

# **VCU-II Software Library**


## **USER'S GUIDE**



---

# Copyright

Copyright © 2015 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  
13905 University Boulevard  
Sugar Land, TX 77479  
<http://www.ti.com/c2000>



## Revision Information

This is version V2.10.00.00 of this document, last updated on May 27, 2019.

# Table of Contents

<b>Copyright</b>	<b>2</b>
<b>Revision Information</b>	<b>2</b>
<b>1 Introduction</b>	<b>4</b>
<b>2 Other Resources</b>	<b>5</b>
<b>3 Library Structure</b>	<b>6</b>
<b>4 Using the VCU Library</b>	<b>8</b>
<b>5 Application Programming Interface (VCU0)</b>	<b>12</b>
5.1 VCU0 Type Definitions	12
5.2 Fast Fourier Transform (VCU0)	14
5.3 Cyclic Redundancy Check (VCU0)	21
5.4 Viterbi Decoding (VCU0)	29
<b>6 Application Programming Interface (VCU2)</b>	<b>32</b>
6.1 VCU2 Type Definitions	32
6.2 Fast Fourier Transform (VCU2)	33
6.3 Cyclic Redundancy Check (VCU2)	45
6.4 Viterbi Decoding (VCU2)	59
6.5 Reed Solomon Decoder (VCU2)	64
6.6 De-Interleaver (VCU2)	69
<b>7 Benchmarks</b>	<b>71</b>
<b>8 Revision History</b>	<b>76</b>
<b>IMPORTANT NOTICE</b>	<b>77</b>

# 1 Introduction

The Texas Instruments® C28x Viterbi, Complex Math and CRC Unit Type-2 (VCU2) is a fully programmable block designed to accelerate the performance of communications and digital signal processing algorithms. The software library provides a series of assembly routines, with C wrappers, to carry out many of the DSP algorithms listed below:

1. Complex and Real FFT
2. Viterbi Decoding
3. CRC
4. Reed-Solomon Encoding/Decoding
5. Interleaver/Deinterleaver

**Chapter 2** provides a host of resources on the VCU 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 programming interface, structures and routines available for VCU0

**Chapter 6** describes the programming interface, structures and routines available for VCU2

The performance of each of the library routines is provided in **Chapter 7**.

**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, all examples have been written for the *F2837x* device and tested on an *F2837xcontrolCard* platform. Each example has a script “**SetupDebugEnv.js**” that can be launched from the *Scripting Console* in CCS. These scripts will setup the watch variables for the example. In some examples graphs (.graphProp) are provided ; these can be imported into CCS during debug.

## 2 Other Resources

The user can get answers to F2837x frequently asked questions(FAQ) from the processors wiki page Links to other references such as training videos will be posted here as well. [F2837x Wiki Page](#).

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 VCU library and examples requires **Codegen Tools v6.4.1or later**

### 3 Library Structure

By default, the library and source code is installed into the following directory:

```
C:\ti\c2000\C2000Ware_X_XX_XX_XX\libraries\dsp\VCU
```

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

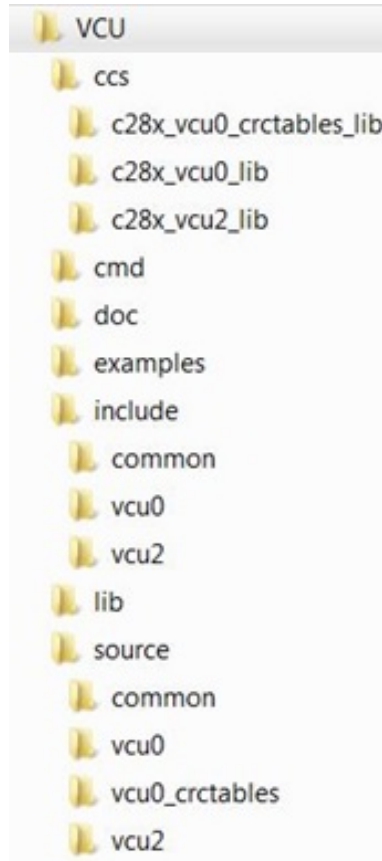


Figure 3.1: Directory Structure of the VCU Library

The user will note (Figure. 3.1) that the source, header and project files for the two VCU types, 0 and 2, are maintained in separate sub-directories titled vcu0 and vcu2. Each VCU type has its own CCS project and .lib output. This allows for legacy compatibility and easy migration of projects that use the older versions of the library.

Folder	Description
<base>	Base install directory. By default this is C:/ti/c2000/C2000Ware_X_XX_XX_XX/libraries/dsp/VCU 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>/cmd	Linker command files used in the examples
<base>/doc	Documentation for the current revision of the library including revision history
<base>/examples	Examples that illustrate the library functions. At the time of writing these examples were built for the F2837x device using the CCS6.0.0.00190 platform
<base>/include	Header files for the VCU library
<base>/lib	Pre-built VCU libraries
<base>/source	Source files for the library.

Table 3.1: VCU Library Directory Structure Description

## 4 Using the VCU Library

The source code and project(s) for the VCU libraries are provided. If you import the library project(s) into CCSv6(or later) you will be able to view and modify the source code for all routines and lookup tables (see Fig. 4.1)

