


Fast Run Time Support Library

USER'S GUIDE



Copyright

Copyright © 2018 Texas Instruments Incorporated. All rights reserved. Other names and brands may be claimed as the property of others.

 Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

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



Revision Information

This is version V2.01.00.00 of this document, last updated on May 27, 2019.

Table of Contents

Copyright	2
Revision Information	2
1 Introduction	4
2 Other Resources	5
3 Library Structure	6
3.1 Header Files	7
3.2 Source Files	8
4 Using the FPU Library	9
4.1 Library Build Configurations	9
4.2 Integrating the Library into your Project	12
4.3 Confirming Which Library is Used	16
5 Application Programming Interface (FPU32)	18
5.1 Introduction to the Single Precision Fast RTS Library API	18
5.2 Fast RTS (FPU32) Library Definitions and Types	19
6 Application Programming Interface (FPU64)	26
6.1 Introduction to the Double Precision Fast RTS Library API	26
6.2 Fast RTS Definitions and Types	27
6.3 Fast Integer Division Definitions and Types	34
7 Benchmarks	48
8 Revision History	51
IMPORTANT NOTICE	53

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 Trigonometric 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*.

2 Other Resources

The user can refer to the F2838x device TRM for detailed description.

Also check out the TI Delfino page: <http://www.ti.com/delfino>

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

Building the FASTRTS libraries and examples require **Codegen Tools v18.12.1.LTS** or later.

3 Library Structure

Header Files	7
Source Files	8

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.

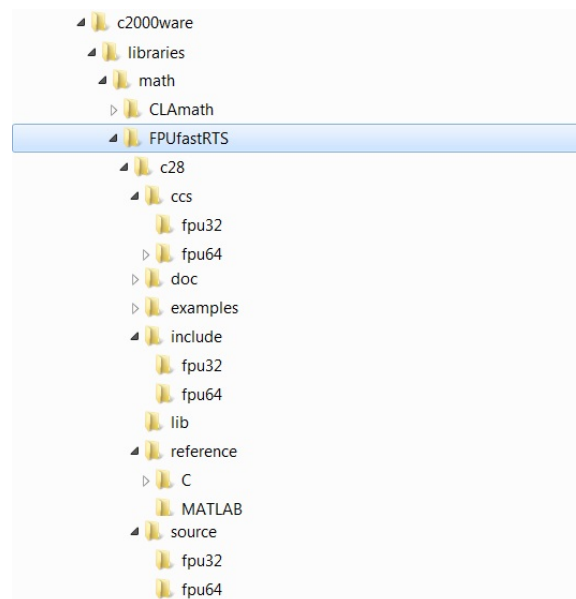


Figure 3.1: Directory Structure of the FASTRTS Library

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 device using the CCS 9.0.1 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

3.1 Header Files

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 superseded by this new file, but it may be used without it.

3.2 Source Files

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

Library Build Configurations	9
Integrating the Library into your Project	12
Confirming Which Library is Used	16

The source code and project(s) for the libraries are provided. The user may import the library project(s) into CCSv9 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

