Fast Run Time Support Library

USER'S GUIDE

FASTRTS-SW-LIB-UG-V2.04.00.00

Copyright

Texas Instruments
12203 Southwest Freeway
Houston, TX 77477
http://www.ti.com/c2000

Revision Information

This is version V2.04.00.00 of this document, last updated on Mar 3, 2022.

Table of Contents

1 Introduction

The Texas Instruments TMS320C28x Floating Point Unit Fast Run Time Support (FASTRTS) Library is a collection of optimized math routines written for C2000 devices that support either a single precision Floating Point Unit (FPU32), an FPU32 with Triginometric Math Unit (TMU type 0), or a double precision FPU (FPU64).

These functions enable C/C++ programmers to take full advantage of the aforementioned hardware accelerators to speed up computation time. This document provides a description of each function included in the library.

chapter 2 provides a host of resources on the FPU in general, as well as training material.

chapter 3 describes the directory structure of the package.

chapter 4 provides step-by-step instructions on how to integrate the library into a project and use any of the math routines.

chapter 5 describes the single precision routines, with their accompanying variables, data types and structures.

chapter 6 describes the double precision routines, with their accompanying variables, data types and structures.

chapter 7 lists the performance of each routine.

chapter 8 provides a revision history of the library.

Examples have been provided for each library routine. They can be found in the *examples* directory. For the current revision, the newest examples using FPU64 have been written and validated on the *F2838x* device based *controlCard*. The FPU32 examples have been validated on F28002x and F28003x based *controlCard*.

2 Other Resources

The user can refer to the F2838x and device2 device TRMs for detailed description.

Also check out the TI C2000 portfolio at: http://www.ti.com/microcontrollers/c2000-real-time-control-mcus/overview

And don't forget the TI community website: http://e2e.ti.com

Building the FASTRTS libraries and examples require **Codegen Tools v20.2.1** or later.

3 Library Structure

By default, the library and source code is installed into the c2000ware directory under the sub-folder

`C:\ti\c2000\c2000ware_<version>\libraries\math\FPUfastRTS`

Figure. 3.1 shows the directory structure while the subsequent table 3.1 provides a description for each folder.

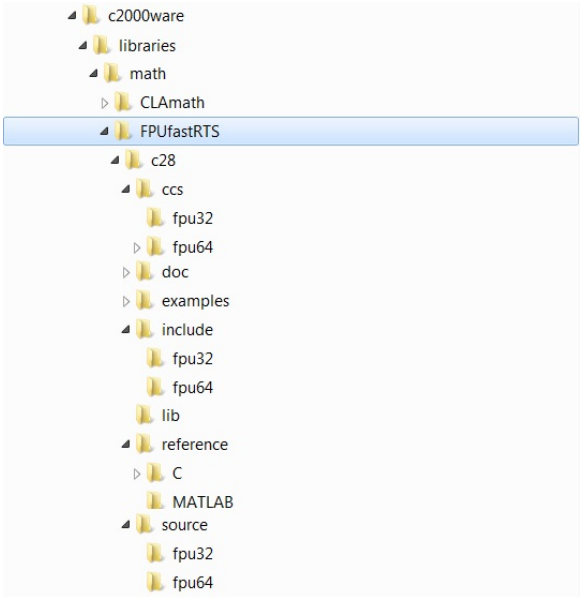

Figure 3.1: Directory Structure of the FASTRTS Library

Folder	Description
<base>	Base install directory. By default this is C:/ti/c2000/c2000ware_2_00_00_00/libraries/math/FPUfastRTS. For the rest of this document <base> will be omitted from the directory names.
<base>/ccs	Project files for the library. Allows the user to reconfigure, modify and re-build the library to suit their particular needs.
<base>/doc	Documentation for the current revision of the library including revision history.
<base>/examples	Examples that demonstrate usage of the library functions were built and validated for the F2838x, F28003x and F28002x devices using the CCS 10.x IDE.
<base>/examples/common	Device specific setup, linker command files and ROM symbol libraries for the examples. Each device gets its own sub-folder.
<base>/include/fpu32	Header files for the single precision floating point FASTRTS library. These include function prototypes and structure definitions.
<base>/include/fpu64	Header files for the double precision floating point FASTRTS library. These include function prototypes and structure definitions.
<base>/lib	Static libraries with EABI configuration (single and double precision).
<base>/source/fpu32	Source files for the single precision floating point FASTRTS library.
<base>/source/fpu64	Source files for the double precision floating point FASTRTS library.

Table 3.1: FPU FastRTS Library Directory Structure Description

The header files are sorted into two folders under *"include"*

- **fpu32**, single precision library header files
- **fpu64**, double precision library header files

The file *"fastrts.h"* is common to both libraries and defines new data types and macros. This legacy header file, *"C28x_FPU_FastRTS.h"*, is superceded by this new file, but it may be used without it.

The source files are sorted into two folders under *“source”*

- **fpu32** - single precision library header files
- **fpu64** - double precision library header files

The source code contains the arithmetic and trigonometric routines, and in the case of the double precision library, the Fast Integer Division (FID) routines as well.

4 Using the FPU Library

The source code and project(s) for the libraries are provided. The user may import the library project(s) into CCSv10 and be able to view and modify the source code for all routines and lookup tables. (see Figure 4.1)

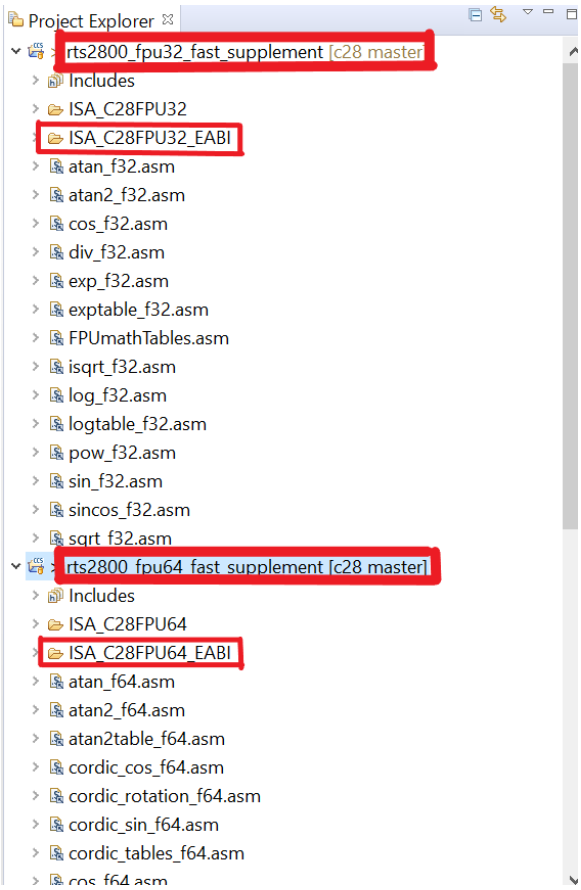

Figure 4.1: FPU Library Project View

The single precision library has both COFF and EABI build configurations i.e. **ISA_C28FPU32**, **ISA_C28FPU32_EABI** (Figure 4.2) while the double precision library has only EABI configuration **ISA_C28FPU64_EABI** (Figure 4.3).

The **ISA_C28FPU32** and **ISA_C28FPU32_EABI** configurations are built with the **–float_support=fpu32** and **–tmu_support=tmu0** run-time support options enabled. Running a build on these configuration will generate **rts2800_fpu32_fast_supplement_coff.lib** and **rts2800_fpu32_fast_supplement_eabi.lib** in the lib folder. An **index library rts2800_fpu32_fast_supplement.lib** is created using **libinfo2000** tool that can be linked against instead of directly linking to a coff or eabi-specific library. Thus any example irrespective of

COFF/EABI can just link to this index library, the linker then uses the index library to automatically choose the appropriate version of the library to use based on the build attribute of the particular example.