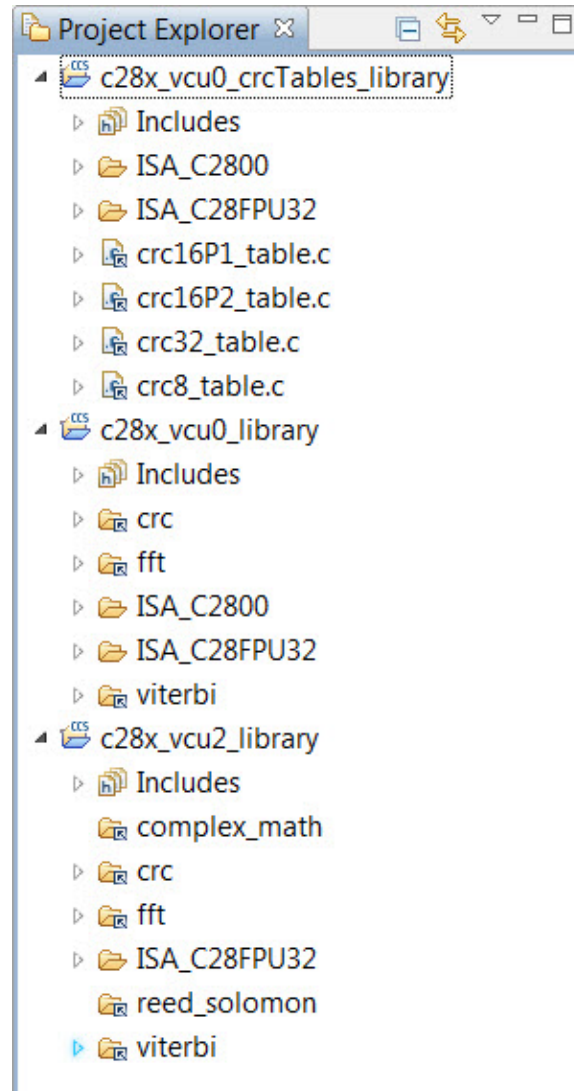


Figure 4.1: VCU Library Project View

The current version of the library(s) has two build configurations (Fig. 4.2) **ISA\_C2800** and **ISA\_C28FPU32**. The difference between the two is the **ISA\_C28FPU32** configuration is built with the **-fpu\_support=fpu32** run-time support option turned on. This allows the VCU library to be integrated into a project which has the **fpu32** option turned on. Each build configuration, when compiled, yields differently titled libraries: **c28x\_vcu<n>\_library.lib** for the ISA C2800 build configuration and **c28x\_vcu<n>\_library\_fpu32.lib** for the floating-point supported build.



**NOTE: ATTEMPTING TO LINK IN THE STANDARD BUILD LIBRARY INTO ANOTHER PROJECT WHICH HAS FPU32 SUPPORT TURNED ON WILL RESULT IN A COMPILER ERROR ABOUT MISMATCHING INSTRUCTION SET ARCHITECTURES, HENCE THE NEED FOR THE ISA\_C28FPU32 BUILD CONFIGURATION**

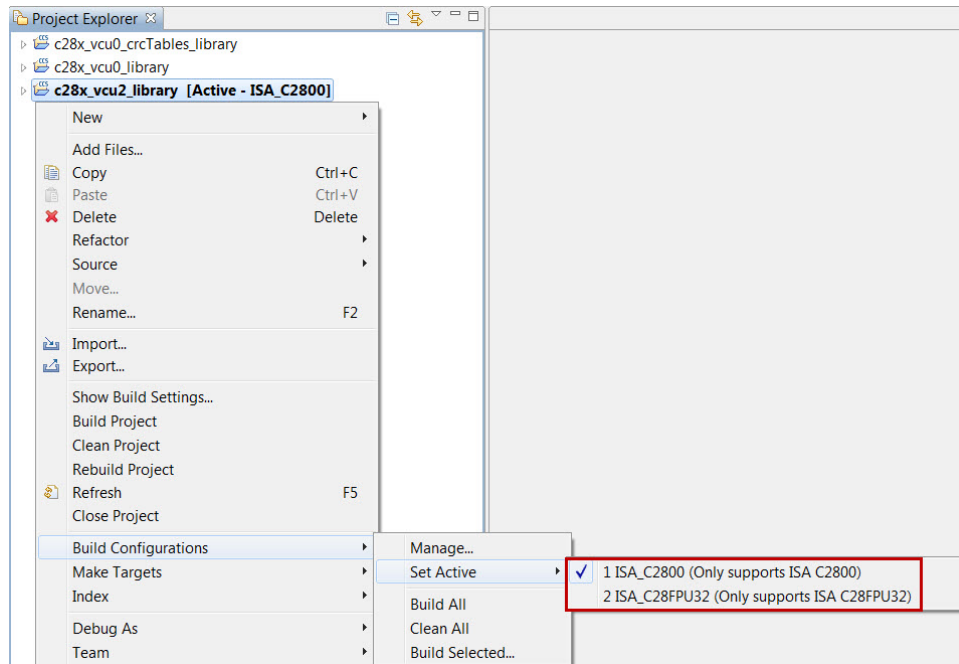


Figure 4.2: Library Build Configurations

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

1. Go to the **Project Properties->Build->Variables(Tab)** and add a new variable (see Fig. 4.3), VCU2\_ROOT\_DIR, and point it to the root directory of the VCU library in **C2000Ware**.

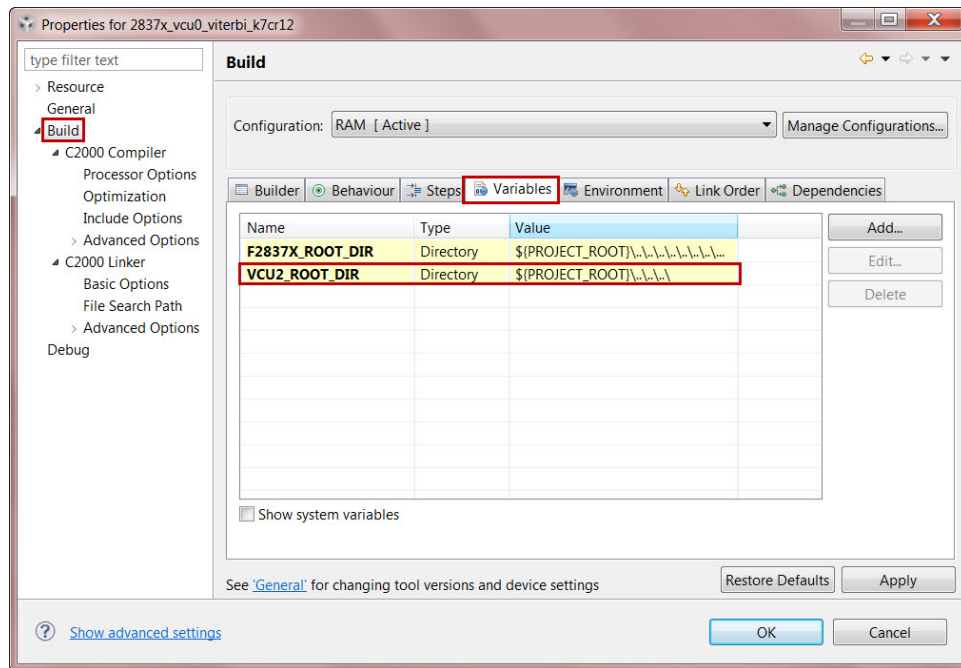


Figure 4.3: Creating a new build variable

Add the new path, **VCU2\_ROOT\_DIR**, to the list of search directories. The paths differ depending on whether you are using the vcu0 or vcu2 libraries. Fig. 4.4 shows the Include options of two projects each using a different vcu library.