4.1 Library Build Configurations

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. Run-

ning 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

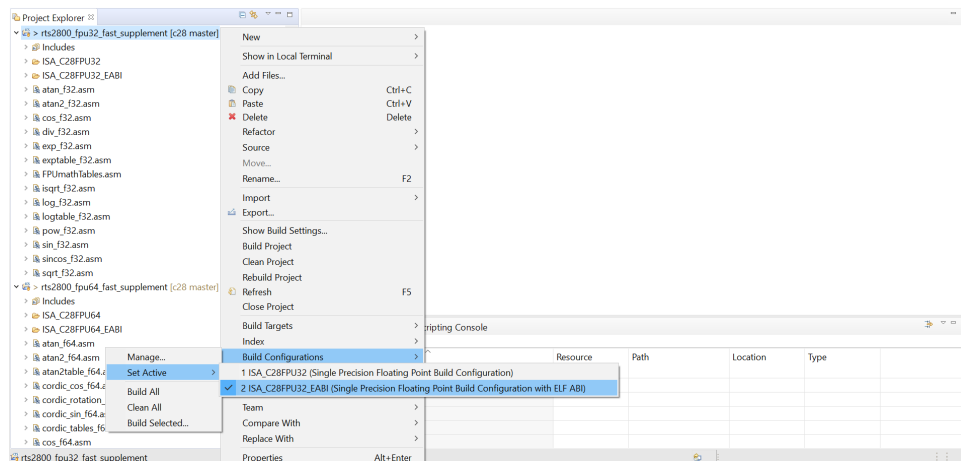


Figure 4.2: Single Precision Library Build Configuration

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

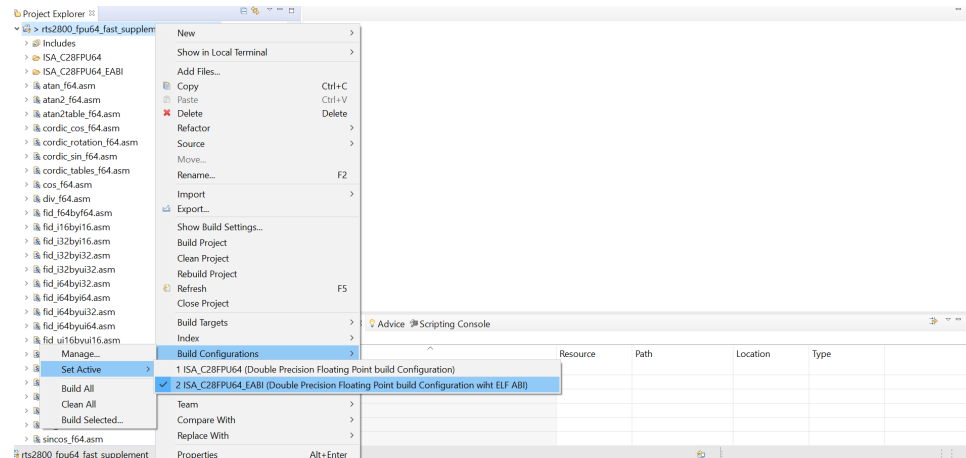


Figure 4.3: Double Precision Library Build Configuration

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 USER'S GUIDE WWW.TI.COM/LIT/SPRU514

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.

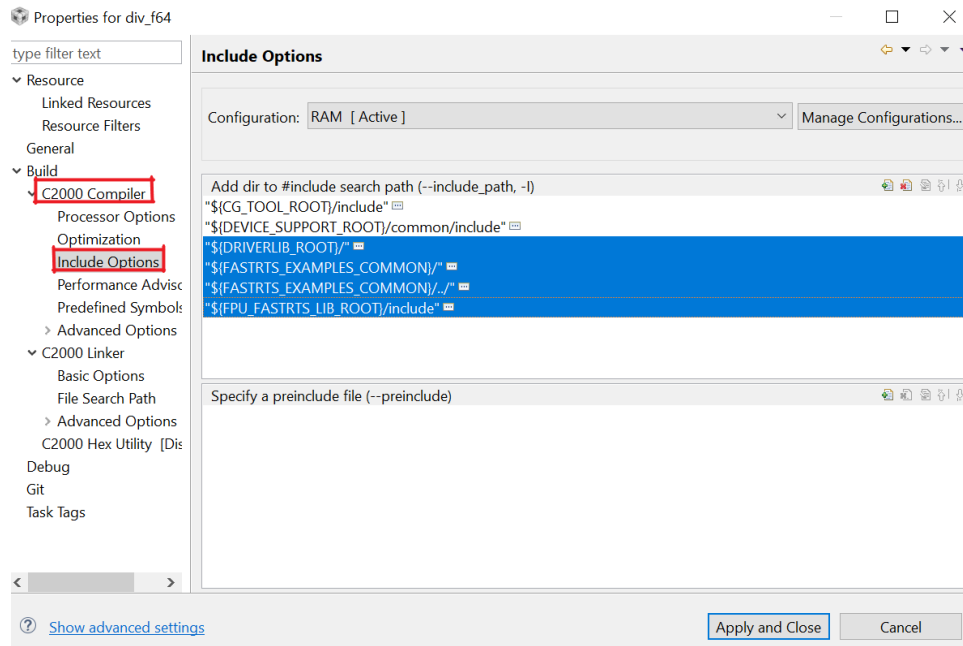


Figure 4.5: Adding the Library Header Path to the Include Options

- For the single precision 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).

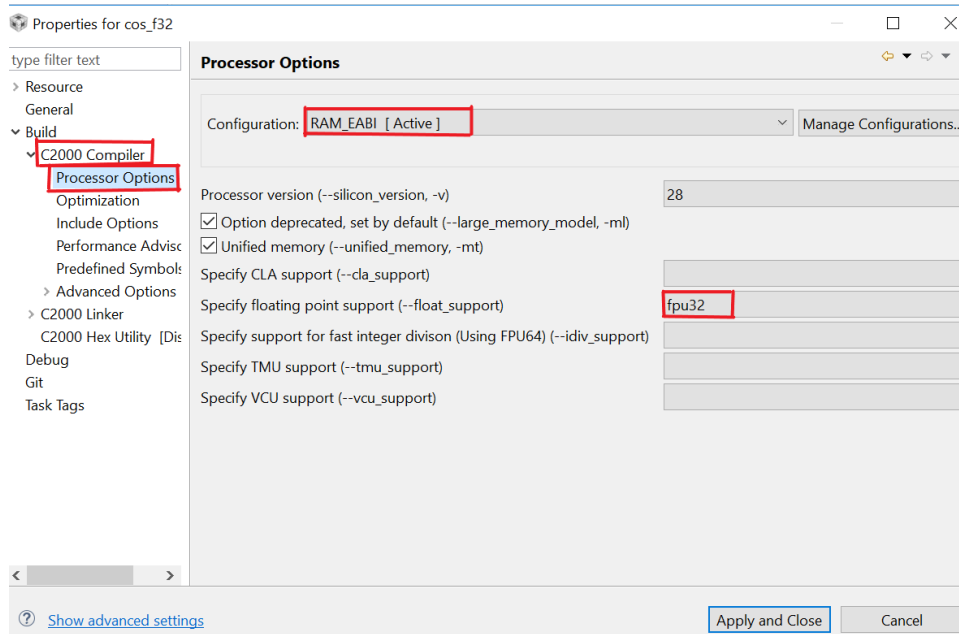


Figure 4.6: Turning on FPU32 support

- For the double precision library set the `--float_support` option to `fpu64` in the **Runtime Model Options** as shown in Figure 4.7.