NOTE: ATTEMPTING TO LINK IN THIS LIBRARY INTO A PROJECT THAT DOES NOT HAVE THE FLOAT_SUPPORT SET TO FPU32 WILL RESULT IN A COMPILER ERROR ABOUT MISMATCHING INSTRUCTION SET ARCHITECTURES

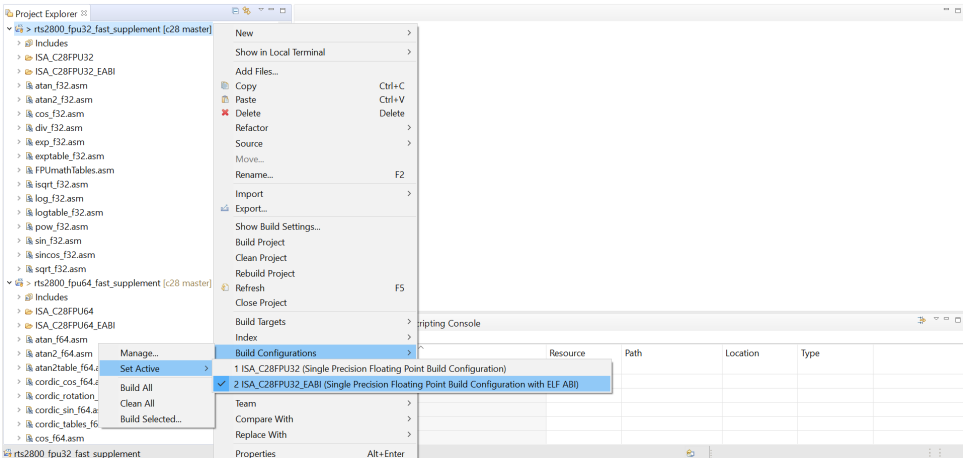

Figure 4.2: Single Precision Library Build Configuration

The **ISA_C28FPU64_EABI** configuration is built with the **–float_support=fpu64** run-time support and **–idiv_support=idiv0** option enabled. Running a build on this configuration will generate **rts2800_fpu64_fast_supplement.lib** in the lib folder.

NOTE: ATTEMPTING TO LINK IN THIS LIBRARY INTO A PROJECT THAT DOES NOT HAVE THE FLOAT_SUPPORT SET TO FPU64 WILL RESULT IN A COMPILER ERROR ABOUT MISMATCHING INSTRUCTION SET ARCHITECTURES

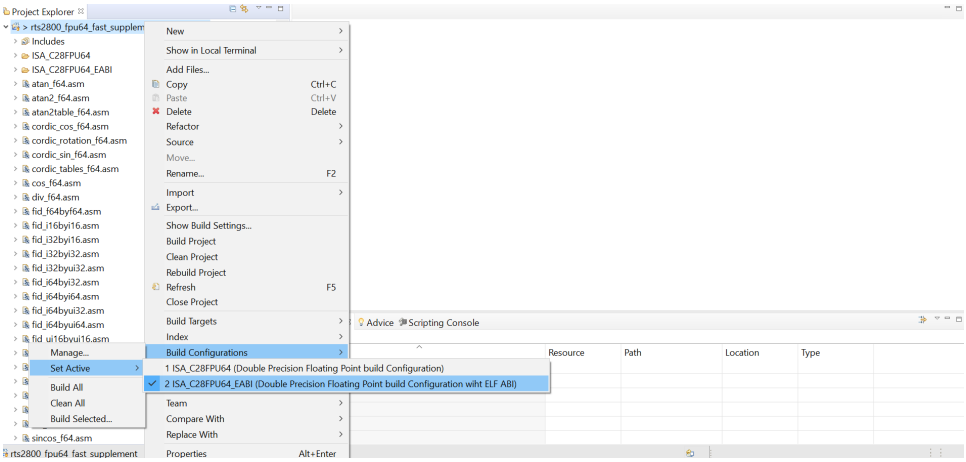

Figure 4.3: Double Precision Library Build Configuration

The table below summarizes all the build configurations and provides description of all single precision and double precision FPU FASTRTS libraries included in the folder.

Floating-Point precision	Library Name	Library Format	Build Configuration	Description
32-bit	rts2800_fpu32_fast_supplement_coff.lib	COFF	ISA_C28FPU32	Single precision COFF FPU FASTRTS library
	rts2800_fpu32_fast_supplement_eabi.lib	EABI	ISA_C28FPU32_EABI	Single precision EABI FPU FASTRTS library
	rts2800_fpu32_fast_supplement.lib	-	-	Index library for above single precision COFF and EABI variants, this needs to be used in project's linker options
64-bit	rts2800_fpu64_fast_supplement.lib	EABI	ISA_C28FPU64_EABI	Double Precision FPU FASTRTS library

Table 4.1: Build configurations and library description

NOTE: THE C2000 COMPILER WILL ONLY SUPPORT 64-BIT FLOATING-POINT INSTRUCTIONS IN EABI FORMAT, THAT IS WHY NO DOUBLE PRECISION COFF LIBRARY IS BEING PROVIDED. ANY PROJECT REQUIRING 64-BIT FLOATING-POINT SUPPORT MUST USE EABI. FOR MORE INFORMATION, PLEASE REFER TO TMS320C28X OPTIMIZING C/C++ COMPILER USERâĂŹS GUIDE WWW.TI.COM/LIT/SPRU514

NOTE: DO TAKE CARE WHILE ADDING/COPYING THE SINGLE PRECISION FPU FASTRTS LIBRARIES TO THE CCS PROJECTS I.E. ALL THE VERSIONS (COFF, EABI) OF THE LIBRARY INCLUDING THE INDEX LIBRARY NEED TO COPIED TO THE PROJECTS SO AS TO ALLOW THE LINKER TO PICK THE CORRECT VERSION OF THE LIBRARY. THIS STEP IS NOT REQUIRED IF THE LIBRARIES ARE BEING LINKED THROUGH PROJECT BUILD OPTIONS. THUS THE RECOMMENDED PRACTICE IS TO ALWAYS LINK THE FPU FASTRTS LIBRARIES INSTEAD OF COPYING.

To begin integrating the library into your project follow these steps:

1. Go to the **Project Properties->Resources->Linked Resources->Path Variables Tab** and add a new variable (see Figure 4.4), **FPU_FASTRTS_LIB_ROOT**, and set its path to the root directory of the floating point Fast Runtime Support library in c2000ware; this is usually the **c28** folder.

   Each example project requires two variables that are device specific,

   - **DRIVERLIB_ROOT**, points to the driver library of the target device on which the code will run
   - **FASTRTS_EXAMPLES_COMMON**, points to the target specific device under the folder **c28/examples/common/**<**device**>; this folder contains the linker command files.

- **DSP_EXAMPLES_COMMONSRC**, contains basic system setup code used in each of the examples; they can be customized to meet the user's needs.

When switching to another compatible device, i.e. one having a hardware floating point unit, change the foregoing variables with the appropriate target device name; there must be a sub-folder in the *"common"* directory with the target device name.

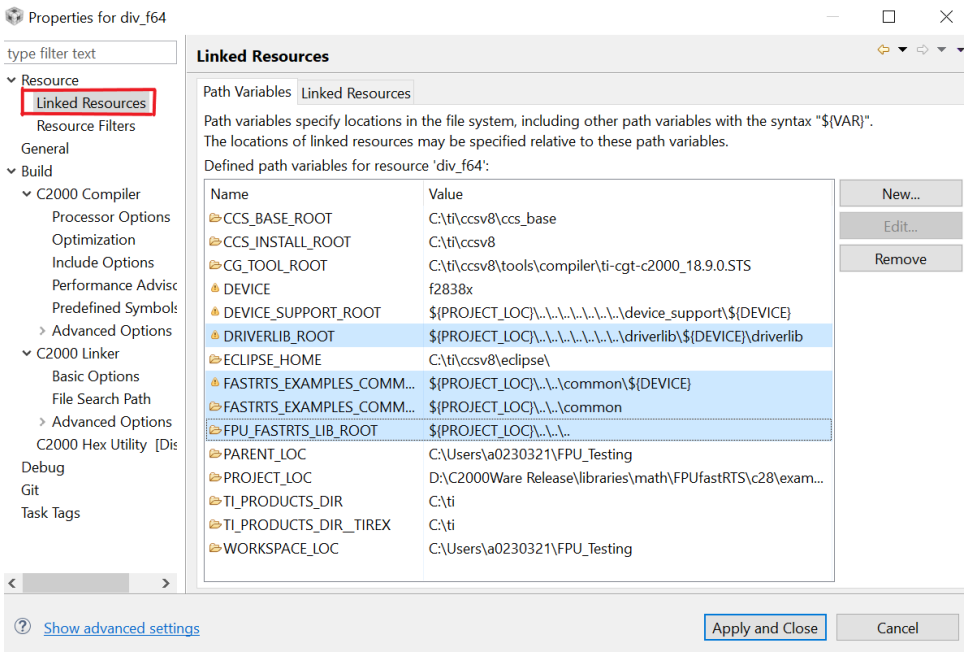

Figure 4.4: Creating a new build variable

Add the new path, **FPU_FASTRTS_LIB_ROOT/include**, to the *Include Options* section of the project properties (Figure 4.5). This option tells the compiler where to find the library header files. In addition, you must add the driver library (**DRIVERLIB_ROOT**)path as well as the common folder (**FASTRTS_EXAMPLES_COMMON**) and (**FASTRTS_EXAMPLES_COMMONSRC**) path for the target device in use.

