G21K C Compiler 2

2.1 INTRODUCTION

G21K is Analog Devices' port of GCC, the Free Software Foundation's C compiler, for the ADSP-21000 family of digital signal processors. Separate versions of the compiler are available for MS-DOS and for UNIX.

G21K is a *driver* program: it controls the operation of other programs (software development tools) during compilation. The process of compiling a source file involves several tools and files.

Other files involved with compilation are the runtime header and the architecture file. The runtime header controls initialization of the C runtime environment and interrupt handling. Aspects of the runtime environment (such as code and data placement, stack and heap placement, and target wait states and banks) are controlled by the architecture file.

2.2 INVOKING G21K

G21K is invoked from the command line, and can be run under a DOS box in Windows. Note that though this documentation refers to the compiler as "G21K", you can type `g21k` on the command line. For a complete listing of all available switches, see the "G21K Compiler Switches" chapter of this manual.

Below is the G21K command line syntax:

`g21k [-switches] filename [.ext] [filename [.ext]]`

Some commonly used switch options are shown on the next page.

2 G21K C Compiler

The compiler can also accept multiple input files in the following way:

```
g21k [-switches] @file_all
```

The `file_all` file lists the files to be compiled; it must be a simple text file with one path/filename per line. This feature provides a workaround for the DOS command line length restriction.

Command line switches may also be placed inside `file_all`, but they will apply to all input files listed (i.e. switches cannot be selectively applied to individual files).

Switch	*Effect*
`-E`	Preprocess source files only
`-S`	Generate assembly source files only
`-c`	Generate object files only
`-O`	Optimize code using some optimizations
`-O2`	Optimize code using more optimizations
`-O3`	Optimize code using all optimizations
`-v`	Generate verbose output
`-ansi`	Disable all non-ANSI language extensions
`-g`	Produce debuggable code for use with CBUG
`-h`	Display list of switches
`-a` *`filename`*	Specify alternate architecture file
`-Ipath`	Specify additional paths to search for include files
`-Dmacro[=value]`	Define a macro for the C preprocessor
`-Lpath`	Specify an additional path to search for library files
`-lxxx`	Include library `libxxx.a` in link line
`-map`	Generate map file (default is `21k.map`)
`-o` *`filename`*	Place output in *`filename`*
`-nomem`	Do not execute runtime memory initializer
`-runhdr` *`filename`*	Specify alternate runtime header file
`-w`	Inhibit all warning messages
`-Wall`	Combine all warnings in this list
`-Wimplicit`	Warn when a function is implicitly declared
`-Wreturn-type`	Warn when a function defaults to returning an `int`
`-Wunused`	Warn when an automatic variable is unused
`-Wswitch`	Warn when a switch does not use all enumeration types
`-Wcomment`	Warn when a comment contains a `/*` sequence
`-Wfloat-convert`	Warn when a `float` number is implicitly converted to a `double`, or a `double` is implicitly converted to a `float`

2.3. ENVIRONMENT VARIABLES

This section describes several environment variables that affect how G21K operates. They work by specifying directories or prefixes to use when searching for various kinds of files.

Note: You can also specify search paths using options such as `-I` and `-L`. These take precedence over paths specified using environment variables, which in turn take precedence over those specified by the configuration of G21K. See the *Options For Directory Search* discussion in the "G21K Compiler Switches" chapter of this manual for more information.

`TMP`

`TMP` specifies the directory to use for temporary files. G21K uses temporary files to hold the output of one stage of compilation which is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

`ADI_DSP`

The value of `ADI_DSP` specifies where to find the ADI tools directory. From this variable the compiler computes the location of the libraries and include files by augmenting the path with:

include files	`$ADI_DSP/21k/include`
libraries	`$ADI_DSP/21k/lib`

2.4 COMPILER STAGES & FILES

G21K compiles a file in stages. Each stage involves a different tool, has its own set of input files, and produces an output file. Command line switches allow compilation to begin or end at any stage. G21K is able to handle multiple files with one invocation.

If a compilation involves more than one stage, the output of each stage is saved in an intermediate file and used as input to the next stage. The location of intermediate files is specified by the `TMP` environment variable. See the *Environment Variables Affecting G21K* discussion in the "G21K Compiler Switches" chapter in this manual for more information. G21K removes intermediate files when finished.

