CONVERGENT TECHNOLOGIES

PROGRAMMER'S NOTES FOR C CROSS-COMPILATION & FLEXNAMES

Revised 05/21/86

Trademark Notice

TABLE OF CONTENTS

1. Introduction

The "Software Generation System" is a new set of compilers, libraries and utilities that allow you to develop applications on a CTIX-based *host* machine that will run on all CTIX-based *target* machines, ie. the MiniFrame, MegaFrame and MightyFrame.

Generating code for any CTIX hardware environment different from that of the *host* machine is called "cross-compiling" or, in the case of assembler code, "cross-assembling."

The first version of the Software Generation System is included in the MightyFrame release, with the MiniFrame and MegaFrame versions to follow in 1986. The purpose of such a system is to simplify the development of an application or user program for all CTIX architectures through the use of a single source program, thus making source code easy to control and maintain.

This document is divided into two sections: the first section discusses cross-compilation issues and the second section discusses flexnames.

2. Cross Compilation Environment

There are three types of architectures to consider when compiling your application:

- a 68010 CPU (MiniFrame or MegaFrame) system with no hardware floating point processor

- a 68020 CPU (MightyFrame or MegaFrame with AP2) system with no hardware floating point processor

- a 68020 CPU (MightyFrame or MegaFrame with AP2) system with a 68881 floating point processor.

In order to support code generated for different architectures, Convergent has provided compilers and libraries that have been built for each of the hardware configurations. This section describes how an end user or a third party vendor can:

- cross-compile to generate code for a specific architecture different from the *host*

- cross-compile to generate code that runs on all CTIX machines

- provide libraries to support each configuration.

2.1 What Code Runs on Which Machines

1. Code compiled to run on the 68010 processor with software floating point runs on all new processors. Thus, all existing applications programs that run on MiniFrame and MegaFrame will run on MightyFrame, except as follows:

    - C applications that access certain operating system tables defined in **/usr/include/sys**. Typical examples are system programs like debuggers, ISAM file servers, and data communications products.

    - Basic, COBOL, FORTRAN, Pascal, or assembly code programs linked with C routines that access certain operating system tables defined in **/usr/include/sys**.

    These exceptional cases require that you compile a separate version for each target system.

Although the vast majority of 68010 applications run on the 68020 based systems, they run less efficiently on the 68020 than they would if they were compiled for the 68020 processor. This is because the 68020 based systems have a more efficient architecture, 32 bit data paths, new instruction sets, and better compiler optimizations. Performance degradation due to alignment problems is avoidable. Refer to the subsection, "How to Get Optimal Code for 68020 Execution."

2. Code that is compiled for the 68020 can only run on the 68020.

3. Code that is compiled for the 68881 math coprocessor will only run on machines with the 68881s.

2.2 How to Set Up Your Compilation Environment for a Specific Target Hardware Configuration

Setting up your compilation environment so that you can cross-compile requires these steps:

1. Include the group *CROSS* at installation time.

   When you specify the *CROSS* group, libraries for cross compilation are installed as subdirectories of the directory **/cross**. Each subdirectory contains a **/lib** and a **/usr/lib**.

   - The directory **/cross/1sw** contains the libraries necessary to cross-compile for 68010-based products.

   - The directory **/cross/2sw** contains the libraries necessary to cross-compile for 68020 with software floating point.

- The directory **/cross/2fp** contains the libraries necessary to cross-compile for 68020 with 68881.

2. Create a subdirectory under **/cross** for the *target* machine (for example, **/cross/mega**). Then copy the include files from the *target* machine and install them on the *host* in a new **/usr/include** directory under the subdirectory you have created (for example, under **/cross/mega/usr/include**).

3. Set and export three environment variables: CENVIRON, LIBROOT, and INCROOT. You can do this in your **.profile** for Bourne shell, or **.login** for C shell, or directly from the shell.

   - CENVIRON specifies the target hardware. (The procedure that installs MightyFrame CTIX from tape sets CENVIRON to CPU=68020 in the file **/etc/profile**.)

     The value of CENVIRON is enclosed in quotes and is specified as two keyword strings, separated by a comma: the first keyword string is CPU (68020 or 68010); the second is floating point (SOFTWARE or 68881).