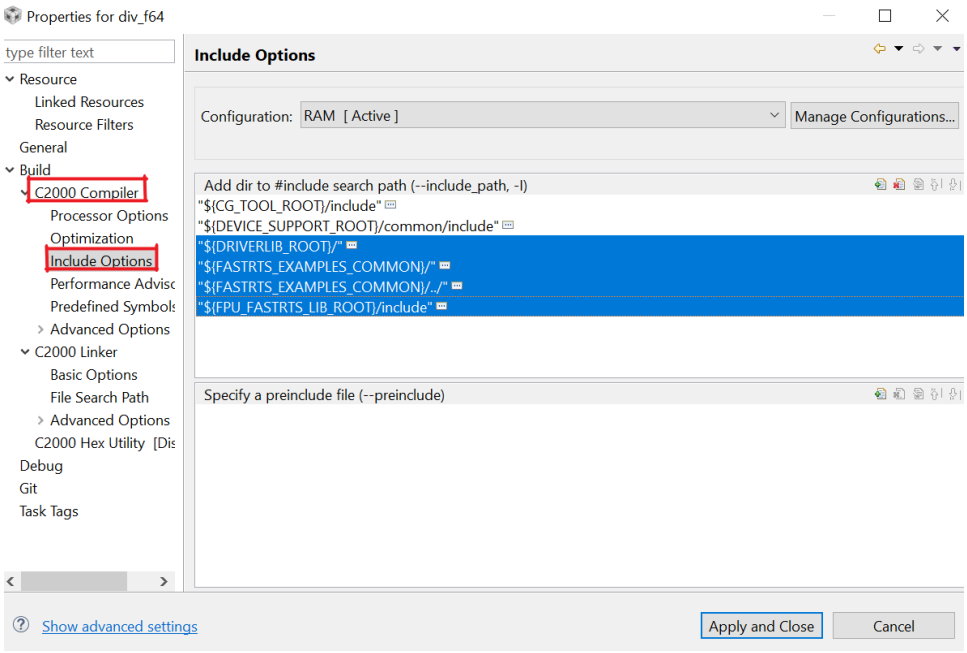

Figure 4.5: Adding the Library Header Path to the Include Options

2. For the single prcecision library set the **–float_support** option to fpu32 in the **Runtime Model Options**. The user may optionally turn on **tmu_support**; none of the library functions use TMU instructions but the library is built with TMU support turned on (Figure 4.6).

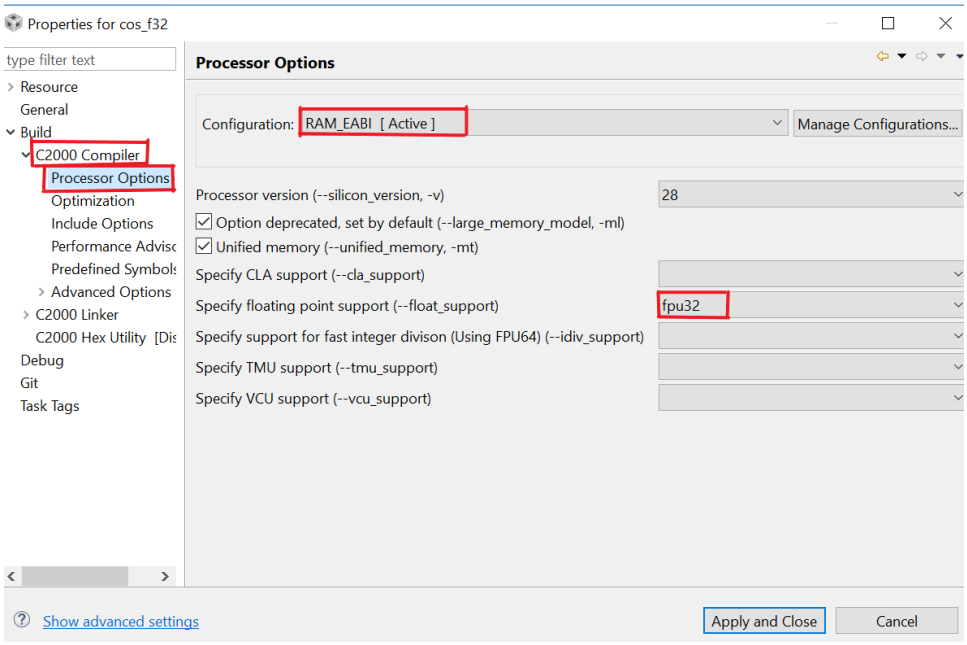

Figure 4.6: Turning on FPU32 support

3. For the double prcecision library set the **–float_support** option to fpu64 in the **Runtime Model**

Options as shown in Figure 4.7.

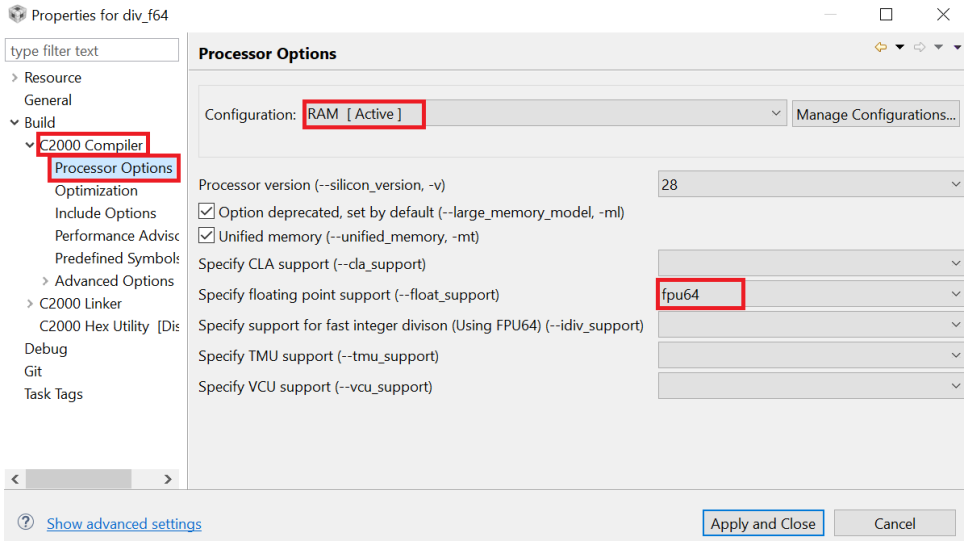

Figure 4.7: Turning on FPU64 support

4. Add the name of the library and its location to the **File Search Path** as shown in Figure 4.8 and Figure 4.9. For the single precision DSP library add **rts2800_fpu32_fast_supplement.lib**, and **rts2800_fpu64_fast_supplement.lib** for the double precision library.

   NOTE: BE SURE TO ENABLE FLOAT_SUPPORT (AND, OPTIONALLY, TMU_SUPPORT IF THE DEVICE SUPPORTS IT) IN YOUR PROJECT PROPERTIES