G21K (and the programs it invokes) prints all error messages to `stdout` when running under MS-DOS, and `stderr` when running under Unix.

2.4.1 Stages

The process of compiling a file may involve any of the following stages, always in this order:

1. **C Preprocessing** runs the C preprocessor on the source file. Comments are removed, macros are expanded, conditional compilation and file inclusion are handled. See Chapter 8, “C Preprocessor,” for further information about the C preprocessor.

2. **Compiling** translates C language into assembly language. Command line switches control compile-time options, such as the inclusion of debugging information and degree of optimization. See Chapter 7, “G21K Switches” for further information about the invocation options.

3. **Assembly Preprocessing** runs the C preprocessor on the assembly source file. See item 1, above in this list.

4. **Assembling** translates assembly language into object code.

5. **Linking** combines individual source object files, the runtime header, and object files from libraries into a single executable. Addresses for objects are assigned and references are resolved to form a single executable file.

2.4.2 File Types

Each stage of compilation requires a specific type of input file (or source file) and produces a specific type of output file.

The stage of compilation that G21K begins with is determined by the filename extension of each source file.

Command line switches determine the stage at which compilation ends. See Section 2.1, above, and Chapter 7, “G21K Switches” for information about the invocation options.

Stage	*Tool Invoked*	*Source Extension*	*Output Extension*[1]
C Preprocess	cpp	.c	.i
Compile	cc1	.i	.s
Assembly Preprocess	cpp	.s, .asm	.is
Assemble	asm21k0	.is	.o
Link[2]	ld21k	.o, .obj, .a, *all other extensions*[3]	.lnk, .exe

[1]Note: All intermediate files (`*.i, *.s, *.is, *.o,` and `*.lnk`) are removed unless the `-save-temps` switch is used. See Chapter 7 for details.

[2]Note: The linker normally creates a `*.lnk` file that becomes the input to the initializer. If the `-nomem` switch is used to disable the initializer, the linker outputs a `.exe` file instead.

[3]Note: Any file with an unrecognized extension is given to the link stage.

For example, to compile the assembly language file `port.asm`, the C language file `wire.c` and the library `libconv.a` into the executable `wire.exe`:

```
g21k port.asm wire.c -o wire.exe -lconv
```

G21K performs these actions:

1. a) G21K calls `cpp` on the file `port.asm` to produce `port.is`.
   b) It then calls the assembler on `port.is` to produce `port.o`.

2. a) G21K calls `cpp` on the file `wire.c` producing `wire.i`.
   b) G21K then calls `cc1` to compile `wire.i` into `wire.s`.
   c) G21K calls `cpp` again to preprocess `wire.s` into `wire.is`.
   d) G21K then calls `asm21k0` on `wire.is` to assemble it into `wire.o`.

3. G21K calls `ld21k` to link the two intermediate object files (`wire.o` and `port.o`) together with `libconv.a` to create the file `wire.lnk`. The `-lname` switch causes the linker to look in the directory `$ADI_DSP/21k/lib` for `libconv.a`.

   The default runtime header (`020_hdr.obj` or `060_hdr.obj`) and architecture file (`21k.ach`) are also used by the linker and are searched for in the `$ADI_DSP/21k/lib` directory.

4. G21K calls `mem21k` on `wire.lnk` to produce `wire.exe`.

2 G21K C Compiler

2.4.3 File Formats

Each source or output file must be in one of three file formats:

- ASCII files contain only printable characters. C and assembly language files are in ASCII. They may be viewed and changed manually with any text editor.

- COFF is the Common Object File Format. COFF allows information about address resolution, debugging symbols, and runtime model to be stored and retrieved efficiently. COFF is a binary format and should not be edited manually. ADSP-21000 family object files and executable files are in COFF.

  Note: COFF is documented in *Using and Understanding COFF* (ISBN 0-937175-31-5) from O'Reilly and Associates, 103A Morris Street, Sebastopol, CA 95472.

- Archive format is used for libraries. The linker can efficiently search an archive file to determine if it must extract a copy of an object file. Archive is a binary format and should not be edited manually.