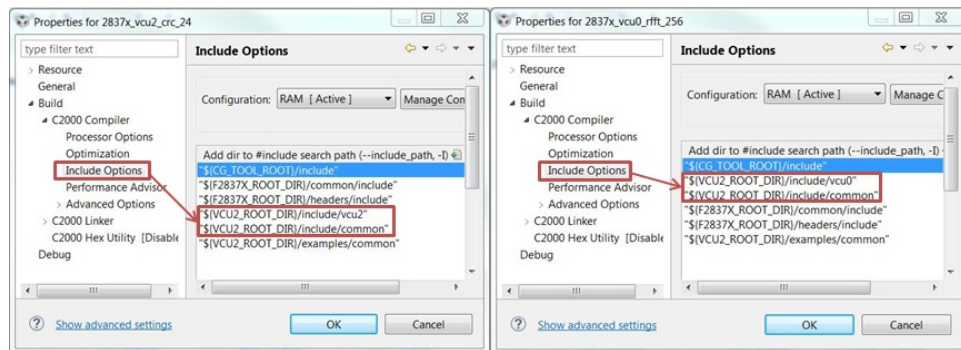


Figure 4.4: Adding the Include Search Path for the Library

2. Enable the **-vcu\_support** option in the **Runtime Model Options** to either **vcu0** or **vcu2** depending on the library used (Fig. 4.5).

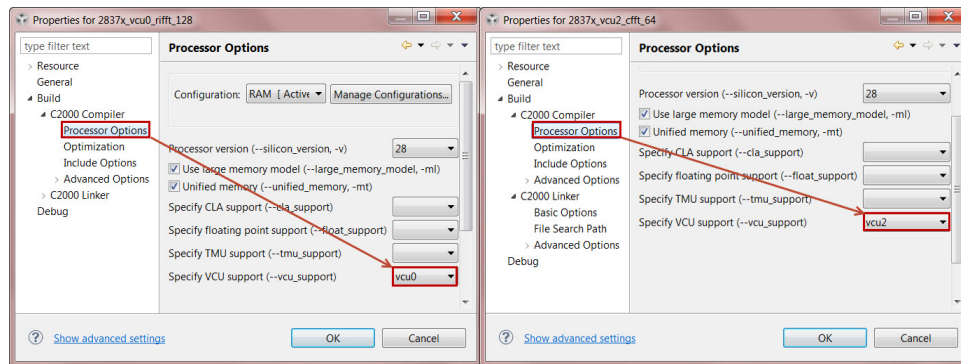


Figure 4.5: Turning on VCU support

3. Add the name of the library and its location to the **File Search Path** as shown in Fig. 4.6. The figure shows build properties for two projects, each using a different vcu library.

**NOTE: IF YOUR PROJECT HAS FPU32 SUPPORT TURNED ON YOU WILL NEED TO ADD THE `c28x_vcu<n>_library_fpu32.lib` LIBRARY IN THE UPPER BOX**

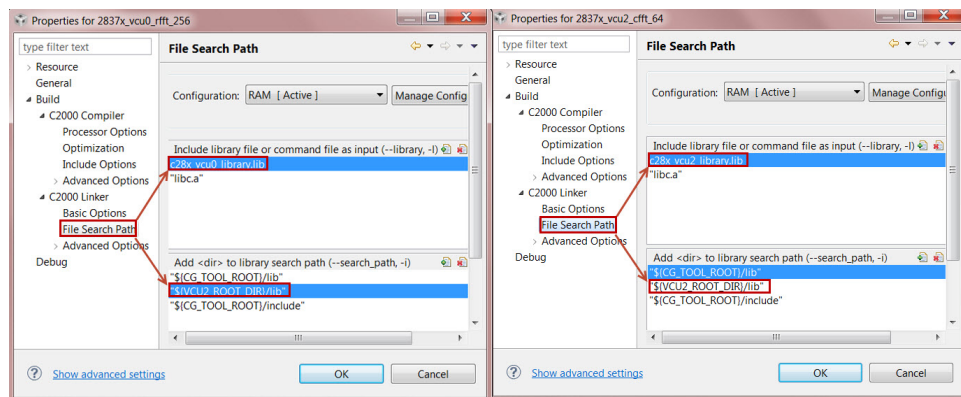


Figure 4.6: Adding the library and location to the file search path

## 5 Application Programming Interface (VCU0)

### 5.1 VCU0 Type Defintions

#### Data Structures

- [cplx16](#)

#### Enumerations

- [CRC\\_parity\\_e](#)

#### 5.1.1 Data Structure Documentation

##### 5.1.1.1 cplx16

**Definition:**

```
typedef struct
{
    SINT16 real;
    SINT16 imag;
}
cplx16
```

**Members:**

***real*** Real Part.  
***imag*** Imaginary Part.

**Description:**

Complex data.

#### 5.1.2 Enumeration Documentation

##### 5.1.2.1 CRC\_parity\_e

**Description:**

Parity enumeration.

The parity is used by the CRC algorithm to determine whether to begin calculations from the low byte (EVEN) or from the high byte (ODD) of the first word (16-bit) in the message.