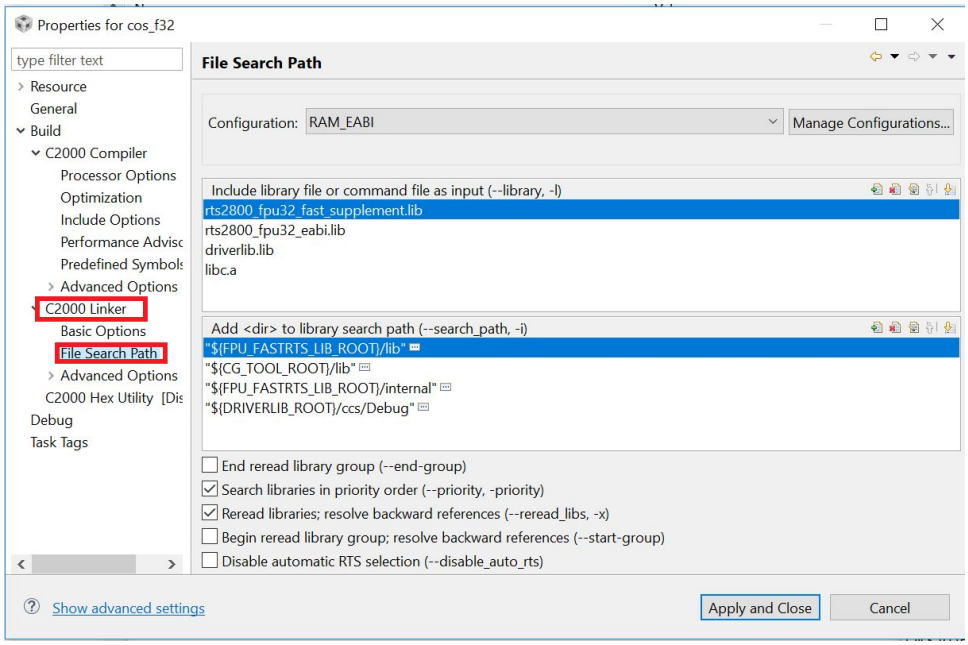

Figure 4.8: Adding the library and location to the file search path

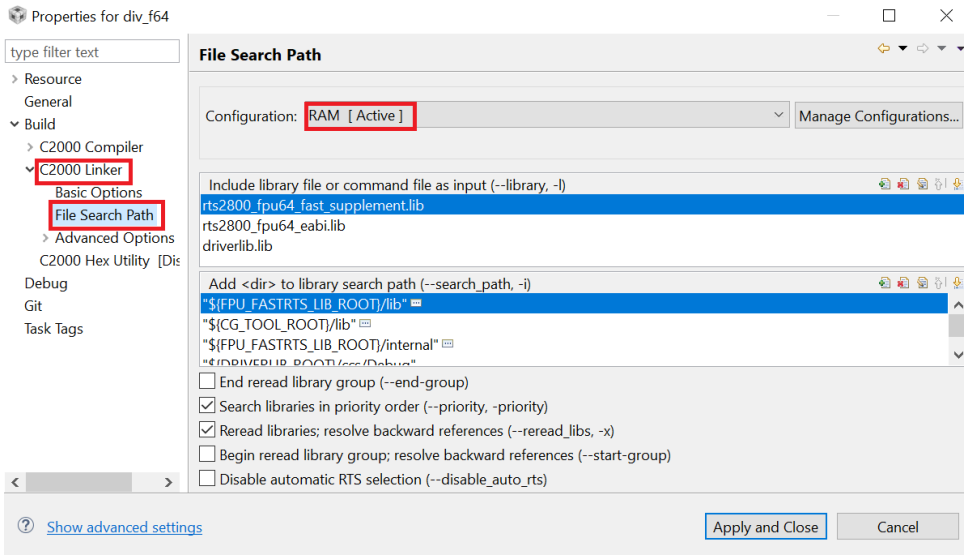

Figure 4.9: Adding the library and location to the file search path

5. In the project properties clear the *Runtime Support Library* option, as shown in Figure 4.10, or set it to auto. If you wish to include one of the other run time support libraries in your project, add it from the *File Search Path* option, as shown in Figure 4.8 and Figure 4.9, and place the fastRTS library above the other RTS library and select the **--priority** check box to ensure the compiler pulls the math routines from the fastRTS but the support routines (like string operations) from the other RTS library.

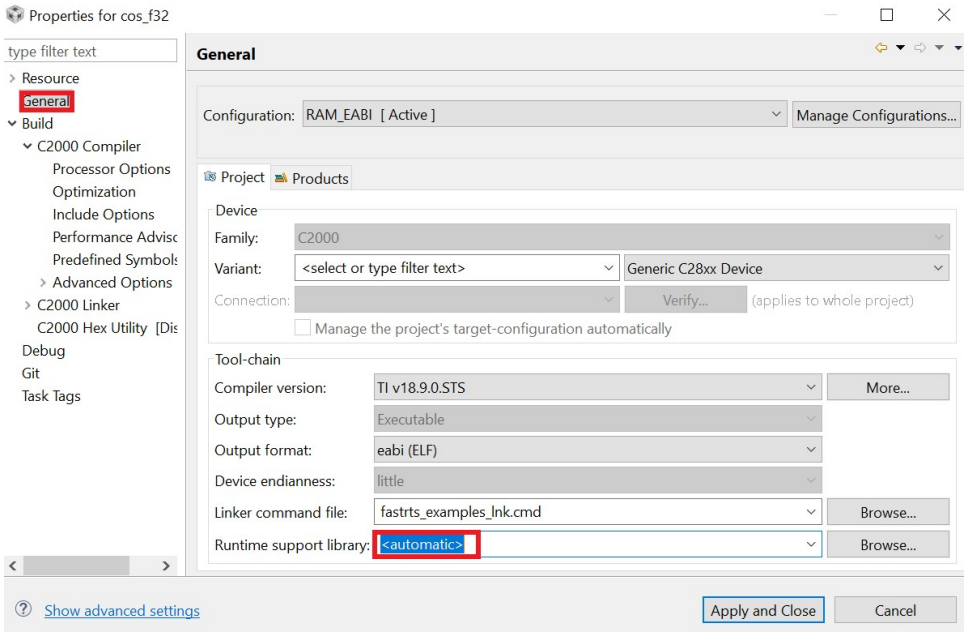

Figure 4.10: Selecting the Runtime Support Library

6. For sin32, cos32 and sincos32 examples on devices with limited memory, change optimisation

settings as shown in Figure 4.11.

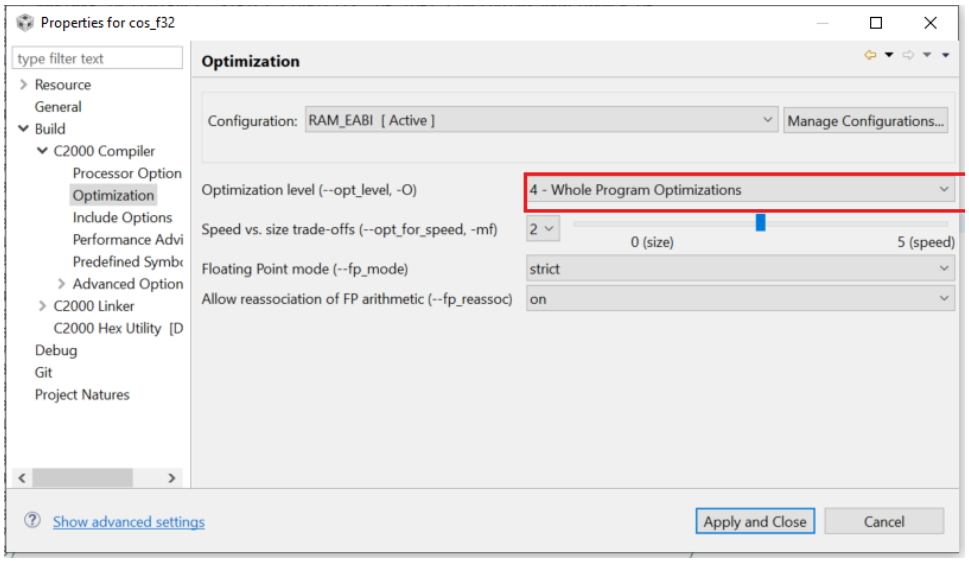

Figure 4.11: Changing Optimisation Settings

7. If you wish to run the examples on another device, change the device strings on Properties -> Build -> Variables and Properties -> Resources -> Linked Resources.

After you build the project, check the .map file. This file is typically in the Debug directory of the project folder. This file will show which functions are being used from which library.

If the fastRTS library is linked in first, you will see something like the listing below (Figure 4.12). Notice the atan function is pulled in from the fastRTS library.

```
.text.2    0    00008800    00000800
                 00008800    000000b2     driverlib.lib : sysctl.obj (.text:_SysCtl_setClock)
                 00008bb2    000000b0     rts2800_fpu32.lib : fputs.obj (.text)
                 00008962    000000a7     driverlib.lib : sysctl.obj (.text:_SysCtl_isPLLValid)
                 00008a09    0000009f     fastrts_examples_main.obj (.text)
                 00008aa8    00000089     rts2800_fpu32.lib : setvbuf.obj (.text)
                 00008b31    00000073                       : fflush.obj (.text)
                 00008ba4    00000067                       : _io_perm.obj (.text)
                 00008c0b    0000005f                       : fclose.obj (.text)
                 00008c6a    0000005e     driverlib.lib : sysctl.obj (.text:_DCC_setCounterSeeds)
                 00008cc8    00000056                   : sysctl.obj (.text:_SysCtl_selectOscSource)
                 00008d1e    00000056     rts2800_fpu32.lib : boot28.obj (.text)
                 00008d74    0000004f     fastrts_atan.obj (.text)
                 00008dc3    0000004a     rts2800_fpu32_fast_supplement.lib : atan_f32.obj (.text)
                 00008e0d    0000003d     rts2800_fpu32.lib : fseek.obj (.text)
                 00008e4a    0000002a     fastrts_examples_setup.obj (.text)
                 00008e74    00000029     rts2800_fpu32.lib : exit.obj (.text)
                 00008e9d    00000025     driverlib.lib : sysctl.obj (.text:_DCC_enableSingleShotMode)
                 00008ec2    00000024                   : sysctl.obj (.text:_SysCtl_selectXTAL)
                 00008ee6    00000024     rts2800_fpu32.lib : cpy_tbl.obj (.text)
                 00008f0a    00000022     driverlib.lib : sysctl.obj (.text:_SysCtl_pollX1Counter)
                 00008f2c    0000001f                   : sysctl.obj (.text:_DCC_setCounter1ClkSource)
                 00008f4b    0000001e     rts2800_fpu32.lib : ll_cmp28.obj (.text)
                 00008f69    0000001e                       : memcpy.obj (.text)
```

Figure 4.12: Determining the Library Association of a Function

If the normal RTS library is linked in first, you will see something like this. Notice the atan function comes from the normal RTS library library.

