAD:=rexx ../fplupload
endif
```

This works because there is only one blank, so `KEDIT` leaves it as it is.

Linking on CMS

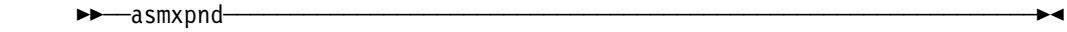
We support two types of C; two `EXEC`s support this.

- The UNIX flavour that uses the standard C library. Use `FPLLINK`.
- Metal C, which only requires a stack frame. Use `FPLLINKM`.
- As of 1.1.12/000A, *CMS/TSO Pipelines* stages in C are built into filter packages. When *CMS/TSO Pipelines* resolves a stage to C code, it builds the appropriate C environment for the program; no bootstrap code is required in the filter package.

`asmxpnd`—Fold long lines with continuation

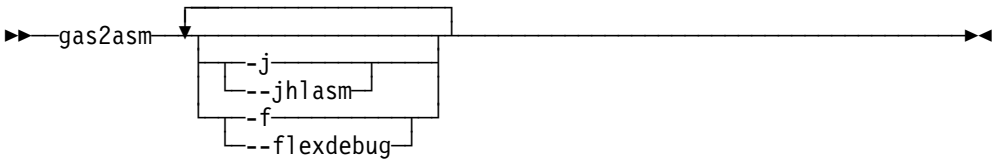
`gas2asm` produces one line output for each line of input. The `asmxpnd` utility ensures that the file format is palatable to HLASM.

`asmxpnd` reads from standard input and writes to standard output. No command flags are recognised.



gas2asm—Convert .s to HLASM

gas2asm reads its input (the GAS file) from standard input and produces its output (the HLASM source) on standard output. gas2asm changes periods (.) in identifiers to number symbols (#).



`-j`
`-jhlasm`
: Generate output for an assembler that supports long ESD names, and case sensitive ordinary symbols. That is, suppress hashing of long external names.

`-f`
`-flexdebug`
: Enable debugging output from the flex scanner. This switch is designed for debugging of gas2asm.

gas2asm restrictions

This section deals with gas2asm when `-j` is not specified.

gas2asm does not detect symbols longer than 64 characters and does not detect two valid symbols that fold to the same uppercase.

External symbols

gas2asm detects external symbols that are longer than eight characters and hashes these down to acceptable symbols, all of which start with a dollar sign. Entry symbols that are so converted are noted on standard error:

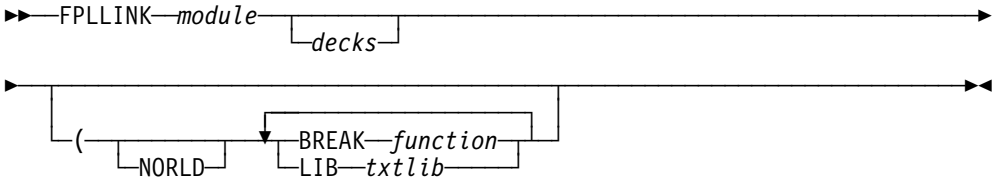
```
$5eoucdb maps deflateInit2_
$2pbg2od maps deflateInit_
```

This conversion is case sensitive. gas2asm also punches a card to define the mapping, for example (from a different program than the one in example above):