File Contents	*Format*	*Origin*	*Filename Extension*
C Code	ASCII	User, C Preprocessor	.c .i
Assembly Code	ASCII	User, C Compiler, or Assembler Preprocessor	.asm .s .is
Object Code	COFF	Assembler	.o or .obj
Executable	COFF	Linker	.lnk or .exe
Library	Archive	Librarian	.a

2.4.4 COFF Tools

Two utilities are included in the release package to deal with the different executable file formats created with UNIX tools and those created with PC tools.

2.4.4.1 CSWAP

This utility converts executable files from the UNIX COFF format to the DOS format, or vice versa. Use the following command (on a PC) to convert from UNIX COFF format to DOS format:

```
cswap.exe sun.exe [-o] dos.exe
```

Use the following command (on a UNIX workstation) to convert from DOS format to UNIX COFF format:

```
cswap dos.exe [-o] sun.exe
```

2.4.4.2 CDUMP

The CDUMP utility accepts an executable file as input, and outputs an ASCII representation of the COFF file. This utility displays the contents of a COFF object or executable file. Use the following syntax to display COFF files:

```
cdump file.exe > output.fil
```

2.5 FILES INVOLVED IN COMPILATION

There are a number of files that are used in the compilation that you do not need to specify when you invoke G21K the compiler searches for them automatically. You can give command line switches to G21K to specify where it should search for these files.

2.5.1 Runtime Header

The runtime header provides interrupt handling, including chip reset. When the target processor is reset, the runtime header makes calls to routines in the C runtime library to initialize the C runtime environment and set the state of the hardware. The assembly language source code for the runtime header is distributed with the release software so that you may modify it if necessary.

Three different versions of the interrupt dispatcher are available. Each version provides different levels of exception handling and, consequently, different execution speeds. Each version also uses different function calls to facilitate interrupt setup and handling. These include:

Interrupt Dispatcher	*Exception Handling*	*Relative Speed*	*Function Calls*
Regular	Extensive	150 instructions	`interrupt()`, `signal()`
Fast	Moderate	75 instructions	`interruptf()`, `signalf()`
Super	Minimum	30 instructions	`interrupts()`, `signals()`

Note: C runtime library functions are provided to facilitate interrupt setup and handling For more information including limitations, see the Library Reference entries for the `interrupt()`, `signal()`, and `raise()` functions in the *ADSP-21000 Family C Runtime Library Manual*. Also see the interrupts discussion in Appendix C of this manual.

2.5.2 Architecture File

The architecture file describes the software and hardware configuration of your system. Architecture file directives specify the placement and naming of elements of the runtime memory environment such as the locations of the C runtime stack, runtime header, heap, and code and data spaces.

G21K also reads the architecture file to determine *segment names* for code and data. The linker uses architecture file information to determine placement of objects in memory. The linker also reads information about the hardware from the architecture file (such as wait states and external memory banks) and stores it in the executable file. Read Chapter 3, “Writing the Architecture Description File” of the *ADSP-21000 Family Assembler Tools Manual* for details about the architecture file. See also Chapter 3 of this manual for specific architecture file directives to use with a C program.

2.5.3 Library Files

Library files contain sets of object files. The linker can access library files and extract object files from them as needed. If a library contains an object file which has a function the linker requires, a copy of that object file is extracted from the library and incorporated into the executable. Only object files which contain functions that remain unresolved when the library is processed will be extracted from the library.

G21K specifies the `-lc` switch to the linker so that the C Runtime Library, `libc.a`, will be searched for functions.

- You can specify additional libraries for the linker to include by placing them on the G21K command line or with the `-l` command-line switch.

- The linker by default searches for libraries specified by the `-l` switch in `$ADI_DSP/21k/lib/` with the prefix lib and the extension `.a` added to the library name.

- Use the `-L` switch to specify other directories for the linker to search for libraries specified by the `-l` switch.

2.5.3.1 Custom Library Files

For information about how to build custom libraries, see Chapter 7 of the *ADSP-21000 Family Assembler Tools Manual* for a discussion of the librarian tool.