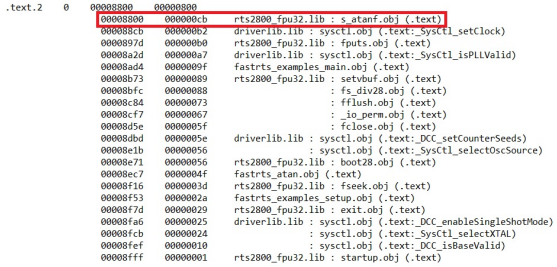

Figure 4.13: Determining the Library Association of a Function

It is possible to place functions from the library at specific locations in memory. You do this in the linker command file with the `--library` option. For example you could place atan() in the section **"ramfuncs"** by modifying the linker command file as follows,

```
ramfuncs   : LOAD = FLASHC,
             RUN = RAMLS1,
             RUN_START(_RamfuncsRunStart),
             LOAD_START(_RamfuncsLoadStart),
             LOAD_SIZE(_RamfuncsLoadSize),
             PAGE = 0
             {
             --library=rts2800_fpu32_fast_supplement.lib<atan_f32.obj> (.text)
             }
```

5 Application Programming Interface (FPU32)

The source code for the single precision library can be found under source/fpu32. Table 6.1 lists all the available routines

Arithmetic and Trigonometric	
atan2f	float32 atan2f(float32 , float32);
atanf	float32 atanf(float32);
cosf	float32 cosf(float32);
expf	float32 expf(float32);
FS$$DIV	float32 FS$$DIV(float32 , float32);
isqrtf	float32 isqrtf(float32);
logf	float32 logf(float32);
powf	float32 powf(float32 , float32);
sincosf	void sincosf(float32 , float32* ,float32 *);
sinf	float32f sin(float32);
sqrtf	float32f sqrt(float32);

Table 5.1: List of Functions

The examples for each of these routines was built using **CGT v20.2.1** with the following options:

```
-v28 -mt -ml -g --diag_warning=225 --float_support=fpu32 --tmu_support=tmu0
--define=CPU1
```

Each example has at least two build configurations, **RAM** and **FLASH**. Although only RAM configurations are validated for F28002x and F28003x device.

5.1 Fast RTS (FPU32) Library Definitions and Types

Functions

float32_t FS $$DIV (float32_t X, float32_t Y)

float32_t acosf (float32_t X)

float32_t asinf (float32_t X)

float32_t atan2f (float32_t Y, float32_t X)

float32_t atanf (float32_t X)

float32_t cosf (float32_t X)

float32_t expf (float32_t X)

float32_t isqrtf (float32_t X)

float32_t logf (float32_t X)

float32_t powf (float32_t X, float32_t Y)

void sincosf (float32_t radian, float32_t *PtrSin, float32_t *PtrCos)

float32_t sinf (float32_t X)

float32_t sqrtf (float32_t X)

5.1.1 Detailed Description

5.1.2 Function Documentation

5.1.2.1

float32_t X,

float32_t Y)

Single-Precision Floating-Point Division.

Prototype:

```
$$DIV float32_t FS $$DIV(
```

Description:

Replaces the single-precision division operation from the standard RTS library. This function uses a Newton-Raphson algorithm.

In C code, an expression of the type Z = Y/X will invoke FS$$DIV

[1]Parameters `in` *X* single precision floating point numerator

`in` *Y* single precision floating point denominator

Returns the quotient Attention

This division routine computes y/x by first estimating (1/x), performing a few iterations of Newton-Raphson approximation to improve the precision of 1/x, and then finally multiplying that estimate by y; it essentially computes y/x as (1/x)$*$y. If 1/x is not perfectly representable in the floating point format it can lead to inaccuracies in the final result. The user is encouraged to use the standard runtime support library division routine if accuracy takes priority over speed.

These are the special cases for division

(0.0/0.0)	= +infinity	
(+FLT_MAX/+FLT_MAX)	= 0.0	LUF = 1
(-FLT_MAX/+FLT_MAX)	= -0.0	LUF = 1
(+FLT_MAX/-FLT_MAX)	= 0.0	LUF = 1
(-FLT_MAX/-FLT_MAX)	= -0.0	LUF = 1
(+FLT_MIN/+FLT_MAX)	= 0.0	LUF = 1
(-FLT_MIN/+FLT_MAX)	= -0.0	LUF = 1
(+FLT_MIN/-FLT_MAX)	= 0.0	LUF = 1
(-FLT_MIN/-FLT_MAX)	= -0.0	LUF = 1

Table 5.2: Special Cases for Division

Note

This is a standard C math function and requires "math.h" to be included

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

25	Cycle count includes the call and return

Table 5.3: Performance Data

5.1.2.2 float32_t acosf (**float32_t** X)

Single-Precision Floating-Point ACOS (radians)

[1]Parameters `in` *X* single precision floating point argument must be in the range $[-,]$

Returns the arc tangent of a floating-point argument X. The return value is an angle in the range $[-\pi, \pi]$ radians. Note

This is a standard C math function and requires "math.h" to be included

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

5.1.2.3 float32_t asinf (

60	Cycle count includes the call and return

Table 5.4: Performance Data

float32_t
X)

Single-Precision Floating-Point ASIN (radians)

[1]Parameters `in` *X* single precision floating point argument must be in the range $[-,]$

Returns the arc tangent of a floating-point argument X. The return value is an angle in the range $[-\pi, \pi]$ radians. Note

This is a standard C math function and requires "math.h" to be included

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

55	Cycle count includes the call and return

Table 5.5: Performance Data

5.1.2.4 float32_t atan2f (

float32_t
Y,
float32_t
X)

Single-Precision Floating-Point ATAN2 (radians)

[1]Parameters `in` *Y* first single precision floating point argument

---

`in` *X* second single precision floating point argument

Returns the 4-quadrant arctangent of floating-point arguments X/Y. The return value is an angle in the range $[-\pi, \pi]$ Note

This is a standard C math function and requires "math.h" to be included

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

50	Cycle count includes the call and return

Table 5.6: Performance Data

5.1.2.5 float32_t atanf (

float32_t
X)

Single-Precision Floating-Point ATAN (radians)

[1]Parameters `in` *X* single precision floating point argument

---

Returns the arc tangent of a floating-point argument X. The return value is an angle in the range $[-\pi, \pi]$ radians. Note

This is a standard C math function and requires "math.h" to be included

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

49	Cycle count includes the call and return

Table 5.7: Performance Data

5.1.2.6 float32_t cosf (

float32_t
X)

Single-Precision Floating-Point Cosine (radians)

[1]Parameters `in` *X* single precision floating point argument

---

Returns the cosine of a floating-point argument X (in radians) using table look-up and Taylor series expansion between the look-up table entries. Note

This is a standard C math function and requires "math.h" to be included

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

38	Cycle count includes the call and return

Table 5.8: Performance Data

5.1.2.7 float32_t expf (

float32_t
X)

Single-Precision Floating-Point Exponent.

[1]Parameters `in` *X* single precision floating point argument

---

Returns the exponent of a floating-point argument X using table look-up and Taylor series expansion of the fractional part of the argument. Note

This is a standard C math function and requires "math.h" to be included

the input domain is limited to +-log(FLT_MAX) ($<$ +-89)

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

61	Cycle count includes the call and return

Table 5.9: Performance Data

5.1.2.8 float32_t isqrtf (**float32_t** X)

Single-Precision Floating-Point 1.0/Square Root.

[1]Parameters `in` *X* single precision floating point argument

---

Returns 1.0 /square root of a floating-point argument X using a Newton- Raphson algorithm. Attention

isqrt(FLT_MAX) and isqrt(FLT_MIN) set the LUF flag.

isqrt(-FLT_MIN) will set both the LUF and LVF flags.

isqrt(0.0) sets the LVF flag.

If X is negative, isqrt(X) will set LVF and return 0.0. Note

This function is not included in the standard RTS library. It is typically computed as *1.0L/sqrt*(X) . To use this function you must modify your code to instead call isqrt(X).

When migrating from an IQmath project, you can modify the IQmath header file to use isqrt(X) when configured for FLOAT_MATH.

This is not a standard C Math function; it requires the "C28x_FPU_FastRTS.h" header instead of "math.h"

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

26	Cycle count includes the call and return

Table 5.10: Performance Data

5.1.2.9 float32_t logf (**float32_t** X)

Single-Precision Floating-Point Natural Logarithm.