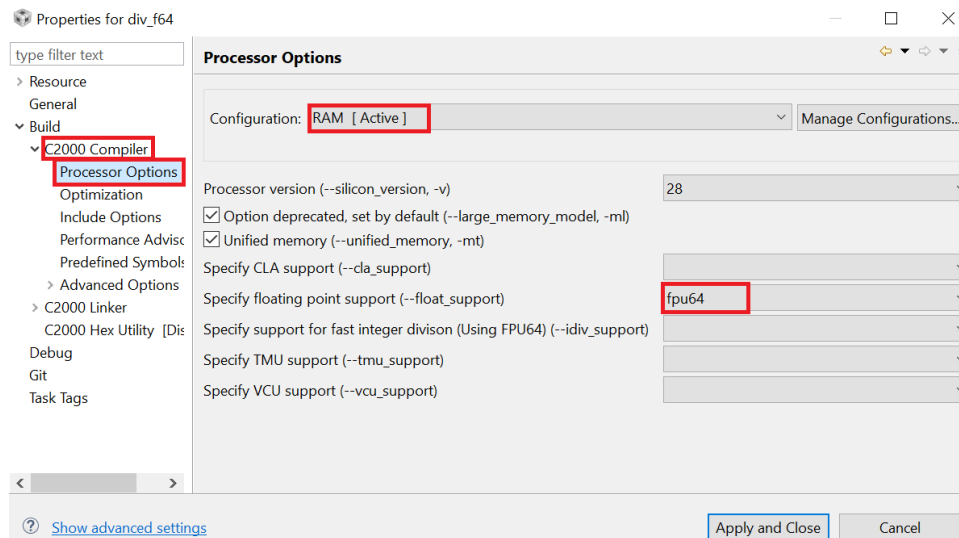


Figure 4.7: Turning on FPU64 support

- 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

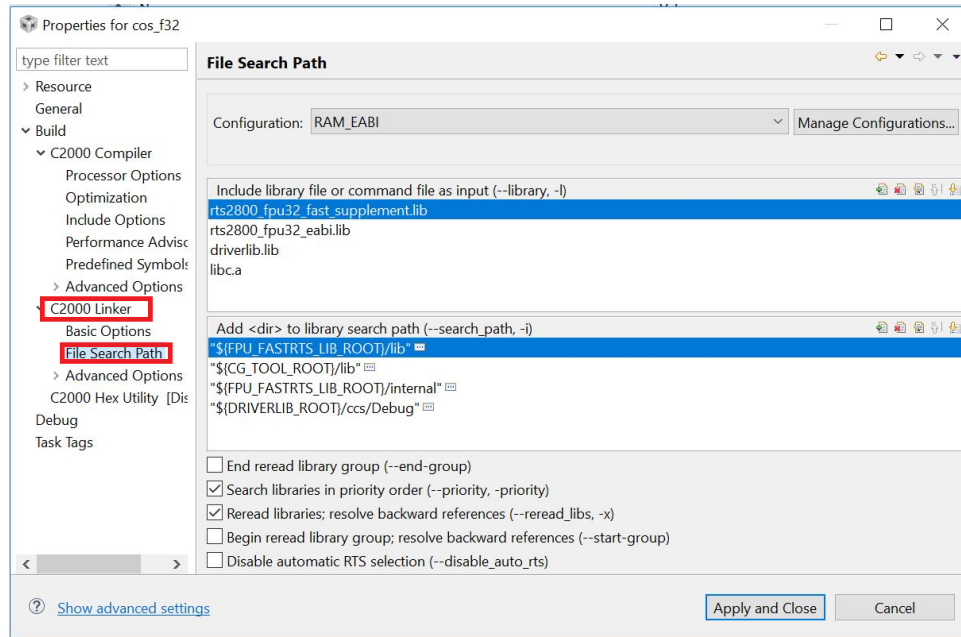


Figure 4.8: Adding the library and location to the file search path

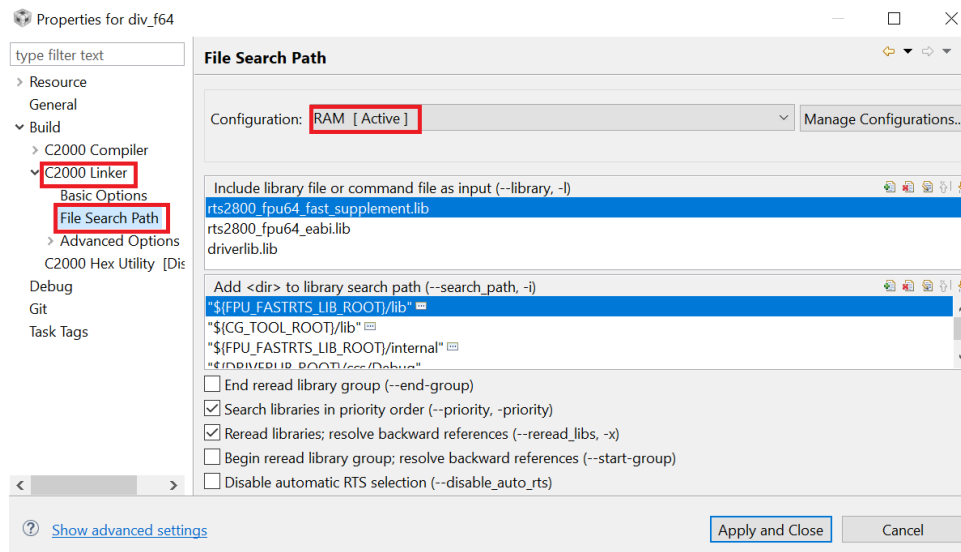


Figure 4.9: Adding the library and location to the file search path

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.