For example, if your message had 10 bytes and started at the address 0x8000 but the first byte was at the high byte position of the first 16-bit word, the user would call the CRC function with odd parity i.e. `CRC_parity_odd`

Address: HI LO

0x8000 : B0 XX

0x8001 : B2 B1

0x8002 : B4 B3

0x8003 : B6 B5

0x8004 : B8 B7

0x8005 : XX B9

However, if the first byte was at the low byte position of the first 16-bit word, the user would call the CRC function with even parity i.e. `CRC_parity_even`

Address: HI LO

0x8000 : B1 B0

0x8001 : B3 B2

0x8002 : B5 B4

0x8003 : B7 B6

0x8004 : B9 B8

**Enumerators:**

***CRC\_parity\_even*** Even parity, CRC starts at the low byte of the first word.

***CRC\_parity\_odd*** Odd parity, CRC starts at the high byte of the first word.

## 5.2 Fast Fourier Transform (VCU0)

### Data Structures

- [cfft16\\_t](#)

### Defines

- [cfft16\\_128P\\_DEFAULTS](#)
- [cfft16\\_256P\\_DEFAULTS](#)
- [cfft16\\_64P\\_BREV\\_DEFAULTS](#)
- [cfft16\\_64P\\_DEFAULTS](#)
- [rfft16\\_128P\\_DEFAULTS](#)
- [rfft16\\_256P\\_DEFAULTS](#)
- [rfft16\\_512P\\_DEFAULTS](#)
- [riff16\\_128P\\_DEFAULTS](#)
- [riff16\\_256P\\_DEFAULTS](#)
- [riff16\\_64P\\_DEFAULTS](#)

### Functions

- void [cfft16\\_128p\\_calc](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))
- void [cfft16\\_256p\\_calc](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))
- void [cfft16\\_64p\\_calc](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))
- void [cfft16\\_brev](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))
- void [cfft16\\_flip\\_re\\_img](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))
- void [cfft16\\_flip\\_re\\_img\\_conj](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))
- void [cfft16\\_init](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))
- void [cfft16\\_unpack\\_asm](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))
- void [cfft16\\_pack\\_asm](#) ([cfft16\\_t](#) \*[cfft16\\_handle\\_s](#))

### 5.2.1 Data Structure Documentation

#### 5.2.1.1 [cfft16\\_t](#)

**Definition:**

```
typedef struct
{
    int *ipcbptr;
    int *workptr;
    int *tfp_ptr;
    int size;
    int nrstage;
    int step;
```

```
    int *brevptr;  
    void (*init)(void *);  
    void (*calc)(void *);  
}  
cfft16_t
```

**Members:**

***ipcbptr*** input buffer pointer  
***workptr*** work buffer pointer  
***tfptr*** twiddle factor table pointer  
***size*** Number of data points.  
***nrstage*** Number of FFT stages.  
***step*** Twiddle factor table search step.  
***brevptr*** Bit reversal table pointer.  
***init*** Function pointer to initialization routine.  
***calc*** Function pointer to calculation routine.

**Description:**

Complex FFT data structure.

## 5.2.2 Define Documentation

### 5.2.2.1 cfft16\_128P\_DEFAULTS

**Definition:**

```
#define cfft16_128P_DEFAULTS
```

**Description:**

Default values for the complex FFT structure for 128 sample points.

### 5.2.2.2 cfft16\_256P\_DEFAULTS

**Definition:**

```
#define cfft16_256P_DEFAULTS
```

**Description:**

Default values for the complex FFT structure for 256 sample points.

### 5.2.2.3 cfft16\_64P\_BREV\_DEFAULTS

**Definition:**

```
#define cfft16_64P_BREV_DEFAULTS
```

**Description:**

Default values for the complex FFT structure for 64 sample points if using bit reversal lookup table (Deprecated)

#### 5.2.2.4 cfft16\_64P\_DEFAULTS

**Definition:**

```
#define cfft16_64P_DEFAULTS
```

**Description:**

Default values for the complex FFT structure for 64 sample points.

#### 5.2.2.5 rfft16\_128P\_DEFAULTS

**Definition:**

```
#define rfft16_128P_DEFAULTS
```

**Description:**

Default values for the complex FFT structure for 128 real sample points.

#### 5.2.2.6 rfft16\_256P\_DEFAULTS

**Definition:**

```
#define rfft16_256P_DEFAULTS
```

**Description:**

Default values for the complex FFT structure for 256 real sample points.

#### 5.2.2.7 rfft16\_512P\_DEFAULTS

**Definition:**

```
#define rfft16_512P_DEFAULTS
```

**Description:**

Default values for the complex FFT structure for 512 real sample points.

#### 5.2.2.8 rfft16\_128P\_DEFAULTS

**Definition:**

```
#define rfft16_128P_DEFAULTS
```

**Description:**

Default values for the Real Inverse FFT structure for 128 points.

#### 5.2.2.9 rfft16\_256P\_DEFAULTS

**Definition:**

```
#define rfft16_256P_DEFAULTS
```

**Description:**

Default values for the Real Inverse FFT structure for 256 points.



### 5.2.2.10 riff16\_64P\_DEFAULTS

**Definition:**

```
#define riff16_64P_DEFAULTS
```

**Description:**

Default values for the Real Inverse FFT structure for 64 points.

## 5.2.3 Typedef Documentation

### 5.2.3.1 cfft16\_handle\_s

**Definition:**

```
typedef cfft16_t *cfft16_handle_s
```

**Description:**

Handle to structure.

## 5.2.4 Function Documentation

### 5.2.4.1 cfft16\_128p\_calc

Calculate the 128 pt Complex FFT.

**Prototype:**

```
void  
cfft16_128p_calc(cfft16_t *cfft16_handle_s)
```

**Parameters:**

**cfft16\_handle\_s** Handle to the FFT structure

**See also:**

[cfft16\\_brev](#) for memory alignment requirements

### 5.2.4.2 void cfft16\_256p\_calc (cfft16\_t \* cfft16\_handle\_s)

Calculate the 256 pt Complex FFT.

**Parameters:**

**cfft16\_handle\_s** Handle to the FFT structure

**See also:**

[cfft16\\_brev](#) for memory alignment requirements

### 5.2.4.3 void cfft16\_64p\_calc (cfft16\_t \* cfft16\_handle\_s)

Calculate the 64 pt Complex FFT.

**Parameters:**

***cfft16\_handle\_s*** Handle to the FFT structure

#### 5.2.4.4 void cfft16\_brev ([cfft16\\_t](#) \* *cfft16\_handle\_s*)

Bit-Reversed Indexing.