```
 punch '*map $3c1qrec __userFiles (extrn) in go390'
```

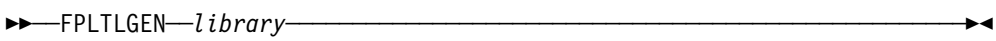
Note that the TXTLIB command quietly deletes all input cards that do not contain X'02' in the first column (but txtlib does not); if you wish to maintain a directory, you must extract the information from the uploaded composite TEXT deck.

FPLLINK—Link a CMS module



FPLLINKM is a wrapper EXEC for programs that use the cmsmain() entry conventions rather than the UNIXy main()

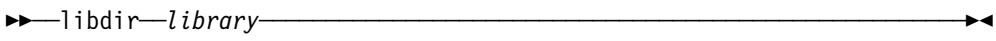
FPLTLGEN—Generate TXTLIB from stacked object decks



library Specify the file name of the input TEXT file. The library will have the same file name as the input file.

Note: FPLTLGEN creates a member for each object deck in the input file, whereas TXTLIB creates a single member for each input file, even when the file contains multiple object modules.

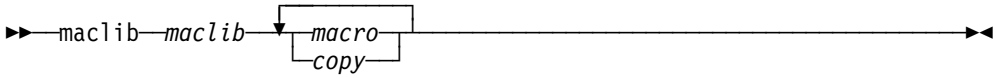
libdir—List directory of a macro or text library



maclib—Generate macro library

maclib generates a CMS format macro library in F(80) EBCDIC for use on CMS.

The input file(s) are ASCII variable length with line end characters.

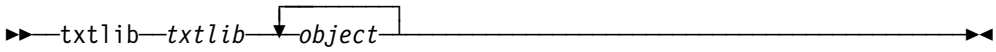


Specify the name of the output file followed by the input file(s) including any file name extensions.

txtlib—Generate text library

txtlib generates a CMS format text library in F(80) EBCDIC for use on CMS.

The input file(s) are F(80) EBCDIC.



Specify the name of the output file followed by the input file(s) including any file name extensions.

Notes:

1. `txtlib` creates a member for each `END` card in an input file, whereas `TXTLIB` creates a single member for an input file. Thus `txtlib` is equivalent to `FPLTLGEN`.
2. Unlike `TXTLIB`, `txtlib` does not filter out records that contains other than `X'02'` in the first position.

Appendix A. Distribution and installation

This appendix describes the development set up and the deliverables you can obtain. The installation procedure is described in Chapter 2, “Cookbook” on page 3; in particular, “Clone cmslib-exec and upload VM objects” on page 4 describes the deliverables.

Directory structure of the source

You can obtain the library source by cloning

`https://github.com/jphartmann/cmslib`

Choose any convenient directory as the root of the directory tree. For the moment we assume that your workstation is a single user Linux and that you keep all files in your home directory. I use `$HOME/src/github/cmslib` as the root for this endeavour, but you can choose anything you like.

Figure 5. Development directory structure and contents

`.`		The root contains a makefile that is embedded from lower level directories.
`cmsbase/`		Assembler code and C code to set up for calling `main()`.
`cmslib/`		Library routines specific for CMS and other library routines I have written.
`glibc/`		The tree search routine, which is lifted from `glibc`. The source says that it was put in the public domain, but `glibc` still adds their copyleft boiler plate. Have your legal department go figure.
`pdplib/`		Source code for the public domain C library. Some are modified by me; others are unchanged. `http://sourceforge.net/projects/pdos/`
`include/`		C language header files for the C library. These files are needed for compile in general. Some are modified by me; others are unchanged.
`samples/`	`minimal/`	The smallest possible executable. It sets the return code to forty-two. Link with `FPLLINKM`.

Making a sample program

To generate a sample program:

- Navigate to the directory that contains the sample you wish. We recommend that you start with `samples/minimal`.
- Issue `make`. This will compile the program and generate the `assemble` file.
- Issue `make up` to upload the assembler source.
- Assemble the program on CMS if you did not do so at the workstation.
- Link the program using `FPLLINK` or `FPLLINKM`, as appropriate.

- Test your new module.

Dealing with multi-file projects.

When your project contains multiple C source files, as is the norm, you must specify all the files in the CSRC make variable. There is a shortcut, however, when your project contains all the C files in the directory:

```
LIB:=mycommand

CSRC:=${basename ${wildcard *.c}}
CFLAGS:=-I.

include ../fplgcc.makefile
```

Assembling on CMS

To assemble on CMS, you need the FPLG2A macro library. If you are assembling a single module, do as you have always done.

To assemble the contents of a library, be sure to specify BATCH as an option to HLASM; if not, you will assemble only the first member of the library. Having obtained a composite TEXT deck, use FPLTLGEN to build the library.

Building a library assembled on your workstation

You can use the txtlib UNIX command to generate the text library on UNIX. Upload binary F(80).

Appendix B. Building a cross compiler with CT-NG

Below is a heavily edited terminal session. Lines starting in a number sign (#) are my comments. We assume that you have already downloaded the source package from http://crosstool-ng.org/ and that it is in some convenient directory.