     For example:

     ```
     CENVIRON="CPU=68020,FPU=68881"
     export CENVIRON
     ```

     Note that the default values are 68010 and SOFTWARE. (That is, if you remove the CENVIRON setting from **/etc/profile** and CENVIRON is not otherwise set, the default values are 68010 and

SOFTWARE.)

- LIBROOT specifies the location of libraries for the loader (**ld**) to search during the final link.

  The value of LIBROOT is enclosed in quotes and is specified as a subdirectory containing the appropriate libraries. It is recommended that you keep all cross-compilation libraries under the **/cross** directory: in this case, there are three possible values for LIBROOT ("**/cross/1sw**", "**/cross/2sw**", and "**/cross/2fp**"). The value of this variable is prepended to the directories the loader normally searches.

  For example:

  ```
  LIBROOT="/cross/2fp"
  export LIBROOT
  cc myprog.c - lm
  ```

  Note with respect to this example that if LIBROOT were not set, the loader would search the library in **/lib/libm.a** in order to complete the linking. With the LIBROOT variable set as above, the loader searches the file **/cross/2fp/lib/libm.a** instead.

  Also note that the LIBROOT setting does not affect the search path of libraries whose *full path name* is specified on the command line.

- The INCROOT environment variable specifies the set of include files to be used for cross development.

  The value of INCROOT is enclosed in quotes and is specified as a subdirectory containing the appropriate include files (for example, **/cross/mega**). As explained earlier, the include files directly under **/usr/include** are equivalent for all CTIX based machines, but a small number of the data structures defined in **/usr/include/sys** differ among the three machines.

EXAMPLES OF POSSIBLE ENVIRONMENT VARIABLE COMBINATIONS

The following combination of environment variables establishes the environment to cross compile for a MiniFrame (the compilation requires system header files and thus the appropriate include files):

```
CENVIRON="CPU=68010,FPU=SOFTWARE"
LIBROOT="/cross/1sw"
INCROOT="/cross/mini"
```

The following combination of environment variables establishes the environment to cross compile for a MightyFrame with software floating point:

```
CENVIRON="CPU=68020,FPU=SOFTWARE"
LIBROOT="/cross/2sw"
```

The following combination of environment variables establishes the environment to cross compile for a MightyFrame with hardware floating point:

```
CENVIRON="CPU=68020,FPU=68881"
LIBROOT="/cross/2fp"
```

2.3 Compiler Options Affecting Cross-Compilation

There are three compiler options (**cc** or **ld** flags) that affect cross-compilation: **– z**, **– T**, and **– G**.

The – **T** (cc) and – **G** (ld) flags allow the loader (**ld**) to resolve references between code whose identifiers have been truncated to eight characters, and code whose identifiers are not or would not otherwise be truncated. (Pre-5.00 CTIX compilers automatically truncate identifiers to eight characters.) These two options are described in more detail in Section 3 of this document.

The – **z** (**cc** or **ld**) option directs the loader on a 5.00 CTIX machine to produce an executable file whose format is different from the default 5.00 format; this option is *required* to cross-compile for a pre-5.00 CTIX machine.

By default, the format of **a.out** (executable) files is different on 5.00 CTIX from pre-5.00 CTIX: on a 5.00 CTIX machine, the loader (**ld**) produces **a.out** files in *AT&T paged file format*, which will execute only on 5.00 CTIX machines; the loader on a pre-5.00 CTIX machine produces **a.out** files in *Berkeley paged file format*, which will execute on both pre-5.00 CTIX and 5.00 CTIX machines.

The – **z** flag is thus provided to enable you to produce code on a 5.00 CTIX machine in *Berkeley paged file format*, which will run on all machines. Note that code produced with the – **z** flag takes longer to page into memory on a 5.00 CTIX machine than code produced without it.

Use the **file** command to identify the file format of an executable file: output of the **file** command displayed with – **z** identifies the file as being in *Berkeley paged file format*; output displayed with – **F** identifies the file as being in *AT&T paged file format*.

2.4 How to Get Optimal Code for 68020 Execution

The 32-bit bus to memory on the 68020 permits longword data aligned on longword addresses to be accessed in half the number of memory cycles as that required on the 68010 processor. The 68020 C compiler is aware of this feature, and lays out a program's data in such a way that 32-bit values reside at 32-bit memory boundaries. This is true for all data types except data residing inside C structures.