[1]Parameters `in` *X* single precision floating point argument

---

Returns the natural logarithm of a floating-point argument X Note

This is a standard C math function and requires "math.h" to be included

the input must be greater than or equal to 1

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

51	Cycle count includes the call and return

Table 5.11: Performance Data

5.1.2.10 float32_t powf (

float32_t
X,
float32_t
Y)

Single-Precision Floating-Point Power Function.

[1]Parameters `in` *X* single precision floating point base argument

`in` *Y* single precision floating point exponent argument

Returns X^Y Note

This is a standard C math function and requires "math.h" to be included

the input X must be greater than or equal to 1

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

109	Cycle count includes the call and return

Table 5.12: Performance Data

5.1.2.11 void sincosf (

float32_t
radian,
float32_t *
PtrSin,
float32_t *
PtrCos)

Single Precision Floating Point Sine and Cosine (radians)

Returns both the sine and cosine of a floating-point argument X (in radians) using table look-up and Taylor series expansion between the look-up table entries.

[1]Parameters `in` *X* single precision floating point argument

`out` *PtrSin* pointer to the sine of the argument

`out` *PtrCin* pointer to the cosine of the argument

Returns none Note

This is not a standard C Math function; it requires the "C28x_FPU_FastRTS.h" header instead of "math.h"

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

44	Cycle count includes the call and return

Table 5.13: Performance Data

5.1.2.12 float32_t sinf (**float32_t** X)

Single-Precision Floating-Point Sine (radians)

[1]Parameters `in` *X* single precision floating point argument

Returns the sine of a floating-point argument X (in radians) using table look-up and Taylor series expansion between the look-up table entries. Note

This is a standard C math function and requires "math.h" to be included

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably.

38	Cycle count includes the call and return

Table 5.14: Performance Data

5.1.2.13 float32_t sqrtf (**float32_t** X)

Single-Precision Floating-Point Square Root.

[1]Parameters `in` *X* single precision floating point argument

Returns the square root of a floating-point argument X using a Newton- Raphson algorithm. Note

This is a standard C math function and requires "math.h" to be included

For COFF executables float and double are both single precision 32-bit floating point types, therefore, the double and float variant of this function may be used interchangeably. Attention

sqrt(FLT_MAX) and sqrt(FLT_MIN) set the LUF flag.

sqrt(-FLT_MIN) will set both the LUF and LVF flags.

sqrt(0.0) sets the LVF flag.

If X is negative, sqrt(X) will set LVF and return 0.0.

This square root routine computes $\sqrt{x}$ by first estimating $\frac{1}{\sqrt{x}}$, performing a few iterations of Newton-Raphson approximation to improve the precision of $\frac{1}{\sqrt{x}}$, and then finally multiplying that estimate by x; it essentially computes $\sqrt{x}$ as $\frac{1}{\sqrt{x}} \times x$. If $\frac{1}{\sqrt{x}}$ is not perfectly representable in the floating point format it can lead to inaccuracies in the final result. The user is encouraged to use the standard runtime support library division routine if accuracy takes priority over speed.

29	Cycle count includes the call and return

Table 5.15: Performance Data

6 Application Programming Interface (FPU64)

The source code for the double precision library can be found under source/fpu64. Table 6.1 lists all the available routines

Arithmetic and Trigonometric	
atan2	float64_t atan2(float64_t , float64_t);
atan	float64_t atan(float64_t);
cos	float64_t cos(float64_t);
FD$$DIV	float64_t FD$$DIV(float64_t , float64_t);
isqrt	float64_t isqrt(float64_t);
sincos	void sincos(float64_t , float64_t* ,float64_t *);
sin	float64_t sin(float64_t);
sqrt	float64_t sqrt(float64_t);
Fast Integer Division	
FID_ui32byui32	void FID_ui32byui32(uint32_t *, uint32_t *);
FID_i32byi32_t	void FID_i32byi32_t(int32_t *, int32_t *);
FID_i32byi32_m	void FID_i32byi32_m(int32_t *, int32_t *);
FID_i32byi32_e	void FID_i32byi32_e(int32_t *, int32_t *);
FID_i32byui32	void FID_i32byui32(int32_t *, int32_t *);
FID_ui64byui32	void FID_ui64byui32(uint64_t *, uint64_t *);
FID_i64byi32_t	void FID_i64byi32_t(int64_t *, int64_t *);
FID_i64byi32_m	void FID_i64byi32_m(int64_t *, int64_t *);
FID_i64byi32_e	void FID_i64byi32_e(int64_t *, int64_t *);
FID_i64byui32	void FID_i64byui32(int64_t *, int64_t *);
FID_ui64byui64	void FID_ui64byui64(uint64_t *, uint64_t *);
FID_i64byi64_t	void FID_i64byi64_t(int64_t *, int64_t *);
FID_i64byi64_m	void FID_i64byi64_m(int64_t *, int64_t *);
FID_i64byi64_e	void FID_i64byi64_e(int64_t *, int64_t *);
FID_i64byui64	void FID_i64byui64(int64_t *, int64_t *);
FID_ui16byui16	void FID_ui16byui16(uint16_t *, uint16_t *);
FID_i16byi16_t	void FID_i16byi16_t(int16_t *, int16_t *);
FID_i16byi16_m	void FID_i16byi16_m(int16_t *, int16_t *);
FID_i16byi16_e	void FID_i16byi16_e(int16_t *, int16_t *);
FID_ui32byui16	void FID_ui32byui16(uint32_t *, uint32_t *);
FID_i32byi16_t	void FID_i32byi16_t(int32_t *, int32_t *);
FID_i32byi16_m	void FID_i32byi16_m(int32_t *, int32_t *);
FID_i32byi16_e	void FID_i32byi16_e(int32_t *, int32_t *);
FID_f64byf64	void FID_f64byf64(float64_t *, float64_t *, float64_t *);

Table 6.1: List of Functions

6.1 Fast RTS Definitions and Types

Data Structures

float32u_t
float64u_t

Macros

LIBRARY_VERSION
ZERO_BY_ZERO_EQ_INF
ZERO_BY_ZERO_EQ_ZER0

Functions

float64u_t __c28xabi_div (float64u_t X, float64u_t Y)
float64u_t atan (float64u_t X)
float64u_t atan2 (float64u_t Y, float64u_t X)
float64u_t cos (float64u_t X)
float64u_t isqrt (float64u_t X)
float64u_t sin (float64u_t X)
void sincos (float64u_t radian, float64u_t *PtrSin, float64u_t *PtrCos)
float64u_t sqrt (float64u_t X)

6.1.1 Detailed Description

6.1.2 Data Structure Documentation

6.1.2.1 float32u_t

Definition:

```
typedef struct
{
    uint32_t ui32;
    int32_t i32;
    float f32;
}
float32u_t
```

Members:

ui32 Unsigned long representation.

i32 Signed long represntaion.

f32 Single precision (32-bit) representation.

Description:

HASH(0x55d8a8fa0c68)

6.1.2.2 float64u_t

Definition:

```
typedef struct
{
    uint64_t ui64;
    int64_t i64;
    float64_t f64;
}
float64u_t
```

Members:

ui64 Unsigned long long representation.

i64 Signed long long represntaion.

f64 Double precision (64-bit) representation.

Description:

HASH(0x55d8a8fa10a0)

6.2 Fast Integer Division Definitions and Types

Functions

void FID_f64byf64 (double *p_num, double *p_den, double *p_quo)
void FID_i16byi16_e (int16_t *p_num_rem, int16_t *p_den_quo)
void FID_i16byi16_m (int16_t *p_num_rem, int16_t *p_den_quo)
void FID_i16byi16_t (int16_t *p_num_rem, int16_t *p_den_quo)
void FID_i32byi16_e (int32_t *p_num_rem, int32_t *p_den_quo)
void FID_i32byi16_m (int32_t *p_num_rem, int32_t *p_den_quo)
void FID_i32byi16_t (int32_t *p_num_rem, int32_t *p_den_quo)
void FID_i32byi32_e (int32_t *p_num_rem, int32_t *p_den_quo)
void FID_i32byi32_m (int32_t *p_num_rem, int32_t *p_den_quo)
void FID_i32byi32_t (int32_t *p_num_rem, int32_t *p_den_quo)
void FID_i32byui32 (int32_t *p_num_rem, int32_t *p_den_quo)
void FID_i64byi32_e (int64_t *p_num_rem, int64_t *p_den_quo)
void FID_i64byi32_m (int64_t *p_num_rem, int64_t *p_den_quo)
void FID_i64byi32_t (int64_t *p_num_rem, int64_t *p_den_quo)
void FID_i64byi64_e (int64_t *p_num_rem, int64_t *p_den_quo)
void FID_i64byi64_m (int64_t *p_num_rem, int64_t *p_den_quo)
void FID_i64byi64_t (int64_t *p_num_rem, int64_t *p_den_quo)
void FID_i64byui32 (int64_t *p_num_rem, int64_t *p_den_quo)
void FID_i64byui64 (int64_t *p_num_rem, int64_t *p_den_quo)
void FID_ui16byui16 (uint16_t *p_num_rem, uint16_t *p_den_quo)
void FID_ui32byui16 (uint32_t *p_num_rem, uint32_t *p_den_quo)
void FID_ui32byui32 (uint32_t *p_num_rem, uint32_t *p_den_quo)
void FID_ui64byui32 (uint64_t *p_num_rem, uint64_t *p_den_quo)
void FID_ui64byui64 (uint64_t *p_num_rem, uint64_t *p_den_quo)

6.2.1 Detailed Description

6.2.2 Function Documentation

6.2.2.1 void FID_f64byf64 (

double *
p_num,
double *
p_den,

double ∗
p_quo)

Division: f64/f64.