Rearranges the input data in bit-reversed index format. If the number of FFT stages is even, the data is bit-reversed into the work buffer and then copied back to the input buffer. In this respect the bit reversal is considered to be in-place. For an odd number of stages the bit-reversed output is placed in the work buffer (off-place). The FFT (not the bit reversal function) will then transfer the data back to the input buffer pointed to by ipcbptr

**Parameters:**

***cfft16\_handle\_s*** Handle to the FFT structure

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 128 point complex FFT requires an input buffer of size 256 words (16-bit), therefore it must be aligned to a boundary of 256. This can be done by assigning the array to a named section (fftInput) using compiler pragmas (in the example, the input is assigned to .econst and aligned to a boundary of 256 using the .align assembler directive)

```
#pragma DATA_SECTION (CFFT16_128p_in_data, "fftInput")
```

and then either assigning this memory to the start of a RAM block in the linker command file, as is done in the examples, or aligning it to a boundary using the align directive

```
fftInput : > RAMLS4, ALIGN = 256, PAGE = 1
```

#### 5.2.4.5 void cfft16\_flip\_re\_img ([cfft16\\_t](#) \* *cfft16\_handle\_s*)

Flip real and imaginary parts of complex number.

This functions is needed in the computation of real FFTs to ensure that the real part of the complex number always ends up at the high word (16-bit) of a 32 bit address

**Parameters:**

***cfft16\_handle\_s*** Handle to the FFT structure

#### 5.2.4.6 void cfft16\_flip\_re\_img\_conj ([cfft16\\_t](#) \* *cfft16\_handle\_s*)

Flip real and imaginary parts of complex number and conjugate.

This functions is needed in the computation of real IFFTs to ensure that the real part of the complex number always ends up at the high word (16-bit) of a 32 bit address

**Parameters:**

***cfft16\_handle\_s*** Handle to the FFT structure

#### 5.2.4.7 void `cfft16_init` (`cfft16_t` \* `cfft16_handle_s`)

Twiddle Factor Table Initialization.

Initializes the `tfptr` (twiddle factor pointer) to the start of the twiddle factor table in memory

**Parameters:**

**`cfft16_handle_s`** Handle to the FFT structure

#### 5.2.4.8 void `cfft16_unpack_asm` (`cfft16_t` \* `cfft16_handle_s`)

Real FFT Unpack.

When using an N/2 pt complex FFT to compute the N-pt real FFT, the result of the complex FFT must be unpacked to get the real value. Refer to <http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM> for the complete derivation and explanation of the algorithm

**Parameters:**

**`cfft16_handle_s`** Handle to the FFT structure

#### 5.2.4.9 void `cfft16_pack_asm` (`cfft16_t` \* `cfft16_handle_s`)

complex IFFT pack

When calculating the IFFT of a Real FFT, the data must be packed before using the complex IFFT to get the result. Refer to <http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM> for the complete derivation and explanation of the algorithm

**Parameters:**

**`cfft16_handle_s`** Handle to the FFT structure

### 5.2.5 Real Fast Fourier Transform

It is possible to run the Fast Fourier Transform on a sequence of real data using the complex FFT. For a 2N point real sequence, the user would treat the data as N-pt complex (no rearrangement required) and run it through an N point complex FFT. In order to derive the correct spectrum, you would have to “unpack” the output. The derivations can be found here:

<http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM>

Similarly, to run an inverse Real FFT, the user would “pack” the data and either run it through an N-point Inverse Complex FFT or an N-point Forward Complex FFT and then conjugating its complex output. Please see the examples folder for how this is done.

**Note 1** When running an inverse real FFT after the forward real FFT, the user must take care to first switch the **Input** and **Output** pointers in the FFT object before calling the FFT routine again

**Note 2** Because the buffers are switched for the inverse FFT, they must both be aligned to a 2N word boundary.

**Note 3** The **pack**, **unpack**, and **FFT** routines scale down the input data to prevent overflows. Therefore, the output of the real inverse FFT process will be a scaled down version of the original. The user may choose to scale the output of intermediate operations to prevent small values being zeroed out

**See also:**

[cfft16\\_unpack\\_asm](#), [cfft16\\_pack\\_asm](#), [cfft16\\_flip\\_re\\_img](#), [cfft16\\_flip\\_re\\_img\\_conj](#)

## 5.3 Cyclic Redundancy Check (VCU0)

### Defines

- `INIT_CRC16`
- `INIT_CRC32`
- `INIT_CRC8`
- `POLYNOMIAL16_1`
- `POLYNOMIAL16_2`
- `POLYNOMIAL32`
- `POLYNOMIAL8`

### Functions

- `void CRC_reset (void)`
- `void genCRC16P1Table ()`
- `void genCRC16P2Table ()`
- `void genCRC32Table ()`
- `void genCRC8Table ()`
- `uint16 getCRC16P1_cpu (uint16 input_crc16_accum, uint16 *msg, CRC_parity_e parity, uint16 rxLen)`
- `uint16 getCRC16P1_vcu (uint32 input_crc16_accum, uint16 *msg, CRC_parity_e parity, uint16 rxLen)`
- `uint16 getCRC16P2_cpu (uint16 input_crc16_accum, uint16 *msg, CRC_parity_e parity, uint16 rxLen)`
- `uint16 getCRC16P2_vcu (uint32 input_crc16_accum, uint16 *msg, CRC_parity_e parity, uint16 rxLen)`
- `uint32 getCRC32_cpu (uint32 input_crc32_accum, uint16 *msg, CRC_parity_e parity, uint16 rxLen)`
- `uint32 getCRC32_vcu (uint32 input_crc32_accum, uint16 *msg, CRC_parity_e parity, uint16 rxLen)`
- `uint16 getCRC8_cpu (uint16 input_crc8_accum, uint16 *msg, CRC_parity_e parity, uint16 rxLen)`
- `uint16 getCRC8_vcu (uint32 input_crc8_accum, uint16 *msg, CRC_parity_e parity, uint16 rxLen)`

### 5.3.1 Define Documentation

#### 5.3.1.1 INIT\_CRC16

**Definition:**

```
#define INIT_CRC16
```

**Description:**

Initial CRC Register Value.

### 5.3.1.2 INIT\_CRC32

**Definition:**

```
#define INIT_CRC32
```

**Description:**

Initial CRC Register Value.

### 5.3.1.3 INIT\_CRC8

**Definition:**

```
#define INIT_CRC8
```

**Description:**

Initial CRC Register Value.

### 5.3.1.4 POLYNOMIAL16\_1

**Definition:**

```
#define POLYNOMIAL16_1
```

**Description:**

CRC16 802.15.4 Polynomial.

### 5.3.1.5 POLYNOMIAL16\_2

**Definition:**

```
#define POLYNOMIAL16_2
```

**Description:**

CRC16 Alternate Polynomial.