For backward compatibility reasons, the elements inside a *struct* remain aligned using the 68010 scheme, which places 32-bit values on 16-bit boundaries in some cases. You can achieve an incremental efficiency improvement by hand-aligning the 32-bit *struct* elements in your code to 32-bit boundaries.

For example, the following C structure is not aligned correctly for optimal 68020 execution:

```
struct example  {
        int     field1;
        short   field2;
        int     field3;
};
```

For correct 32-bit alignment in this example, either pad the structure (add a *short* to insert two slack bytes between field2 and field3), or reverse the order of field2 and field3.

Note that some Convergent libraries and kernel routines have been compiled using the *old* alignment scheme: these *structs* should not be changed or they will no longer correspond to the definitions in force when the code was compiled.

2.5 How to Add Your Own Libraries to be Used When Cross-Compiling

This section describes how to add libraries to support each of three possible machine configurations.

You will probably want to provide a different library for each architecture if your goal is to achieve optimal performance or if the library performs floating point operations. If, however, a library does not perform floating point, it can be built for the 68010 architecture, and code linked with it will run on any of the three machines.

Any library you create to be used for cross-compilation should be installed in the appropriate subdirectory under **/cross**. This allows users who have set the CENVIRON, LIBROOT, and INCROOT environment variables correctly to access your libraries (along with Convergent's) by specifying the **– l** option to **cc** or **(ld)**.

2.6 Linking Object Files Compiled for Different Machines

This section describes

- object file "markings"
- what circumstances allow an object file or library compiled for one architecture to be linked with an object file or library compiled for another architecture
- **a.out** "markings" resulting from linked objects compiled for different architectures.

In general, it is best to avoid linking object files compiled for different architectures. In some cases, however, it is useful to do this: for example, to prevent having to support many versions of libraries.

2.6.1 Object File Markings

When a source file is compiled, the resulting object file is "marked" (that is, bits are set in the file header) with the type of CPU and FPU instructions used.

- The CPU marking is 68010 or 68020. If a compilation for 68020 uses no 68020-specific instructions, the object file is marked 68010.

- The FPU marking is SOFTWARE, 68881, or no floating point. (An object file is defined as being compiled for no floating point if the keywords *float* and *double* were not used and no floating point constants appear in the program.) Table 1 shows what FPU markings result from the possible compilation environments.

Table 1 : FPU Object File Markings

	no FPU specified	FPU=SOFTWARE	FPU=68881
code with *no* floating point variables or constants	none	none	none
code *with* floating point variables or constants	SOFTWARE	SOFTWARE	68881

To determine how an object file is marked, use the **file** command.

Note that when the **file** command is applied to an archived library, it prints only the *archive format* of the library: 5.0 archive corresponds to pre-5.00 CTIX archive format, and 5.2 corresponds to 5.00 CTIX archive format.

To determine how an *archived* object file is marked, follow this procedure:

1. **cd** to **/tmp**.

2. Obtain a list of **.o** files in the library by using the **ar** command with the **– t** option.

   For example,

   ```
   ar – t /lib/libc.a |more
   ```

3. Extract the **.o** file from the archive using the **ar** command with the **– x** option. (This does not remove the file from the library; it only copies the **.o** file to the current directory.)

   For example,

   ```
   ar – x /lib/libc.a  acct.o
   ```

4. Use the **file** command on the **.o** in **/tmp** (your current directory).

   For example,

   ```
   file acct.o
   ```

2.6.2 a.outs Resulting From Linked Objects

Most combinations of object files with different markings are legal. Table 2 shows all possible combinations and the file markings of their resulting **a.outs**. Here are some general rules about linking differently-marked objects:

- Objects with FPU marked SOFTWARE *can not* be linked with objects marked 68881.

- In legal combinations, when objects have different file markings, the **a.out** is always "upgraded" to the most advanced object marking: for example, if a 68010 object without floating point is linked with a 68020 object with software floating point, the **a.out** is marked with 68020 and software floating point. (Note that this principle applies equally to normal environments where no cross-compilation is involved: for example, when a source file that has no floating point variables or constants references a C run-time function, like *printf*, that uses floating point, the executable file is marked with floating point.)

When the upgraded marking of an **a.out** will conflict with the value of CENVIRON on the machine doing the linking, the linker issues a warning and proceeds with the final link.