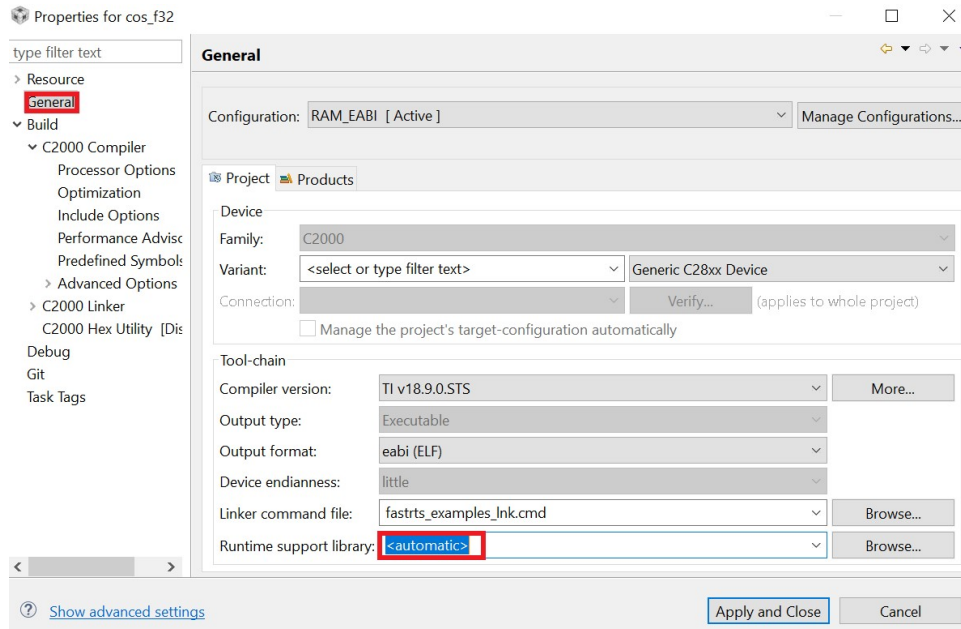


Figure 4.10: Selecting the Runtime Support Library

4.3 Confirming Which Library is Used

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 4.11). Notice the atan function is pulled in from the fastRTS library.

```
.text.2 0 00008800 00000000
00008800 00000002 driverlib.lib : sysctl.obj (.text:_SysCtl_setClock)
00008802 00000000 rts2800_fpu32.lib : fpu32.obj (.text)
00008806 00000007 driverlib.lib : sysctl.obj (.text:_SysCtl_isPLLValid)
00008809 0000000f fastrts_examples_main.obj (.text)
0000880a 00000009 rts2800_fpu32.lib : setbuf.obj (.text)
00008811 00000073 rts2800_fpu32.lib : fflush.obj (.text)
00008814 00000067 : _io_perm.obj (.text)
00008816 0000005f : fclose.obj (.text)
0000881a 0000005e driverlib.lib : sysctl.obj (.text:_DCC_setCounterSeeds)
0000881c 00000056 : sysctl.obj (.text:_SysCtl_selectOscSource)
0000881e 00000056 rts2800_fpu32.lib : boot28.obj (.text)
00008820 0000004f fastrts_atan.obj (.text)
00008823 0000004a rts2800_fpu32_fast_supplement.lib : atan_f32.obj (.text)
00008826 00000033 rts2800_fpu32.lib : fseek.obj (.text)
00008829 0000002a fastrts_examples_setup.obj (.text)
0000882b 00000029 rts2800_fpu32.lib : exit.obj (.text)
0000882d 00000025 driverlib.lib : sysctl.obj (.text:_DCC_enableSingleShotMode)
0000882f 00000024 : sysctl.obj (.text:_SysCtl_selectXTAL)
00008831 00000024 rts2800_fpu32.lib : copy_tbl.obj (.text)
00008834 00000022 driverlib.lib : sysctl.obj (.text:_SysCtl_pollXCounter)
00008837 0000001f driverlib.lib : sysctl.obj (.text:_DCC_setCounterClockSource)
00008839 0000001e rts2800_fpu32.lib : ll_cmp28.obj (.text)
0000883b 0000001e : memcpy.obj (.text)
```

Figure 4.11: 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.


```

.text.2 0 00008000 00000000
00008000 000000cb rts2800_fpu32.lib : s_atanf.obj (.text)
0000800b 00000062 driverlib.lib : sysctl.obj (.text:_sysctl_setClock)
0000801d 00000000 rts2800_fpu32.lib : fpu32.obj (.text)
0000802d 000000a7 driverlib.lib : sysctl.obj (.text:_sysctl_isPLIVValid)
0000804d 0000009f fastrts_examples_main.obj (.text)
00008073 00000089 rts2800_fpu32.lib : setvbuf.obj (.text)
0000808c 00000088 : fs_div28.obj (.text)
00008084 00000073 : fflush.obj (.text)
000080f7 00000067 : _io_perm.obj (.text)
0000805e 0000005f : fclose.obj (.text)
000080bd 0000005e driverlib.lib : sysctl.obj (.text:_DCC_setCounterSeeds)
0000801b 00000056 : sysctl.obj (.text:_sysctl_selectOscSource)
00008071 00000056 rts2800_fpu32.lib : boot28.obj (.text)
000080c7 0000004f fastrts_atan.obj (.text)
00008016 0000001d rts2800_fpu32.lib : fseek.obj (.text)
00008053 0000002a fastrts_examples_setup.obj (.text)
0000807d 00000029 rts2800_fpu32.lib : exit.obj (.text)
000080a6 00000025 driverlib.lib : sysctl.obj (.text:_DCC_enableSingleShotMode)
000080cb 00000024 : sysctl.obj (.text:_sysctl_selectXTAL)
000080ef 00000010 : sysctl.obj (.text:_DCC_isBaseValid)
000080ff 00000001 rts2800_fpu32.lib : startup.obj (.text)

```

Figure 4.12: 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)

5.1 Introduction to the Single Precision Fast RTS Library API

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 v18.12.1.LTS** 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 F2838x device.