```
[/usr/data] ls
crosstool-ng-1.20.0.tar.bz2
[/usr/data] tar -xf crosstool-ng-1.20.0.tar.bz2
[/usr/data] cd crosstool-ng-1.20.0
[/usr/data/crosstool-ng-1.20.0] ./configure --prefix=${HOME}
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
   <much more>
checking for library containing initscr... -lncurses
configure: creating ./config.status
config.status: creating Makefile
[/usr/data/crosstool-ng-1.20.0] make
  SED    'ct-ng'
  SED    'scripts/crosstool-NG.sh'
   <much more>
  SED    'docs/ct-ng.1'
  GZIP   'docs/ct-ng.1.gz'
[/usr/data/crosstool-ng-1.20.0] make install
  GEN    'config/configure.in'
  GEN    'paths.mk'
   <much more>
For auto-completion, do not forget to install 'ct-ng.comp' into
your bash completion directory (usually /etc/bash_completion.d)
[/usr/data/crosstool-ng-1.20.0] cd ..
[/usr/data] ct-ng list-samples
Status  Sample name
  LN    config
   <much more>
[G..]   powerpc-unknown_nofpu-linux-gnu
[G.X]   s390-ibm-linux-gnu
[G..]   s390x-ibm-linux-gnu
[G..]   sh4-unknown-linux-gnu
[G..]   sparc-unknown-linux-gnu
   <much more>
[/usr/data] ct-ng s390x-ibm-linux-gnu
  CONF  config/config.in
#
# configuration saved
#

************************************************************

Initially reported by: Harold Grovesteen
URL: http://sourceware.org/ml/crossgcc/2009-11/msg00052.html

************************************************************

Now configured for "s390x-ibm-linux-gnu"
[/usr/data] ct-ng build
[INFO ]  Performing some trivial sanity checks
```