2.6 OPTIMIZATION

Four options for optimization are available for G21K:

- no optimization
- some optimizations, specified by the `-O` switch
- more optimizations, specified by the `-O2` switch
- all optimizations, specified by the `-O3` switch

Higher levels of optimization produce code that runs more quickly on the target processor, but requires more time to compile.

The `-g` switch, which generates information for the debugger, disables all optimizations, regardless of `-O`, `-O2`, or `-O3` switches.

A for loop with a null statement is eliminated by the compiler during optimization unless the variable is global, static, or volatile. If you use these loops for delays, the following example illustrates a possible code change from:

```
int x;
for (x=0; x<10; x++);
```

to:

```
volatile int x;
for (x=0; x<10; x++);
```

2.7 TARGET SYSTEM SELECTION

G21K reads the architecture file to determine which ADSP-21000 Family processor to use (that is, ADSP-21020 or ADSP-2106x SHARC). It uses this information to pass the necessary switches to all underlying stages of the compilation (as discussed in Section 2.4). See section 3.2.1.5, "PROCESSOR= Directive" for further information.

2.8 EXAMPLE

A typical development scenario is to write or modify your code, compile it, and examine its behavior with CBUG. The program used in this example, `primes.c`, computes the first twenty prime numbers. The source code for this example is in the directory

```
$ADI_DSP/21K/examples/primes
```

This example uses G21K to compile `primes.c into primes.exe` and uses CBUG to examine the execution of `primes.exe`.

As you work through this example, refer to the following documentation for detail:

CBUG debugger	Chapter 9, *CBUG C Source-Level Debugger*
Installation information for the ADSP-21000 Development Software	Release Note

2.8.1 The primes.c Program

The `primes.c` program stores computed prime numbers in primes[ ], an array of integers.

```
/* primes.c - list the first twenty prime numbers */
/* Analog Devices, 1993 */
main()
{
    int index = 0;
    int n_primes = 1;
    int primes[20] = 2; /* Initialize to first prime number */
    int testnum = 3;
    while (n_primes < 20)
    {
       /* find a number that is indivisible by known primes. */
        while (index < n_primes)
            if (!(testnum % primes[index++]))
            {
/* if a number is divisible by a known prime, it cannot be
   prime. Start over with the next odd number */
                testnum += 2;
                index = 0;
            }

        /* this number is prime, add it to the list. */
        primes[n_primes++] = testnum;
    }
        /* start checking at the next odd number */
        testnum += 2;
        index = 0;
    exit(0);
}
```

2.8.2 primes.ach

The architecture file `primes.ach` defines a target architecture suitable for the ADSP-21020 EZ-LAB Board. The linker places the runtime header in `seg_rth`. Compiled code is placed in `seg_pmco`, data in `seg_dmda`. The C runtime stack is placed in `seg_stak`.

Note: The stack begins at 0x1FF and grows downward in memory (towards numerically smaller addresses). MEM21K uses `seg_init` to hold initialization information and target state information.

```
.system primes_demo;
.processor ADSP21020
.segment /pm /ram /begin=0x0000 /end=0x00FF   seg_rth;
.segment /pm /ram /begin=0x0100 /end=0x04FF   seg_pmco;
.segment /pm /ram /begin=0x0500 /end=0x05FF   seg_init;
.segment /dm /ram /begin=0x0000 /end=0x017F   seg_dmda;
.segment /dm /ram /begin=0x0180 /end=0x01FF   seg_stak;
.bank/pm0/wtstates=0/wtmode=internal/begin=0x000000;
.bank/pm1/wtstates=0/wtmode=internal/begin=0x008000;
.bank/dm0/wtstates=0/wtmode=internal/begin=0x00000000;
.bank/dm1/wtstates=0/wtmode=internal/begin=0x20000000;
.bank/dm2/wtstates=1/wtmode=internal/begin=0x40000000;
.bank/dm3/wtstates=0/wtmode=internal/begin=0x80000000;
.endsys;
```

2.8.3 Compiling primes.c

If you have not yet installed G21K, do so now. Instructions for installation are in the Release Note.

Compile primes.c into primes.exe by typing:

```
g21k primes.c -a primes.ach -o primes.exe -g
```

G21K preprocesses, compiles, preprocesses, assembles, links and initializes the primes program, producing an ADSP-21000 executable. The runtime header and C runtime library are linked in automatically.