5.2 Fast RTS (FPU32) Library Definitions and Types

Modules

- [Fast RTS \(FPU32\) Library Definitions and Types](#)

Functions

- float32_t FS [\\$\\$DIV](#) (float32_t X, float32_t Y)
- 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)

5.2.1 Function Documentation

5.2.1.1 \$\$DIV

Single-Precision Floating-Point Division.

Prototype:

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

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

Parameters:

- ← **X** single precision floating point numerator
- ← **Y** single precision floating point denominator

Returns:

the quotient

Attention:

1. 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.
2. These are the special cases for division

Case	Result	Flags
(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:

1. This is a standard C math function and requires "math.h" to be included
2. 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.

Cycles	Comment
25	Cycle count includes the call and return

Table 5.3: Performance Data

5.2.1.2 atan2f

Single-Precision Floating-Point ATAN2 (radians).

Prototype:

```
float32_t
atan2f(float32_t Y,
       float32_t X)
```

Parameters:

- ← **Y** first single precision floating point argument
- ← **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:

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

2. 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.

Cycles	Comment
50	Cycle count includes the call and return

Table 5.4: Performance Data

5.2.1.3 float32_t atanf (float32_t X)

Single-Precision Floating-Point ATAN (radians).

Parameters:

← **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:

1. This is a standard C math function and requires "math.h" to be included
2. 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.

Cycles	Comment
49	Cycle count includes the call and return

Table 5.5: Performance Data

5.2.1.4 float32_t cosf (float32_t X)

Single-Precision Floating-Point Cosine (radians).

Parameters:

← **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:

1. This is a standard C math function and requires "math.h" to be included
2. 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.

Cycles	Comment
38	Cycle count includes the call and return

Table 5.6: Performance Data

5.2.1.5 float32_t expf (float32_t X)

Single-Precision Floating-Point Exponent.

Parameters:

← **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:

1. This is a standard C math function and requires "math.h" to be included
2. the input domain is limited to $+\log(\text{FLT_MAX})$ ($< +89$)
3. 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.

Cycles	Comment
61	Cycle count includes the call and return

Table 5.7: Performance Data

5.2.1.6 float32_t isqrtf (float32_t X)

Single-Precision Floating-Point 1.0/Square Root.

Parameters:

← **X** single precision floating point argument

Returns:

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

Attention:

1. isqrt(FLT_MAX) and isqrt(FLT_MIN) set the LUF flag.
2. isqrt(-FLT_MIN) will set both the LUF and LVF flags.
3. isqrt(0.0) sets the LVF flag.
4. If X is negative, isqrt(X) will set LVF and return 0.0.

Note:

1. This function is not included in the standard RTS library. It is typically computed as $1.0L/\text{sqrt}(X)$. To use this function you must modify your code to instead call isqrt(X).
2. When migrating from an IQmath project, you can modify the IQmath header file to use isqrt(X) when configured for FLOAT_MATH.
3. This is not a standard C Math function; it requires the "C28x_FPU_FastRTS.h" header instead of "math.h"
4. 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.

Cycles	Comment
26	Cycle count includes the call and return

Table 5.8: Performance Data

5.2.1.7 float32_t logf (float32_t X)

Single-Precision Floating-Point Natural Logarithm.

Parameters:

← **X** single precision floating point argument

Returns:

the natural logarithm of a floating-point argument X

Note:

1. This is a standard C math function and requires "math.h" to be included
2. the input must be greater than or equal to 1
3. 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.

Cycles	Comment
51	Cycle count includes the call and return

Table 5.9: Performance Data

5.2.1.8 float32_t powf (float32_t X, float32_t Y)

Single-Precision Floating-Point Power Function.

Parameters:

← **X** single precision floating point base argument

← **Y** single precision floating point exponent argument

Returns:

X^Y

Note:

1. This is a standard C math function and requires "math.h" to be included
2. the input X must be greater than or equal to 1
3. 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.

Cycles	Comment
109	Cycle count includes the call and return

Table 5.10: Performance Data

5.2.1.9 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.

Parameters:

- ← ***X*** single precision floating point argument
- ***PtrSin*** pointer to the sine of the argument
- ***PtrCin*** pointer to the cosine of the argument

Returns:

none

Note:

1. This is not a standard C Math function; it requires the "C28x_FPU_FastRTS.h" header instead of "math.h"
2. 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.

Cycles	Comment
44	Cycle count includes the call and return

Table 5.11: Performance Data

5.2.1.10 float32_t sinf (float32_t *X*)

Single-Precision Floating-Point Sine (radians).

Parameters:

- ← ***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:

1. This is a standard C math function and requires "math.h" to be included
2. 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.

Cycles	Comment
38	Cycle count includes the call and return

Table 5.12: Performance Data

5.2.1.11 float32_t sqrtf (float32_t *X*)

Single-Precision Floating-Point Square Root.