```
. [ERROR]  Don't set LD_LIBRARY_PATH. It screws up the build.
. [00:00] / touch: cannot touch '/usr/data/.build/backtrace': No such file or directory
. [ERROR]
. # This is a bug in CT-NG, but it is easy to fix:
. [/usr/data] mkdir .backtrace
. # I do require the library path in general, but not here
. [/usr/data] unset LD_LIBRARY_PATH
. [/usr/data] ct-ng build
. [INFO ]  Performing some trivial sanity checks
. [INFO ]  Build started 20150115.095838
. [INFO ]  Building environment variables
. [EXTRA]  Preparing working directories
. [EXTRA]  Installing user-supplied crosstool-NG configuration
. [EXTRA]  ===================================================================
. [EXTRA]  Dumping internal crosstool-NG configuration
. [EXTRA]    Building a toolchain for:
. [EXTRA]      build  = x86_64-unknown-linux-gnu
. [EXTRA]      host   = x86_64-unknown-linux-gnu
. [EXTRA]      target = s390x-ibm-linux-gnu
. [EXTRA]  Dumping internal crosstool-NG configuration: done in 0.03s (at 00:01)
. [INFO ]  ===================================================================
. [INFO ]  Retrieving needed toolchain components' tarballs
. [EXTRA]    Retrieving 'linux-3.15.4'
.    <lots and lots of stuff>
. [INFO ]  Build completed at 20150115.102036
. [INFO ]  (elapsed: 21:57.08)
. [INFO ]  Finishing installation (may take a few seconds)...
. [21:58] / [/usr/data]
. # Here are your cross compiled executables:
. [/usr/data] ls ~/x-tools/s390x-ibm-linux-gnu/bin
. s390x-ibm-linux-gnu-addr2line     s390x-ibm-linux-gnu-gcc         s390x-ibm-linux-gnu-nm
. s390x-ibm-linux-gnu-ar            s390x-ibm-linux-gnu-gcc-4.9.1   s390x-ibm-linux-gnu-objcopy
. s390x-ibm-linux-gnu-as            s390x-ibm-linux-gnu-gcc-ar      s390x-ibm-linux-gnu-objdump
. s390x-ibm-linux-gnu-c++           s390x-ibm-linux-gnu-gcc-nm      s390x-ibm-linux-gnu-populate
. s390x-ibm-linux-gnu-cc            s390x-ibm-linux-gnu-gcc-ranlib  s390x-ibm-linux-gnu-ranlib
. s390x-ibm-linux-gnu-c++filt       s390x-ibm-linux-gnu-gcov        s390x-ibm-linux-gnu-readelf
. s390x-ibm-linux-gnu-cpp           s390x-ibm-linux-gnu-gprof       s390x-ibm-linux-gnu-size
. s390x-ibm-linux-gnu-ct-ng.config  s390x-ibm-linux-gnu-ld          s390x-ibm-linux-gnu-strings
. s390x-ibm-linux-gnu-elfedit       s390x-ibm-linux-gnu-ld.bfd      s390x-ibm-linux-gnu-strip
. s390x-ibm-linux-gnu-g++           s390x-ibm-linux-gnu-ldd
. # And here are binutils without the qualifiers
. [/usr/data] ls ~/x-tools/s390x-ibm-linux-gnu/s390x-ibm-linux-gnu/bin
. ar  as  ld  ld.bfd  nm  objcopy  objdump  ranlib  strip
```

Appendix C. Notes

This appendix contains miscellaneous information that you may need if you dive into the guts of things.

Calling conventions (Linux ABI)

The code generated by the compiler use these register conventions. They are not negotiable as this is the z/Linux binary `API`; this is the environment in which we must live. You might wish to inspect `run390.assemble` to see how Assembler code interacts with C.

- Floating point parameters and result are passed in the floating point registers. Refer to the `ABI` for details of which registers are saved by who. Binary floating point is used throughout, unless floating point is suppressed by `-msoft-float`.

For the general registers:

- Register 15 contains the stack pointer. The stack grows downwards. The basic message is do not touch it.
- Register 14 contains the return address for a function call. The return address is not preserved.
- Registers 6-15 are saved/restored as needed across function calls. The callee saves registers and restores them.
- Registers 2-5 are used for the first four parameters, depending on size.
- The function result is in register 2.
- Registers 0-5 are used scratch registers. They are neither saved nor restored.

Gnu assembler operation codes

In general, the source output from the compiler uses the mnemonic operation codes defined in the Principles of Operation manual, but there are differences in the extended branch mnemonics (after all, the compiler is generating code for `gas`, not HLASM).

- `jle` means jump less or equal (mask X'C') rather than jump long equal. There are several more that you cannot pass to HLASM. `gas2asm` copes with this.
- Several other extended mnemonics are not recognised by HLASM. Refer to the comments in `gas2asm.l`.

Downgrading to G3 level set (P/390)

G5 is the earliest level set supported by the compiler.

In theory you could supply macros to expand G5 instructions to older machine operations, but the compiler uses aggressively the new instructions for strings that are zero terminated unless you force it to call an external function instead (if you can; I am not sure).

One thing you cannot do is start calling external functions without care:

- There may not be allocated a stack frame.

- You do not know which registers are available, so you will have to save and restore. You are, however, free to use the area below where R15 points as long as you do not call another function. Just be sure to restore R15 if you change it.

  However, general registers 0 and 1 are available at a function entry point, so they can be used for prologue code (this assumes that `gas2asm` is modified to emit a macro identifying a prologue).

- You do not know whether a base register is available. And even if one is (it is usually register 13), you may have pushed things beyond the end of the available register.
- Your additional code may push a jump target beyond the 64K range of a relative jump.

In summary, a bit of a challenge, but probably not impossible.

Downgrading to System/370

Just more of the same as in the previous section. For example, `la` might be substituted for several operands to `lhi`; other operand classes may require literals. As the machine has no relative branches, any branch will need a register, but you will not know that one is available.

There is, however, some hope as information may be gleaned from the generated code by `gas2asm`; I shall append details when they become available.

Floating point considerations in a 370 virtual machine

A 370 virtual machine in an unmodified CP will not be able to use binary floating point as that requires enabling by a bit in the ESA format of CR0.

Thus, you must compile with `-msoft-float`. This in turn will expose the floating point simulation routines, which you must supply. You will have to convert between binary floating point and hexadecimal and back (which is how the G4 hardware did binary floating point).

Position independent code

The compiler flag `-fpic` works in concert with a dynamic loader to allow multiple concurrent invocations with each invocation maintaining its own writable static. Language Environment supports program objects, which can contain both shared and private sections.

As for `GCC`, specifying `-fpic` with a `s390` target essentially creates a section of writable static that is loaded with the runtime offsets. The assembler code is modified (relative to not specifying `-fpic`) as follows:

1. Sprinkle the code with hints like these:

```
   .cfi_startproc
   stm   %r11,%r15,44(%r15)   #,,
   .cfi_offset 15, -36
   basr  %r13,0   #
.L6:
   ahi   %r15,-152   #,
   .cfi_def_cfa_offset 248
   lr %r11,%r15   #,
   .cfi_def_cfa_register 11
```

CFI stands for call frame information. This is also used by the debugger. The first line simply says function begin; it must be paired with a `.cfi_endproc`. The second line defines the offset from R15 to the information for this frame, in this case 8 bytes into the stack frame. The last two lines defines that cfa (?) as being 246 from register 11, which is the new stack frame (strange).

2. Establish a global offset table:

```
   l  %r12,.L7-.L6(%r13)   #,
   la %r12,0(%r12,%r13) #,
```

3. Expand function calls:

```
   l  %r1,.L12-.L6(%r13)   # tmp87,
   bas   %r14,0(%r1,%r12)  #,
```

4. Replace pointers with a rather hokey construct which presumably indicates an offset:

```
.L12:
   .long fplgccpf@PLTOFF
.L9:
   .long .LC2@GOTOFF
```

5. Define the global offset table:

```
.L7:
   .long _GLOBAL_OFFSET_TABLE_-.L6
   .align   2  #
   .cfi_endproc
```

All of this was from a program that has no static data. From our perspective, we could have used address constants just as well (but the code is demonstrably free of relocation other than `@GOTOFF` and the like).

For our purposes, there are two distinct areas to be addressed:

1. Allocating and initialising the global offset table. No doubt the loaded resolves `_GLOBAL_OFFSET_TABLE_` to a piece of storage it allocates, so that the address becomes relative to the literal base register, but there must be magic and mirrors here, for the relative address is in the `.text` segment.

   The various addresses could become pseudo registers and we can also allocate PRs for writable static and commons.

2. Allocating and initialising writable static and common areas. Unless we write our own dynamic loader for CMS (we have one for OS/j), we must add initialisation code to the program.

3.

This does not seem to be a way forward.

Modifying the generated code to support multiple sections

An alternative mapping from ELF to object could be this:

`.text`	Map to `RSECT` as currently done.
`.data`	Assemble as private code for initialisation, but allocate in an external dummy section for execution.
`.com`	Allocate an external dummy section. (The `DXD` instruction.)
`.bss`	Allocate a private dummy section. This also applies to uninitialised static storage.

Writable storage is allocated along with the stack frame by the bootstrap code. The base of this work area is carried forward in the slot of the stack frame that is reserved for the compiler.

A bit of magic and mirrors will generate code that accesses writable static through the pseudo registers.

When using `HLASM`, there is no concept of a private external dummy section, so the `.bss` segment must be allocated in an external dummy section that has a unique name.

Naturalising the ELF module format

The build chain provided by z/Linux or set up with `CT-NG` can generate an executable, but the program cannot directly on `CMS` for several reasons. However, if one were to replace the bootstrap code that the linker adds, one could build modules that contain code that will be able to execute given a stack frame.

A couple of niggles (aka smops):

- The CMS bootstrap code must be available as an ELF object file. Not insurmountable.
- A CMS command to load such a module, perhaps as a nucleus extension.
- A method to deal with writable static and uninitialised storage areas.
  - Most reentrant (thread safe) libraries do not use such storage. This can be detected and/or enforced by the as yet hypothetical ELF loader on CMS.
  - Unless the code is in an interrupt handler, there will be no preemption and the use of writable static is OK as long as the contents is not required to survive, *e.g.*, a module recursion.

MVS considerations

The bootstrap code is very much CMS, but even some of the C code build CMS parameter lists to perform their function. These latter can be found because they call underlying assembler.