[1]Parameters `out` *p_num,pointer* to the dividend

`out` *p_den,pointer* to the divisor

`out` *p_quo,pointer* to the quotient

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

36	Cycle count includes the call and return

Table 6.2: Performance Data

6.2.2.2 void FID_i16byi16_e (

int16_t ∗
p_num_rem,
int16_t ∗
p_den_quo)

Division: i16/i16 (Euclidean)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

---

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

---

$$Q = sgn(n)\lfloor\frac{dividend}{|divisor|}\rfloor$$

$$R = dividend - |divisor| \times \lfloor\frac{dividend}{|divisor|}\rfloor$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

26	Cycle count includes the call and return

Table 6.3: Performance Data

6.2.2.3 void FID_i16byi16_m (

int16_t ∗
p_num_rem,
int16_t ∗
p_den_quo)

Division: i16/i16 (Modulo or Floored)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

---

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

---

$$Q = \lfloor \frac{dividend}{divisor} \rfloor$$

$$R = dividend - divisor \times \lfloor \frac{dividend}{divisor} \rfloor$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

26	Cycle count includes the call and return

Table 6.4: Performance Data

6.2.2.4 void FID_i16byi16_t (

int16_t ∗
p_num_rem,
int16_t ∗
p_den_quo)

Division: i16/i16 (Truncated)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

---

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

---

$$Q = trunc(\frac{dividend}{divisor})$$

$$R = dividend - divisor \times trunc(\frac{dividend}{divisor})$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

26	Cycle count includes the call and return

Table 6.5: Performance Data

6.2.2.5 void FID_i32byi16_e (**int32_t** * p_num_rem, **int32_t** * p_den_quo)

Division: i16/i16 (Euclidean)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = sgn(n)\lfloor\frac{dividend}{|divisor|}\rfloor$$

$$R = dividend - |divisor| \times \lfloor\frac{dividend}{|divisor|}\rfloor$$

Returns none Note

while the divisor (denominator) is signed 16-bits, the quotient is larger than 16-bit and occupies the same memory location. Therefore the divisor must occupy a 32-bit location

For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

28	Cycle count includes the call and return

Table 6.6: Performance Data

6.2.2.6 void FID_i32byi16_m (**int32_t** * p_num_rem, **int32_t** * p_den_quo)

Division: i32/i16 (Modulo or Floored)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = \lfloor\frac{dividend}{divisor}\rfloor$$

$$R = dividend - divisor \times \lfloor \frac{dividend}{divisor} \rfloor$$

Returns none Note

while the divisor (denominator) is signed 16-bits, the quotient is larger than 16-bit and occupies the same memory location. Therefore the divisor must occupy a 32-bit location

For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

28	Cycle count includes the call and return

Table 6.7: Performance Data

6.2.2.7 void FID_i32byi16_t (**int32_t** * p_num_rem, **int32_t** * p_den_quo)

Division: i32/i16 (Truncated)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = trunc(\frac{dividend}{divisor})$$

$$R = dividend - divisor \times trunc(\frac{dividend}{divisor})$$

Returns none Note

while the divisor (denominator) is signed 16-bits, the quotient is larger than 16-bit and occupies the same memory location. Therefore the divisor must occupy a 32-bit location

For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

28	Cycle count includes the call and return

Table 6.8: Performance Data

6.2.2.8 void FID_i32byi32_e (**int32_t** * p_num_rem,

int32_t ∗
p_den_quo)

Division: i32/i32 (Euclidean)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

---

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

---

$$Q = sgn(n)\lfloor\frac{dividend}{|divisor|}\rfloor$$

$$R = dividend - |divisor| \times \lfloor\frac{dividend}{|divisor|}\rfloor$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

24	Cycle count includes the call and return

Table 6.9: Performance Data

6.2.2.9 void FID_i32byi32_m (

int32_t ∗
p_num_rem,
int32_t ∗
p_den_quo)

Division: i32/i32 (Modulo or Floored)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

---

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

---

$$Q = \lfloor\frac{dividend}{divisor}\rfloor$$

$$R = dividend - divisor \times \lfloor\frac{dividend}{divisor}\rfloor$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

24	Cycle count includes the call and return

Table 6.10: Performance Data

6.2.2.10 void FID_i32byi32_t (**int32_t** * p_num_rem, **int32_t** * p_den_quo)

Division: i32/i32 (Truncated)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = trunc(\frac{dividend}{divisor})$$

$$R = dividend - divisor \times trunc(\frac{dividend}{divisor})$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

24	Cycle count includes the call and return

Table 6.11: Performance Data

6.2.2.11 void FID_i32byui32 (**int32_t** * p_num_rem, **int32_t** * p_den_quo)

Division: i32/ui32 (Truncated)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = trunc(\frac{dividend}{divisor})$$

$$R = dividend - divisor \times trunc(\frac{dividend}{divisor})$$

Note

please note that the divisor is unsigned, so you can use the full 32-bit unsigned range for the denominator. When calling the function be sure to cast the pointer to this unsigned denominator as a signed pointer (int32_t *)

For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm. Returns none

24	Cycle count includes the call and return

Table 6.12: Performance Data

6.2.2.12 void FID_i64byi32_e (**int64_t** * p_num_rem, **int64_t** * p_den_quo)

Division: i64/i32 (Euclidean)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = sgn(n)\lfloor\frac{dividend}{|divisor|}\rfloor$$

$$R = dividend - |divisor|\times\lfloor\frac{dividend}{|divisor|}\rfloor$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

36	Cycle count includes the call and return

Table 6.13: Performance Data

6.2.2.13 void FID_i64byi32_m (**int64_t** * p_num_rem, **int64_t** * p_den_quo)

Division: i64/i32 (Modulo or Floored)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

722.7

int64_t ∗
p_den_quo)

Division: i64/i64 (Euclidean)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

---

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

---

$$Q = sgn(n)\lfloor\frac{dividend}{|divisor|}\rfloor$$

$$R = dividend - |divisor| \times \lfloor\frac{dividend}{|divisor|}\rfloor$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

51	Cycle count includes the call and return

Table 6.16: Performance Data

6.2.2.16 void FID_i64byi64_m (

int64_t ∗
p_num_rem,
int64_t ∗
p_den_quo)

Division: i64/i64 (Modulo or Floored)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

---

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

---

$$Q = \lfloor\frac{dividend}{divisor}\rfloor$$

$$R = dividend - divisor \times \lfloor\frac{dividend}{divisor}\rfloor$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

6.2.2.17 void FID_i64byi64_t (

51	Cycle count includes the call and return

Table 6.17: Performance Data

int64_t ∗
p_num_rem,
int64_t ∗
p_den_quo)

Division: i64/i64 (Truncated)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = trunc(\frac{dividend}{divisor})$$

$$R = dividend - divisor \times trunc(\frac{dividend}{divisor})$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

51	Cycle count includes the call and return

Table 6.18: Performance Data

6.2.2.18 void FID_i64byui32 (

int64_t ∗
p_num_rem,
int64_t ∗
p_den_quo)

Division: i64/ui32 (Truncated)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = trunc(\frac{dividend}{divisor})$$

$$R = dividend - divisor \times trunc(\frac{dividend}{divisor})$$

Returns none Note

please note that the divisor is unsigned, so you can use the full 32-bit unsigned range for the denominator. This variable will also hold the 33-bit signed remainder, which must then be sign-extended to 64-bits (standard integer size). The divisor (remainder) variable needs to be of signed 64-bit integer type.