Table 2 : a.outs Resulting From Linked Objects

	10	10+ sw	20	20+ sw	20+ fp
10	10	10+ sw	20	20+ sw	20+ fp
10+ sw	10+ sw	10+ sw	20+ sw	20+ sw	illegal
20	20	20+ sw	20	20+ sw	20+ fp
20+ sw	20+ sw	20+ sw	20+ sw	20+ sw	illegal
20+ fp	20+ fp	illegal	20+ fp	illegal	20+ fp

3. Flexnames

3.1 Implications of the Flexname Feature

"Flexname" is the AT&T term for the feature that allows a C program identifier (e.g., a variable or function name) to be longer than eight characters. Support of long identifier names is a significant part of the AT&T System V.2 standard.

Pre-5.00 CTIX compilers automatically truncate identifiers to eight characters. For example, when a C program calls the ISAM procedure *StoreISAMRecord*, the pre-5.00 compiler truncates the identifier to *StoreISA*. (The *StoreISAMRecord* procedure is similarly known to the pre-5.00 CTIX archiver and ISAM library as *StoreISA*.)

Porting code written for a machine *without* flexname support to a 5.00 CTIX machine is normally very easy. Porting code written for a machine *with* flexname support to a pre-5.00 CTIX machine can introduce bugs that can be difficult to find unless you are sensitive to flexname-related issues and consequences.

These are the Convergent Technologies program development products that are impacted by flexnames: the C compiler (**cc**) and loader (**ld**), **make**, **ar**, **strip**, **size**, **nm**, **as**, **dump**, **lint**, **cflow**, **sdb**, **adb**, **cxref**, **prof** and **convert**.

Because of flexname differences in these tools and other flexname-related issues, *we recommend that you recompile programs written for the pre-5.00 CTIX compiler and assembler when you receive 5.00 CTIX software.*

Note that Convergent Technologies will release new versions of its libraries that do not presently use flexnames. These are ISAM, Sort/Merge, Forms, several compiler libraries, and data communications libraries. Until new versions of these libraries become available, use backward compatibility compiler options when you link them on a 5.00 CTIX machine.

3.2 Backward Compatibility Options on the Compiler and Assembler

The C compiler and assembler provide two options to allow you to link code from the pre-CTIX 5.00 compiler and assembler, whose identifiers are truncated, with code whose identifiers are not or would not otherwise be truncated:

- the – **T** flag tells **cc** and **as** to truncate *all* identifiers to eight characters and to pass the – **G** option to the loader. The loader (**ld**) then resolves references on the basis of the first eight characters only.

  This option causes the 5.00 CTIX compiler and assembler to behave exactly like the pre-5.00 compiler and assembler with respect to identifier length.

- the – **G** flag does *not* cause identifiers to be truncated during compilation, but tells (**ld**) to use the following algorithm when it attempts to resolve external references: if two external names do not initially match and one of them is exactly eight characters, allow the match if their first eight characters match and issue a warning. (The warning message can be suppressed if you specify the – **w** flag.)

3.3 Converting Your Programs and Libraries to a Flexname Environment

Note that Convergent Technologies currently supports two formats for assembly code. At 3.0 we changed the assembler to the new format and called it *mas* and *mcc* at 3.0 as and cc where the old format. At 5.0 *as* and *cc* use the new format.

The pre-CTIX 5.0 *as* format is derived from Convergent's original, non-optimizing C compiler. This assembler does not support flexnames.

The new assembler *mas* supports assembly language format of Convergent's new, optimizing C compiler *mcc*, introduced at the 3.00 CTIX release. This assembler supports flexnames. Note that CTIX 5.00 *as* format is equivalent to *mas* and CTIX 5.00 *cc* is equivalent to *mcc*.

Convergent will be providing a tool to ease the assembly code conversion process.

These are the steps required to convert your own programs and libraries to a flexname environment:

1. If there is assembly code, and it uses identifiers longer than eight characters, convert it by assembling it using the new assembler.

2. Recompile all the code in your libraries and rebuild your libraries with the **ar** utility. Use the 5.00 CTIX version of **ar**, since **ar** uses a new archive format with this release. Although the loader supports pre-5.00 archive format, we recommend that you convert all of your libraries.

3. Recompile your C programs, assemble your assembly programs, and link them with the necessary libraries.