The `-v` command line switch causes G21K to display the invocation of each tool during each stage. Information about tool paths, tool names, intermediate files, and switches is produced.

2.8.4 Running & Examining primes.exe With CBUG

CBUG is used to control and observe execution of a program, and to examine and change the values of variables during runtime. The `-g` command line switch causes G21K to generate debugging information which describes variables, data types and control flow; CBUG uses this information to allow you to step through code and examine and set variables.

In this example, you invoke CBUG, step through the source code while allowing the executable to run on the simulator, and examine variables.

2.8.4.1 Invoking CBUG

CBUG is a menu option in the ADSP-21000 Family simulators. The simulator is designed to run from within Microsoft Windows. To invoke the ADSP-2106x SHARC simulator, double-click on its icon with the mouse. As the program begins to run, it will load default executable and architecture files as specified in its `.ini` file. Select the **Execution | Simulator Reset** menu function to prepare the simulator for new files. Then use the **File | Load** menu function to call up a dialog box where the file to be loaded is specified.

Note: The architecture file must be loaded first!

Select the `primes.ach` architecture file and click on the OK button. Repeat this process for the `primes.exe` executable file. (**Note:** Detailed information on using the simulator is available in the *ADSP-21000 Family Assembler Tools & Simulator Manual*.)

The simulator displays a menu bar at the top of the screen and a Program Memory window in the center of the screen.

To get CBUG started, follow these steps:

1. Open CBUG by selecting the **Execution | CBUG** menu function.
2. Begin running your program by choosing the **Execution | Run/Halt** menu function or by using the F4 hotkey.
3. Accept CBUG's confirmation query.
4. CBUG begins running your code.

CBUG automatically puts a breakpoint at `main()` in your program. The code in the runtime header initializes any global data, the C runtime stack, target wait states and banks, and interrupt handling before execution stops at `main()`.

2.8.4.2 Displaying Data

The CBUG window contains two subwindows, one for source code and one for status display, and command buttons at the top. If CBUG does not show your source, check the CBUG Status window for an error message.

Set up a display window to watch the array primes[ ] during execution:

1. Open a memory window using the **Memory | Memory** menu function.

2. Use the **Memory | GoTo** menu function and type primes as the expression to display.

3. The windows display changes to show the location where primes[] is stored. Remember that you can move this window so that it does not obscure your CBUG window.

Open another display window to watch the value of testnum . Execute one line of source code by clicking Next, and then open a third display window with the expression primes[index] .

CBUG is now set up to show what prime numbers have been found, what number is being tested and what prime number testnum is being checked against. You do not have to open a new display window to examine a variable. You can show the value of a variable at any point by choosing Print from the menu. The output of Print goes in the status window and is not updated.

2.8.4.3 Stepping

The source line where execution begins is highlighted.

1. Begin execution by clicking the Next button.

2. Click Next repeatedly until line 11 is highlighted.

3. Notice how primes[] and testnum change in their display windows as they are initialized.

4. Use Next to execute individual lines of code. Step through the inner loop a few times. You may want to display the inner loop control expression, !(testnum % primes[index]), to predict loop execution.

2.8.4.4 Using Breakpoints

Breakpoints let you stop execution when control reaches a certain location in the code so you don't have to step through each line of code.

1. Stop execution when a number is found to be prime, at line 26, by
   a) double-clicking on line 26,
   b) or by choosing Break from the Breaks menu and enter 26 to set a breakpoint.

2. Click Continue to run until a breakpoint is hit.

3. Let primes run to completion by removing the breakpoint at line 26 and setting a breakpoint at line 32 where the program exits.

4. Click Continue. CBUG runs for a few seconds.

5. You may stop execution by pressing any key; be sure to note the CBUG Status window if you don't tells you how to reestablish your CBUG context.

6. When execution halts, primes[] is filled with twenty prime numbers.

2.9 COMPILER RESTRICTIONS

Some applications will experience problems because of compiler conflicts. The following list of restrictions addresses some known conflicts to avoid.