Off the top of my head, you will need to address these CMS issues that will become apparent as you assemble the code:

- Storage allocation is done via CMS macros to define a subpool and allocate stack and heap space.
- The C environment is anchored off the user save area, which is found from the system save area. You might be able to substitute name/token services. However, on CMS this is a four instruction sequence and it is heavily used because the compiler reserves no register for `DSA` or equivalent.
- Console I/O must be mapped to `TGET/TPUT` or equivalent.
- Disk I/O is done by callable services. These will need mapping to `DCB` or what not.

Change activity

3 Aug 2011	Initial writing.
5 Oct 2011	Editorial changes.
9 Oct 2011	Add data areas `cmsstack.h` `cmssubcom.h` `scblock.h`. Add functions `cmssubcr()` `cmssubdl()`.
. 2 Apr 2014	Add name hashing to deal with collisions when folding long external names.
. 10 Jan 2015	Add information on interlanguage calls. Minor editorial changes.
. 13 Jan 2015	Change the name of the `asmxpnd` utility to be in line with *CMS/TSO Pipelines*.
.	Add information about availability of downloadable material.
.	Add documentation of the `gas2asm` and `asmxpnd` utilities.
. 15 Jan 2015	Restructure to use standard UNIX makefile rather that a REXX script to compile and convert.
. 8 Feb 2015	Add notes about alternative or supplemental approaches.
. 20 Jun 2016	Update to reflect the repositories on `github`.
.	Add `maclib` and `txtlib`.
. 25 Jun 2016	Final edit for the `github` distribution.
.	Add `libdir`.

Index

Special Characters

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

Table Definitions

Figures

Headings

Revisions

id	File	Page	References
MOD1	GCC390S.SCRIPT		
		i	
			ii, ii, vi, vi, vi, vi, vi, vi, vii, vii, viii, viii, ix, ix, ix, ix, ix, ix, 1, 5, 6, 6, 6, 6, 7, 7, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 11, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 16, 16, 20, 20, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 23, 23, 24, 24, 27, 29, 30, 30, 32, 34, 37, 38, 38
MOD2	GCC390S.SCRIPT		
		i	
			ii, ii, vi, vi, vi, vii, vii, viii, viii, viii, ix, ix, 2, 2, 3, 4, 5, 5, 5, 5, 5, 5, 6, 6, 6, 7, 7, 7, 9, 9, 9, 9, 9, 9, 12, 12, 12, 12, 12, 12, 13, 13, 14, 14, 18, 18, 18, 18, 19, 19, 25, 25, 26, 26, 27, 27, 27, 27, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30, 30, 30, 30

Processing Options

Runtime values:

Document fileid	GCC390 ENLIGD
Document type	USERDOC
Document style	DEFAULT
Profile	EDFPRF40
Service Level	0033
SCRIPT/VS Release	4.0.0
Date	16.06.25
Time	14:21:43
Device	PSA
Number of Passes	2
Index	YES
SYSVAR B	SPINE
SYSVAR G	INLINE
SYSVAR R	GCC390
SYSVAR S	OFFSET
SYSVAR W	GCC390
SYSVAR X	YES
SYSVAR Z	YES

Formatting values used:

Annotation	NO
Cross reference listing	YES
Cross reference head prefix only	NO
Dialog	LABEL
Duplex	SB
DVCF conditions file	(none)
DVCF value 1	(none)
DVCF value 2	(none)
DVCF value 3	(none)

DVCF value 4 ..... (none)
DVCF value 5 ..... (none)
DVCF value 6 ..... (none)
DVCF value 7 ..... (none)
DVCF value 8 ..... (none)
DVCF value 9 ..... (none)
Explode ..... NO
Figure list on new page ..... NO
Figure/table number separation ..... NO
Folio-by-chapter ..... NO
Head 0 body text ..... (none)
Head 1 body text ..... Chapter
Head 1 appendix text ..... Appendix
Hyphenation ..... YES
Justification ..... NO
Language ..... ENGL
Keyboard 395
Layout ..... OFF
Leader dots ..... NO
Master index ..... (none)
Partial TOC (maximum level) 4
Partial TOC (new page after) ..... INLINE
Print example id's ..... NO
Print cross reference page numbers ..... YES
Process value ..... (none)
Punctuation move characters ..... (none)
Read cross-reference file GCC390
Running heading/footing rule ..... NONE
Show index entries ..... NO
Table of Contents (maximum level) 3
Table list on new page ..... YES
Title page (draft) alignment ..... RIGHT
Write cross-reference file GCC390