Parameters:

← **X** single precision floating point argument

Returns:

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

Note:

1. This is a standard C math function and requires "math.h" to be included
2. 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:

1. sqrt(FLT_MAX) and sqrt(FLT_MIN) set the LUF flag.
2. sqrt(-FLT_MIN) will set both the LUF and LVF flags.
3. sqrt(0.0) sets the LVF flag.
4. If X is negative, sqrt(X) will set LVF and return 0.0.
5. 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.

Cycles	Comment
29	Cycle count includes the call and return

Table 5.13: Performance Data

6 Application Programming Interface (FPU64)

6.1 Introduction to the Double Precision Fast RTS Library API

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

6.2 Fast RTS Definitions and Types

Modules

- [Fast RTS Definitions and Types](#)

Data Structures

- [float32u_t](#)
- [float64u_t](#)

Defines

- [LIBRARY_VERSION](#)
- [ZERO_BY_ZERO_EQ_INF](#)
- [ZERO_BY_ZERO_EQ_ZERO](#)

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)

6.2.1 Data Structure Documentation

6.2.1.1 float32u_t

Definition:

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

Members:

ui32 Unsigned long representation.
i32 Signed long representation.

f32 Single precision (32-bit) representation.

Description:

32-bit Double Precision Float The union of a double precision value, an unsigned long and a signed long allows for manipulation of the hex representation of the floating point value as well as signed and unsigned arithmetic to determine error metrics. This data type is only defined if the compiler option `-float_support` is set to `fpu32`

6.2.1.2 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 representation.

f64 Double precision (64-bit) representation.

Description:

64-bit Double Precision Float The union of a double precision value, an unsigned long long and a signed long long allows for manipulation of the hex representation of the floating point value as well as signed and unsigned arithmetic to determine error metrics. This data type is only defined if the compiler option `-float_support` is set to `fpu64`

6.2.2 Define Documentation

6.2.2.1 LIBRARY_VERSION

Definition:

```
#define LIBRARY_VERSION
```

Description:

FAST RTS Library Version.

6.2.2.2 ZERO_BY_ZERO_EQ_INF

Definition:

```
#define ZERO_BY_ZERO_EQ_INF
```

Description:

Determines if 0/0 results in a result of infinity.

6.2.2.3 ZERO_BY_ZERO_EQ_ZERO

Definition:

```
#define ZERO_BY_ZERO_EQ_ZERO
```

Description:

Determines if 0/0 results in a result of zero.

6.2.3 Typedef Documentation

6.2.3.1 v_pfn_v

a function pointer with a void pointer argument returning nothing

Definition:

```
#define void(*)

*0em          group__FASTRTS__F64_g6caba9ec8db12bb66408df233448d07c
               v_pfn_v
               (void
               *
               )
```

6.2.4 Function Documentation

6.2.4.1 `float64u_t __c28xabi_div (float64u_t X, float64u_t Y)`

Single-Precision Floating-Point Division.

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

Parameters:

- ← **X** double precision floating point numerator
- ← **Y** double precision floating point denominator

Returns:

the quotient

Attention:

1. 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.

2. These are the special cases for division

Case	Result	Flags
(0.0/0.0)	= +infinity	
(+DBL_MAX/+DBL_MAX)	= 0.0	LUF = 1
(-DBL_MAX/+DBL_MAX)	= -0.0	LUF = 1
(+DBL_MAX/-DBL_MAX)	= 0.0	LUF = 1
(-DBL_MAX/-DBL_MAX)	= -0.0	LUF = 1
(+DBL_MIN/+DBL_MAX)	= 0.0	LUF = 1
(-DBL_MIN/+DBL_MAX)	= -0.0	LUF = 1
(+DBL_MIN/-DBL_MAX)	= 0.0	LUF = 1
(-DBL_MIN/-DBL_MAX)	= -0.0	LUF = 1

Table 6.2: Special Cases for Division

Note:

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

Cycles	Comment
51	Cycle count includes the call and return

Table 6.3: Performance Data

6.2.4.2 `float64u_t atan (float64u_t X)`

Single-Precision Floating-Point ATAN (radians).

Parameters:

← **X** double 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

Cycles	Comment
78	Cycle count includes the call and return

Table 6.4: Performance Data

6.2.4.3 `float64u_t atan2 (float64u_t Y, float64u_t X)`

Single-Precision Floating-Point ATAN2 (radians).

Parameters:

← **Y** first double precision floating point argument

← **X** second double 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

Cycles	Comment
81	Cycle count includes the call and return

Table 6.5: Performance Data

6.2.4.4 `float64u_t cos (float64u_t X)`

Single-Precision Floating-Point Cosine (radians).

Parameters:

← **X** double 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

Cycles	Comment
63	Cycle count includes the call and return

Table 6.6: Performance Data

6.2.4.5 `float64u_t isqrt (float64u_t X)`

Single-Precision Floating-Point 1.0/Square Root.

Parameters:

← **X** double precision floating point argument

Returns:

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

Attention:

1. `isqrt(FLT_MAX)` and `isqrt(FLT_MIN)` set the LUF flag.
2. `isqrt(-FLT_MIN)` will set both the LUF and LVF flags.
3. `isqrt(0.0)` sets the LVF flag.
4. If X is negative, `isqrt(X)` will set LVF and return 0.0.
5. 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.