For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

38	Cycle count includes the call and return

Table 6.19: Performance Data

6.2.2.19 void FID_i64byui64 (

int64_t ∗
p_num_rem,
int64_t ∗
p_den_quo)

Division: i64/ui64 (Truncated)

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

$$Q = trunc(\frac{dividend}{divisor})$$

$$R = dividend - divisor \times trunc(\frac{dividend}{divisor})$$

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

52	Cycle count includes the call and return

Table 6.20: Performance Data

6.2.2.20 void FID_ui16byui16 (

uint16_t ∗
p_num_rem,
uint16_t ∗
p_den_quo)

Division: ui16/ui16.

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

24	Cycle count includes the call and return

Table 6.21: Performance Data

6.2.2.21 void FID_ui32byui16 (**uint32_t** * p_num_rem, **uint32_t** * p_den_quo)

Division: ui32/ui16.

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

Returns none Note

while the divisor (denominator) is unsigned 16-bits, the quotient is larger than 16-bit and occupies the same memory location. Therefore the divisor must occupy a 32-bit location

For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

26	Cycle count includes the call and return

Table 6.22: Performance Data

6.2.2.22 void FID_ui32byui32 (**uint32_t** * p_num_rem, **uint32_t** * p_den_quo)

Division: ui32/ui32.

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

23	Cycle count includes the call and return

Table 6.23: Performance Data

6.2.2.23 void FID_ui64byui32 (**uint64_t** * p_num_rem, **uint64_t** * p_den_quo)

Division: ui64/ui32.

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

Returns none Note

Since the memory location that holds the divisor must also hold the quotient it must be 64-bits wide. The same is true of the memory location that holds the 64-bit dividend and subsequently the 32-bit remainder

For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

35	Cycle count includes the call and return

Table 6.24: Performance Data

6.2.2.24 void FID_ui64byui64 (**uint64_t** * p_num_rem, **uint64_t** * p_den_quo)

Division: ui64/ui64.

[1]Parameters `out` *p_num_rem,pointer* to the dividend and subsequently remainder

`out` *p_den_quo,pointer* to the divisor and subsequently quotient

Returns none Note For best performance, make the arguments to the function global; local variables are pushed on to the stack and will cause pipeline stalls when multiple back-to-back stack accesses (read followed by write or vice versa) take place in the algorihtm.

51	Cycle count includes the call and return

Table 6.25: Performance Data

7 Benchmarks

The benchmarks for the single precision library routines were obtained with the following compiler settings:

```
-v28 -mt -ml -g --diag_warning=225 --float_support=fpu32 --tmu_support=tmu0
```

while the benchmarks for the double precision library routines were obtained with the following compiler settings:

```
-v28 -mt -ml -g --diag_warning=225 --float_support=fpu64
```

Table. 7.1 summarizes the performance metrics for the single precision library routines. These numbers were obtained using profiling the code in the examples directory by simulating it on pre silicon platform

Single Precision Routine	Constraints	Cycles
Arithmetic and Trigonometric		
atan2f	[1]	50
atanf	[1]	49
cosf	[1]	38
expf	[1]	61
FS$$DIV	[1]	25
isqrtf	[1]	26
powf	[1]	109
sincosf	[1]	44
sinf	[1]	38
sqrtf	[1]	29

Table 7.1: Benchmark for the Single Precision FPU Library Routines.

Table. 7.2 summarizes the performance metrics for the double precision library routines. These numbers were obtained using profiling the code in the examples directory by simulating it on pre silicon platform

Double Precision Routine	Constraints	Cycles
Arithmetic and Trigonometric		
atan2		81
atan		78
cos		63
Continued on next page		

[1] Includes call and return instructions.

Table 7.2 – continued from previous page

Double Precision Routine	Constraints	Cycles
FD$$DIV		51
isqrt		59
sincos		73
sin		63
sqrt		63
Fast Integer Division		
FID_ui32byui32		23
FID_i32byi32_t		24
FID_i32byi32_m		24
FID_i32byi32_e		24
FID_i32byui32		24
FID_ui64byui32		35
FID_i64byi32_t		36
FID_i64byi32_m		36
FID_i64byi32_e		36
FID_i64byui32		38
FID_ui64byui64		51
FID_i64byi64_t		51
FID_i64byi64_m		51
FID_i64byi64_e		51
FID_i64byui64		52
FID_ui16byui16		24
FID_i16byi16_t		26
FID_i16byi16_m		26
FID_i16byi16_e		26
FID_ui32byui16		26
FID_i32byi16_t		28
FID_i32byi16_m		28
FID_i32byi16_e		28
FID_f64byf64		36

Table 7.2: Benchmark for the Double Precision FPU Library Routines.

Table. 7.3 summarizes the performance metrics for the single precision and double precision critical routines used in their corresponding example projects. These numbers were obtained using profiling the code in the examples directory by executing on F2838x hardware.

Executable file	**Function name**	**Cycles**
atan_f32.out	atanf	39
atan_f64.out	atan	70
atan2_f32.out	atan2f	41
atan2_f64.out	atan2	72
cordic_cos_f64.out	CORDIC_F64_cos	2750
cordic_sin_f64.out	CORDIC_F64_sin	2848
cos_f32.out	cosf	29
cos_f64.out	cos	54
div_f32.out	__c28xabi_divf	222
div_f64.out	__c28xabi_div	42
exp_f32.out	expf	52
FID_f64byf64.out	FID_f64byf64	31
FID_i16byi16.out	FID_i16byi16_t	17
FID_i32byi16.out	FID_i32byi16_t	19
FID_i32byi32.out	FID_i32byi32_t	14
FID_i32byui32.out	FID_i32byui32	14
FID_i64byi32.out	FID_i64byi32_t	26
FID_i64byi64.out	FID_i64byi64_t	42
FID_i64byui32.out	FID_i64byui32	29
FID_i64byui64.out	FID_i64byui64	42
FID_ui16byui16.out	FID_ui16byui16	15
FID_ui32byui16.out	FID_ui32byui16	17
FID_ui32byui32.out	FID_ui32byui32	13
FID_ui64byui32.out	FID_ui64byui32	26
FID_ui64byui64.out	FID_ui64byui64	42
isqrt_f32.out	isqrtf	17
isqrt_f64.out	isqrt	50
pow_f32.out	powf	59
sin_f32.out	sinf	29
sin_f64.out	sin	54
sincos_f32.out	sincosf	34
sincos_f64.out	sincos	63
sqrt_f32.out	sqrtf	20
sqrt_f64.out	sqrt	54

Table 7.3: Benchmark for the single and double precision FPU fastRTS Library Routines used in their corresponding example projects

8 Revision History

V2.04.00.00: Minor Update
Added support for F28003x based control cards.
V2.04.00.00: Minor Update
Migrated to compiler 20.2.1 and CCS 10.x.
V2.03.00.00: Minor Update
Added support for F28002x based control cards.
V2.01.00.00: Major Update

- All the libraries and examples have been migrated from COFF to EABI configuration.
- The examples are provided with RAM_ROMTABLES configuration to make use prestored tables in ROM while computation.
- FLASH configuation is added to the examples.
- Added double precision version of existing algorithms as well as new division algorithms
  - **Arithmetic and Trigonometric**
    - atan2_f64.asm
    - atan_f64.asm
    - cos_f64.asm
    - div_f64.asm
    - isqrt_f64.asm
    - sincos_f64.asm
    - sin_f64.asm
    - sqrt_f64.asm
  - **Fast Integer Divide**
    - fid_f64byf64.asm
    - fid_i16byi16.asm
    - fid_i32byi16.asm
    - fid_i32byi32.asm
    - fid_i32byui32.asm
    - fid_i64byi32.asm
    - fid_i64byi64.asm
    - fid_i64byui32.asm
    - fid_i64byui64.asm
    - fid_ui16byui16.asm
    - fid_ui32byui16.asm
    - fid_ui32byui32.asm
    - fid_ui64byui32.asm
    - fid_ui64byui64.asm
- The comments, examples and folder structure of the single precision library were modified and the following new algorithms added,
  - exp_f32.asm
  - pow_f32.asm

V1.00: Moderate Update
No changes were made to the library itself. The following was done to incorporate the 1.00

release into the controlSUITE structure:

- Updated the directory structure to fit into controlSUITE.
- Added a project to build the library using CCS 4.
- Added an example project using a CCS 4 based project
- Updated this document for CCS 4 and controlSUITE information.

Changes from Beta1 to V1.00

- Removed the version name from the library name. This makes updating to a new library easier.
- Added sincos() function
- Sin and Cos:
  - Corrected the constant value of 0.166 to 0.166667
  - Changed the truncated 2*pi/512 value to a rounded value of 2*pi/512. Previously this value was truncated.
  - In Beta 1, the int (Radian * 512/(2*pi)) calculation was done using float to 16-bit int. Changed this to 32-bit int to accommodate a larger range of input values.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265