### 5.3.1.6 POLYNOMIAL32

**Definition:**

```
#define POLYNOMIAL32
```

**Description:**

CRC32 PRIME Polynomial.

### 5.3.1.7 POLYNOMIAL8

**Definition:**

```
#define POLYNOMIAL8
```

**Description:**

CRC8 PRIME Polynomial.

## 5.3.2 Function Documentation

### 5.3.2.1 CRC\_reset

Workaround to the silicon issue of first VCU calculation on power up being erroneous.

**Prototype:**

```
void  
CRC_reset(void)
```

**Description:**

Due to the internal power-up state of the VCU module, it is possible that the first CRC result will be incorrect. This condition applies to the first result from each of the eight CRC instructions. This rare condition can only occur after a power-on reset, but will not necessarily occur on every power on. A warm reset will not cause this condition to reappear. The application can reset the internal VCU CRC logic by performing a CRC calculation of a single byte in the initialization routine. This routine only needs to perform one CRC calculation and can use any of the CRC instructions

### 5.3.2.2 void genCRC16P1Table ()

Generate the CRC lookup table using the polynomial 0x8005.

This function generates the CRC16 table for every possible byte, i.e.  $2^8 = 256$  table values, using the CRC16\_802\_15\_4 polynomial 0x8005. It expects a global array, `crc16p1_table`, to be defined in the application code

### 5.3.2.3 void genCRC16P2Table ()

Generate the CRC lookup table using the polynomial 0x1021.

This function generates the CRC16 table for every possible byte, i.e.  $2^8 = 256$  table values, using the CRC16\_ALT polynomial 0x1021. It expects a global array, `crc16p2_table`, to be defined in the application code

### 5.3.2.4 void genCRC32Table ()

Generate the CRC lookup table using the polynomial 0x04c11db7.

This function generates the CRC32 table for every possible byte, i.e.  $2^8 = 256$  table values, using the CRC32\_PRIME polynomial 0x04c11db7. It expects a global array, `crc32_table`, to be defined in the application code

### 5.3.2.5 void genCRC8Table ()

Generate the CRC lookup table using the polynomial 0x7.

This function generates the CRC8 table for every possible byte, i.e.  $2^8 = 256$  table values, using the CRC8\_PRIME polynomial 0x07. It expects a global array, `crc8_table`, to be defined in the application code

### 5.3.2.6 uint16 getCRC16P1\_cpu (uint16 *input\_crc16\_accum*, uint16 \* *msg*, CRC\_parity\_e *parity*, uint16 *rxLen*)

C- function to get the 16-bit CRC.

Calculate the 16-bit CRC of a message buffer by using the lookup table, *crc16p1\_table*, based on the polynomial 0x8005.

**Parameters:**

***input\_crc16\_accum*** The seed value for the CRC, in the event of a multi-part message, the result of the previous *crc16* can be used as the initial value for the current segment *crc16* calculation until the final *crc* is derived.

***msg*** Address of the message buffer

***parity*** Parity of the first message word. The parity determines whether the CRC begins at the low byte (*CRC\_parity\_even*) or at the high byte (*CRC\_parity\_odd*) of the first word

***rxLen*** Length of the message in bytes

**Returns:**

CRC result

### 5.3.2.7 getCRC16P1\_vcu

VCU(ASM)- function to get the 16-bit CRC.

**Prototype:**

```
uint16  
getCRC16P1_vcu (uint32 input_crc16_accum,  
                uint16 *msg,  
                CRC_parity_e parity,  
                uint16 rxLen)
```

**Description:**

Calculate the 16-bit CRC of a message buffer by using the VCU instructions *VCRC16P1H\_1* and *VCRC16P1L\_1*

**Parameters:**

***input\_crc16\_accum*** The seed value for the CRC, in the event of a multi-part message, the result of the previous *crc16* can be used as the initial value for the current segment *crc16* calculation until the final *crc* is derived.

***msg*** Address of the message buffer

***parity*** Parity of the first message word. The parity determines whether the CRC begins at the low byte (*CRC\_parity\_even*) or at the high byte (*CRC\_parity\_odd*) of the first word

***rxLen*** Length of the message in bytes

**Returns:**

CRC result

### 5.3.2.8 getCRC16P2\_cpu

C- function to get the 16-bit CRC.



**Prototype:**

```
uint16  
getCRC16P2_cpu(uint16 input_crc16_accum,  
               uint16 *msg,  
               CRC_parity_e parity,  
               uint16 rxLen)
```

**Description:**

Calculate the 16-bit CRC of a message buffer by using the lookup table, `crc16p2_table`, based on the polynomial 0x1021.

**Parameters:**

**input\_crc16\_accum** The seed value for the CRC, in the event of a multi-part message, the result of the previous crc16 can be used as the initial value for the current segment crc16 calculation until the final crc is derived.

**msg** Address of the message buffer

**parity** Parity of the first message word. The parity determines whether the CRC begins at the low byte (`CRC_parity_even`) or at the high byte (`CRC_parity_odd`) of the first word

**rxLen** Length of the message in bytes

**Returns:**

CRC result

### 5.3.2.9 getCRC16P2\_vcu

VCU(ASM)- function to get the 16-bit CRC.

**Prototype:**

```
uint16  
getCRC16P2_vcu(uint32 input_crc16_accum,  
               uint16 *msg,  
               CRC_parity_e parity,  
               uint16 rxLen)
```

**Description:**

Calculate the 16-bit CRC of a message buffer by using the VCU instructions `VCRC16P2H_1` and `VCRC16P2L_1`

**Parameters:**

**input\_crc16\_accum** The seed value for the CRC, in the event of a multi-part message, the result of the previous crc16 can be used as the initial value for the current segment crc16 calculation until the final crc is derived.

**msg** Address of the message buffer

**parity** Parity of the first message word. The parity determines whether the CRC begins at the low byte (`CRC_parity_even`) or at the high byte (`CRC_parity_odd`) of the first word

**rxLen** Length of the message in bytes

**Returns:**

CRC result

### 5.3.2.10 getCRC32\_cpu

C- function to get the 32-bit CRC.

**Prototype:**

```
uint32
getCRC32_cpu(uint32 input_crc32_accum,
             uint16 *msg,
             CRC_parity_e parity,
             uint16 rxLen)
```

**Description:**

Calculate the 32-bit CRC of a message buffer by using the lookup table, `crc32_table`, based on the polynomial 0x04c11db7.