Note:

1. This function is not included in the standard RTS library. It is typically computed as $1.0L/\text{sqrt}(X)$. To use this function you must modify your code to instead call `isqrt(X)`.
2. When migrating from an IQmath project, you can modify the IQmath header file to use `isqrt(X)` when configured for `FLOAT_MATH`.
3. This is not a standard C Math function; it requires the "fastrts.h" header instead of "math.h"

Cycles	Comment
59	Cycle count includes the call and return

Table 6.7: Performance Data

6.2.4.6 `float64u_t sin (float64u_t X)`

Single-Precision Floating-Point Sine (radians).

Parameters:

← **X** double 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

Cycles	Comment
63	Cycle count includes the call and return

Table 6.8: Performance Data

6.2.4.7 `void sincos (float64u_t radian, float64u_t * PtrSin, float64u_t * PtrCos)`

Double 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.

Parameters:

- ← **X** double precision floating point argument
- **PtrSin** pointer to the sine of the argument
- **PtrCin** pointer to the cosine of the argument

Returns:

none

Note:

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

Cycles	Comment
73	Cycle count includes the call and return

Table 6.9: Performance Data

6.2.4.8 `float64u_t` sqrt (`float64u_t` X)

Single-Precision Floating-Point Square Root.

Parameters:

← **X** double 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

Attention:

1. sqrt(FLT_MAX) and sqrt(FLT_MIN) set the LUF flag.
2. sqrt(-FLT_MIN) will set both the LUF and LVF flags.
3. sqrt(0.0) sets the LVF flag.
4. If X is negative, sqrt(X) will set LVF and return 0.0.

Cycles	Comment
63	Cycle count includes the call and return

Table 6.10: Performance Data

6.3 Fast Integer Division Definitions and Types

Modules

- [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)

6.3.1 Function Documentation

6.3.1.1 FID_f64byf64

Division: f64/f64.

Prototype:

```
void
FID_f64byf64 (double *p_num,
              double *p_den,
              double *p_quo)
```

Parameters:

- *p_num,pointer* to the dividend
- *p_den,pointer* to the divisor
- *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 algorithm.

Cycles	Comment
36	Cycle count includes the call and return

Table 6.11: Performance Data

6.3.1.2 void FID_i16byi16_e (int16_t * p_num_rem, int16_t * p_den_quo)

Division: i16/i16 (Euclidean).

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder
- *p_den_quo,pointer* to the divisor and subsequently quotient

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

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

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 algorithm.

Cycles	Comment
26	Cycle count includes the call and return

Table 6.12: Performance Data

6.3.1.3 void FID_i16byi16_m (int16_t * p_num_rem, int16_t * p_den_quo)

Division: i16/i16 (Modulo or Floored).

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder

→ ***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 algorithm.

Cycles	Comment
26	Cycle count includes the call and return

Table 6.13: Performance Data

6.3.1.4 void FID_i16byi16_t (int16_t * *p_num_rem*, int16_t * *p_den_quo*)

Division: i16/i16 (Truncated).

Parameters:

→ ***p_num_rem,pointer*** to the dividend and subsequently remainder

→ ***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 algorithm.

Cycles	Comment
26	Cycle count includes the call and return

Table 6.14: Performance Data

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

Division: i16/i16 (Euclidean).

Parameters:

- **p_num_rem,pointer** to the dividend and subsequently remainder
- **p_den_quo,pointer** to the divisor and subsequently quotient

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

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

Returns:

none

Note:

1. 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
2. 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 algorithm.

Cycles	Comment
28	Cycle count includes the call and return

Table 6.15: Performance Data

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

Division: i32/i16 (Modulo or Floored).

Parameters:

- **p_num_rem,pointer** to the dividend and subsequently remainder
- **p_den_quo,pointer** to the divisor and subsequently quotient

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

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

Returns:

none

Note:

1. 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
2. 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 algorithm.

Cycles	Comment
28	Cycle count includes the call and return

Table 6.16: Performance Data

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

Division: i32/i16 (Truncated).

Parameters:

- ***p_num_rem,pointer*** to the dividend and subsequently remainder
- ***p_den_quo,pointer*** to the divisor and subsequently quotient

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

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

Returns:

none

Note:

1. 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
2. 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 algorithm.

Cycles	Comment
28	Cycle count includes the call and return

Table 6.17: Performance Data

6.3.1.8 void FID_i32byi32_e (int32_t * *p_num_rem*, int32_t * *p_den_quo*)

Division: i32/i32 (Euclidean).

Parameters:

- ***p_num_rem,pointer*** to the dividend and subsequently remainder
- ***p_den_quo,pointer*** to the divisor and subsequently quotient

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

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

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 algorithm.

Cycles	Comment
24	Cycle count includes the call and return

Table 6.18: Performance Data

6.3.1.9 void FID_i32byi32_m (int32_t * p_num_rem, int32_t * p_den_quo)

Division: i32/i32 (Modulo or Floored).

Parameters:

- **p_num_rem,pointer** to the dividend and subsequently remainder
- **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 algorithm.

Cycles	Comment
24	Cycle count includes the call and return

Table 6.19: Performance Data

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

Division: i32/i32 (Truncated).

Parameters:

- **p_num_rem,pointer** to the dividend and subsequently remainder
- **p_den_quo,pointer** to the divisor and subsequently quotient

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

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

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 algorithm.

Cycles	Comment
24	Cycle count includes the call and return

Table 6.20: Performance Data

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

Division: i32/ui32 (Truncated).