- User libraries that expect or return double parameters need to be compiled with the same size doubles as `.c` files. For example, a library created with `-fno-short-double` (or created in release 3.0, which used 64-bit `doubles`) cannot be used with `.c` files compiled in this release without the switch. The functions that contain or return double parameters may fail. Also, any assembly routines that expect or return 64-bit `doubles` need to be changed or used with `.c` files compiled with `-fno-short-double`.

- The compiler uses the `21K.ach` architecture file as the default if you do not use the `-a` switch to specify another architecture file. The `21060.ach` architecture file is used only as the default when you don't use the `-a` switch and you don't have a `21K.ach` architecture file available in the `/adi_dsp/21k/lib` subdirectory.

- The compiler will not generate correct code if a global structure of exactly two words (64 bits) is passed by value with the optimizer (`-O`, `-O2`, or `-03`) on.

- The error

  ```
  asm operand requires impossible reload
  ```

  will be generated if a block of code uses too many registers. If this situation occurs, examine `asm()` statements and macros (such as the circular buffer macros) which contain `asm()` statements.

- An include path that ends with a `"\"`, the directory will not be seen by preprocessor. For example:

  ```
  g21k x.c -Id:\ -Id:\mypath\
  ```

  Neither include directory will be seen. Instead use

  ```
  g21k x.c -Id:\mypath -Id:
  ```

- If a C symbol is also used by the library, the simulator / emulator will only show the last symbol linked.

- Floating point emulation differences may cause slightly different compiler output. Code compiled with 386 with 387 emulation may be different than code compiled with 387 or 486.

- The `fract` type is not supported in this release. Users should `#define fract float` or `-Dfract=float` if their code uses `fract`.

- Floating constants that overflow during evaluation at compile time result in zero. For example:

```
#include <float.h>
volatile float fltmax=FLT_MAX;
main() {
 float f1=fltmax*fltmax;
   /* calculated by 21k at runtime, ok */
float f2=FLT_MAX*FLT_MAX;
   /* overflows on pc during compilation and evaluates to 0
*/
```

The first number evaluates to `INF`, correctly. The second number evaluates to 0.

- The character sequence `\E` is “escape.”
- It is illegal to use the following:

```
register float x asm(“f3”);
```

  Instead use the fixed-point register prefix, r:

```
register float x asm(“r3”);
```

- Interrupt handlers contain library functions that use `DAG` registers. If you also use these registers for circular buffers, the `DAG` value may change. Only use `I0` and `I1` for circular buffers because the library routines do not use these registers.
- Circular buffers used in two or more `*.c` files to be linked together cannot be used with the `-g` switch. The `CIRCULAR_BUFFER` macro can only be in one file with `-g`, whereas for normal operation, the `CIRCULAR_BUFFER` macro should be in every file to be linked so the compiler does not use `CIRCULAR_BUFFER` registers for other purposes.
- `CIRC_READ` and `CIRC_WRITE` modify values cannot be more than six bits wide. A workaround is to use `CIRC_MODIFY` (see the `circbuf.c` and `circbuf1.c` examples).
- `PM` circular buffers do not work.
- Circular buffers of types that require more than one memory location (such as complex numbers and `doubles`) do not work with the `CIRC_READ` and `CIRC_WRITE` macros.
- Stopping compilation with `Control-C` is unreliable.
- Do not call C functions from inside an `asm()` function.
- All `#include` filenames must have the `.h` extension.
- Strict prototype checking is enforced. For example:

```
extern foo(char);
foo(int i) {...}
```

  is legal in ANSI, but `G21K` complains about mismatch.

- Macro substitutions made with `#define` statement cannot begin with a number. For example:

  ```
  #define 2me i * j
  ```

  does not work.

- Double precision constants have a limit of 1.E64. Double precision variables are accurate to 54 bits. Double precision numbers are truncated rather than rounded.

- Do not use `G21K` for assembly if you want to pass down assembly defines. For example:

  ```
  g21k -DLABEL main.asm
  ```

  does not pass the define down to the assembler. While

  ```
  asm21k -DLABEL main.asm
  ```

  works as expected.

- `G21K` functions cannot return program memory structures.

- `G21K` generates signed numbers as the product of a multiplication. The results are bit exact. However, the runtime code generates incorrect overflow and underflow interrupts.