**Parameters:**

**input\_crc32\_accum** The seed value for the CRC, in the event of a multi-part message, the result of the previous crc32 can be used as the initial value for the current segment crc32 calculation until the final crc is derived.

**msg** Address of the message buffer

**parity** Parity of the first message word. The parity determines whether the CRC begins at the low byte (`CRC_parity_even`) or at the high byte (`CRC_parity_odd`) of the first word

**rxLen** Length of the message in bytes

**Returns:**

CRC result

### 5.3.2.11 getCRC32\_vcu

VCU(ASM)- function to get the 32-bit CRC.

**Prototype:**

```
uint32
getCRC32_vcu(uint32 input_crc32_accum,
             uint16 *msg,
             CRC_parity_e parity,
             uint16 rxLen)
```

**Description:**

Calculate the 32-bit CRC of a message buffer by using the VCU instructions `VCRC32H_1` and `VCRC32L_1`

**Parameters:**

**input\_crc32\_accum** The seed value for the CRC, in the event of a multi-part message, the result of the previous crc32 can be used as the initial value for the current segment crc32 calculation until the final crc is derived.

**msg** Address of the message buffer

**parity** Parity of the first message word. The parity determines whether the CRC begins at the low byte (`CRC_parity_even`) or at the high byte (`CRC_parity_odd`) of the first word

**rxLen** Length of the message in bytes

**Returns:**

CRC result

### 5.3.2.12 getCRC8\_cpu

C- function to get the 8-bit CRC.

**Prototype:**

```
uint16  
getCRC8_cpu(uint16 input_crc8_accum,  
            uint16 *msg,  
            CRC_parity_e parity,  
            uint16 rxLen)
```

**Description:**

Calculate the 8-bit CRC of a message buffer by using the lookup table, `crc8_table`, based on the polynomial 0x7.

**Parameters:**

***input\_crc8\_accum*** The seed value for the CRC, in the event of a multi-part message, the result of the previous crc8 can be used as the initial value for the current segment crc8 calculation until the final crc is derived.

***msg*** Address of the message buffer

***parity*** Parity of the first message word. The parity determines whether the CRC begins at the low byte (`CRC_parity_even`) or at the high byte (`CRC_parity_odd`) of the first word

***rxLen*** Length of the message in bytes

**Returns:**

CRC result

### 5.3.2.13 getCRC8\_vcu

VCU(ASM)- function to get the 8-bit CRC.

**Prototype:**

```
uint16  
getCRC8_vcu(uint32 input_crc8_accum,  
            uint16 *msg,  
            CRC_parity_e parity,  
            uint16 rxLen)
```

**Description:**

Calculate the 8-bit CRC of a message buffer by using the VCU instructions, `VCRC8L_1` and `VCRC8H_1`

**Parameters:**

***input\_crc8\_accum*** The seed value for the CRC, in the event of a multi-part message, the result of the previous crc8 can be used as the initial value for the current segment crc8 calculation until the final crc is derived.

***msg*** Address of the message buffer

***parity*** Parity of the first message word. The parity determines whether the CRC begins at the low byte (`CRC_parity_even`) or at the high byte (`CRC_parity_odd`) of the first word determines whether the CRC begins at the low byte (EVEN) or at the high byte (ODD).

***rxLen*** Length of the message in bytes

**Returns:**

CRC result

## 5.4 Viterbi Decoding (VCU0)

### Enumerations

- [vitMode\\_t](#)

### Functions

- void [cnvDec\\_asm](#) (int nBits, int \*in\_p, int \*out\_p, int flag)
- void [cnvDecInit\\_asm](#) (int nTranBits)
- void [cnvDecMetricRescale\\_asm](#) ()

### Variables

- int32 [VIT\\_gold\\_vt\\_data](#) []
- int16 [VIT\\_in\\_data](#) []
- int16 [VIT\\_quant\\_data](#) []

## 5.4.1 Enumeration Documentation

### 5.4.1.1 vitMode\_t

**Description:**

Viterbi decode mode enumeration.

**Enumerators:**

**CNV\_DEC\_MODE\_DEC\_ALL** Decodes all output bits.

**CNV\_DEC\_MODE\_OVLP\_INIT** Use window overlap method, only metrics and transitions update

**CNV\_DEC\_MODE\_OVLP\_DEC** Use window overlap method, update transitions/metrics/trace through current & previous blocks, decode previous block only

**CNV\_DEC\_MODE\_OVLP\_LAST** last block in overlap

## 5.4.2 Function Documentation

### 5.4.2.1 cnvDec\_asm

Viterbi Decoder

**Prototype:**

```
void
cnvDec_asm(int nBits,
           int *in_p,
           int *out_p,
           int flag)
```

**Description:**

This routine performs the trellis decoding. It has four modes of operation

- 0: Update metrics and transition history, trace and decodes all (for header packets)
- 1: Update metrics and transition history for only 1st block in payload
- 2: Update metrics and transition history, trace back through the current and previous blocks, decodes previous block giving nBits/2 bits
- 3: Update metrics and transition history, trace back through the current and previous blocks, decodes current and previous block giving nBits/2 bits

**Parameters:**

**nBits** Number of Coded bits for this block

**in\_p** Address of input buffer

**out\_p** Address of output buffer

**flag** Mode of operation

#### 5.4.2.2 cnvDecInit\_asm

Initialize Viterbi Decoder.

**Prototype:**

```
void  
cnvDecInit_asm(int nTranBits)
```

**Description:**

Initialize state metric table to a large negative value given by CNV\_DEC\_METRIC\_INIT and initialize the transition and wrap pointers

**Parameters:**

**nTranBits** Number of Coded bits

#### 5.4.2.3 cnvDecMetricRescale\_asm

State Metrics Rescale.

**Prototype:**

```
void  
cnvDecMetricRescale_asm()
```

**Description:**

Rescale the state metrics by finding the lowest metric and dividing the rest by it. This prevents overflow between successive decoder stages

### 5.4.3 Variable Documentation

#### 5.4.3.1 int32 [VIT\\_gold\\_vt\\_data\[\]](#)

Golden trace history (VT0/VT1); can be used to verify functionality.

5.4.3.2 int16 [VIT\\_in\\_data](#)[]

Input fed into the C-model encoder.