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder
- *p_den_quo,pointer* to the divisor and subsequently quotient

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

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

Note:

1. 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 *)
2. 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 algorithm.

Returns:

none

Cycles	Comment
24	Cycle count includes the call and return

Table 6.21: Performance Data

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

Division: i64/i32 (Euclidean).

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder
- *p_den_quo,pointer* to the divisor and subsequently quotient

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

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

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 algorithm.

Cycles	Comment
36	Cycle count includes the call and return

Table 6.22: Performance Data

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

Division: i64/i32 (Modulo or Floored).

Parameters:

→ **p_num_rem, pointer** to the dividend and subsequently remainder

→ **p_den_quo, pointer** to the divisor and subsequently quotient

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

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

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 algorithm.

Cycles	Comment
36	Cycle count includes the call and return

Table 6.23: Performance Data

6.3.1.14 void FID_i64byi32_t (int64_t * *p_num_rem*, int64_t * *p_den_quo*)

Division: i64/i32 (Truncated).

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder
- *p_den_quo,pointer* to the divisor and subsequently quotient

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

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

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 algorithm.

Cycles	Comment
36	Cycle count includes the call and return

Table 6.24: Performance Data

6.3.1.15 void FID_i64byi64_e (int64_t * *p_num_rem*, int64_t * *p_den_quo*)

Division: i64/i64 (Euclidean).

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder
- *p_den_quo,pointer* to the divisor and subsequently quotient

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

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

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 algorithm.

Cycles	Comment
51	Cycle count includes the call and return

Table 6.25: Performance Data

6.3.1.16 void FID_i64byi64_m (int64_t * *p_num_rem*, int64_t * *p_den_quo*)

Division: i64/i64 (Modulo or Floored).

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder
- *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 algorithm.

Cycles	Comment
51	Cycle count includes the call and return

Table 6.26: Performance Data

6.3.1.17 void FID_i64byi64_t (int64_t * *p_num_rem*, int64_t * *p_den_quo*)

Division: i64/i64 (Truncated).

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder
- *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 algorithm.

Cycles	Comment
51	Cycle count includes the call and return

Table 6.27: Performance Data

6.3.1.18 void FID_i64byui32 (int64_t * *p_num_rem*, int64_t * *p_den_quo*)

Division: i64/ui32 (Truncated).

Parameters:

→ ***p_num_rem,pointer*** to the dividend and subsequently remainder

→ ***p_den_quo,pointer*** to the divisor and subsequently quotient

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

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

Returns:

none

Note:

1. 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.
2. 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 algorithm.

Cycles	Comment
38	Cycle count includes the call and return

Table 6.28: Performance Data

6.3.1.19 void FID_i64byui64 (int64_t * *p_num_rem*, int64_t * *p_den_quo*)

Division: i64/ui64 (Truncated).

Parameters:

→ ***p_num_rem,pointer*** to the dividend and subsequently remainder

→ ***p_den_quo,pointer*** to the divisor and subsequently quotient

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

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

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 algorithm.

Cycles	Comment
52	Cycle count includes the call and return

Table 6.29: Performance Data

6.3.1.20 void FID_ui16byui16 (uint16_t * *p_num_rem*, uint16_t * *p_den_quo*)

Division: ui16/ui16.

Parameters:

- ***p_num_rem,pointer*** to the dividend and subsequently remainder
- ***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 algorithm.

Cycles	Comment
24	Cycle count includes the call and return

Table 6.30: Performance Data

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

Division: ui32/ui16.

Parameters:

- ***p_num_rem,pointer*** to the dividend and subsequently remainder
- ***p_den_quo,pointer*** to the divisor and subsequently quotient

Returns:

none

Note:

1. 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
2. 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 algorithm.

Cycles	Comment
26	Cycle count includes the call and return

Table 6.31: Performance Data

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

Division: ui32/ui32.

Parameters:

→ *p_num_rem,pointer* to the dividend and subsequently remainder

→ *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 algorithm.

Cycles	Comment
23	Cycle count includes the call and return

Table 6.32: Performance Data

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

Division: ui64/ui32.

Parameters:

→ *p_num_rem,pointer* to the dividend and subsequently remainder

→ *p_den_quo,pointer* to the divisor and subsequently quotient

Returns:

none

Note:

1. 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
2. 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 algorithm.

Cycles	Comment
35	Cycle count includes the call and return

Table 6.33: Performance Data

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

Division: ui64/ui64.

Parameters:

- *p_num_rem,pointer* to the dividend and subsequently remainder
- *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 algorithm.

Cycles	Comment
51	Cycle count includes the call and return

Table 6.34: 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
Continued on next page		

¹Includes call and return instructions.

Table 7.2 – continued from previous page

Double Precision Routine	Constraints	Cycles
atan		78
cos		63
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

V2.01.00.00: 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 pre-stored tables in ROM while computation.
- FLASH configuration 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 $\text{int}(\text{Radian} * 512/(2\pi))$ 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

Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
RF/IF and ZigBee® Solutions	www.ti.com/lprf

Applications

Audio	www.ti.com/audio
Automotive	www.ti.com/automotive
Broadband	www.ti.com/broadband
Digital Control	www.ti.com/digitalcontrol
Medical	www.ti.com/medical
Military	www.ti.com/military
Optical Networking	www.ti.com/opticalnetwork
Security	www.ti.com/security
Telephony	www.ti.com/telephony
Video & Imaging	www.ti.com/video
Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2018, Texas Instruments Incorporated