5.4.3.3 int16 [VIT\\_quant\\_data](#)[]

Output from the C-model encoder.

## 6 Application Programming Interface (VCU2)

### 6.1 VCU2 Type Definitions

#### Data Structures

- [complexShort\\_t](#)

#### Enumerations

- [Bool\\_e](#)

#### 6.1.1 Data Structure Documentation

##### 6.1.1.1 complexShort\_t

**Definition:**

```
typedef struct
{
    int16_t real;
    int16_t imag;
}
complexShort_t
```

**Members:**

***real*** Real Part.

***imag*** Imaginary Part.

**Description:**

Complex data (CPACK = 0).

On reset the CPACK bit is 0, therefore, this is the default complex structure

#### 6.1.2 Enumeration Documentation

##### 6.1.2.1 Bool\_e

**Description:**

Boolean enumeration.



## 6.2 Fast Fourier Transform (VCU2)

### Data Structures

- [\\_CFFT\\_Obj\\_](#)

### Functions

- void [CFFT\\_conjugate](#) (void \*pBuffer, uint16\_t size)
- void [CFFT\\_init1024Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_init128Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_init256Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_init32Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_init512Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_init64Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_pack](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_run1024Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_run128Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_run256Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_run32Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_run512Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_run64Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [CFFT\\_unpack](#) ([CFFT\\_Handle](#) hndCFFT)
- void [ICFFT\\_run1024Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [ICFFT\\_run128Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [ICFFT\\_run256Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [ICFFT\\_run32Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [ICFFT\\_run512Pt](#) ([CFFT\\_Handle](#) hndCFFT)
- void [ICFFT\\_run64Pt](#) ([CFFT\\_Handle](#) hndCFFT)

### Variables

- const int16\_t \* [vcu0\\_twiddleFactors](#)
- const int16\_t \* [vcu2\\_twiddleFactors](#)

## 6.2.1 Data Structure Documentation

### 6.2.1.1 \_CFFT\_Obj\_

**Definition:**

```
typedef struct
{
    int16_t *pInBuffer;
    int16_t *pOutBuffer;
    const int16_t *pTwiddleFactors;
    int16_t nSamples;
    int16_t nStages;
    int16_t twiddleSkipStep;
    void (*init)(void *);
    void (*run)(void *);
}
_CFFT_Obj_
```

**Members:**

***pInBuffer*** Input buffer pointer.  
***pOutBuffer*** Output buffer pointer.  
***pTwiddleFactors*** Twiddle Factor pointer.  
***nSamples*** Number of samples.  
***nStages*** HASH(0x270c8f0)  
***twiddleSkipStep*** Twiddle factor table search(skip) step.  
***init*** Function pointer to CFFT initialization routine.  
***run*** Function pointer to CFFT computation routine.

**Description:**

CFFT structure.

## 6.2.2 Function Documentation

### 6.2.2.1 CFFT\_conjugate

Take the complex conjugate of the entries in an array of complex numbers.

**Prototype:**

```
void
CFFT_conjugate(void *pBuffer,
               uint16_t size)
```

**Parameters:**

***pBuffer*** Pointer to the buffer of complex data to be conjugated  
← ***size*** Size of the buffer (multiple of 2 32-bits locations)

### 6.2.2.2 void CFFT\_init1024Pt ([CFFT\\_Handle](#) hndCFFT)

Initializes the CFFT object.

**Parameters:**

← ***hndCFFT*** handle to the CFFT object

**6.2.2.3 void CFFT\_init128Pt (CFFT\_Handle *hndCFFT*)**

Initializes the CFFT object.

**Parameters:**

← ***hndCFFT*** handle to the CFFT object

**6.2.2.4 void CFFT\_init256Pt (CFFT\_Handle *hndCFFT*)**

Initializes the CFFT object.

**Parameters:**

← ***hndCFFT*** handle to the CFFT object

**6.2.2.5 void CFFT\_init32Pt (CFFT\_Handle *hndCFFT*)**

Initializes the CFFT object.

This routine is used to initialize the CFFT object and must be called atleast once before using either the CFFT or ICFFT routines

**Parameters:**

← ***hndCFFT*** handle to the CFFT object

**6.2.2.6 void CFFT\_init512Pt (CFFT\_Handle *hndCFFT*)**

Initializes the CFFT object.

**Parameters:**

← ***hndCFFT*** handle to the CFFT object

**6.2.2.7 void CFFT\_init64Pt (CFFT\_Handle *hndCFFT*)**

Initializes the CFFT object.

**Parameters:**

← ***hndCFFT*** handle to the CFFT object

**6.2.2.8 void CFFT\_pack (CFFT\_Handle *hndCFFT*)**

Pack the input prior to running the inverse complex FFT to get the real inverse FFT.

In order to reverse the process of the forward real FFT,

$$F_e(k) = \frac{F(k) + F(\frac{N}{2} - k)^*}{2}$$

$$F_o(k) = \frac{F(k) - F(\frac{N}{2} - k)^*}{2} e^{j\frac{2\pi k}{N}}$$

where  $f_e$  is the even elements,  $f_o$  the odd elements. The array for the IFFT then becomes:

$$Z(k) = F_e(k) + jF_o(k), \quad k = 0 \dots \frac{N}{2} - 1$$

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Note:**

- This is an in-place algorithm; the routine writes the output to the input buffer itself
- The assumption is that the user will run the packed sequence through an IFFT sequence i.e. conjugate -> Forward FFT -> conjugate. The packed output is conjugated in this routine obviating the need for the first conjugate in the IFFT sequence

**See also:**

<http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM> for the entire derivation

#### 6.2.2.9 void CFFT\_run1024Pt (CFFT\_Handle hndCFFT)

Runs the Complex FFT routine.

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 1024 point complex FFT requires an input buffer of size 2048 words (16-bit), therefore it must be aligned to a boundary of 2048. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMGS4, ALIGN = 2048, PAGE = 1
```

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **pInBuffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOutBuffer** and **pInBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.10 void CFFT\_run128Pt (CFFT\_Handle hndCFFT)

Runs the Complex FFT routine.

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 128 point complex FFT requires an input buffer of size 256 words (16-bit), therefore it must be aligned to a boundary of 256. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMLS3, ALIGN = 256, PAGE = 1
```

#### 6.2.2.11 void CFFT\_run256Pt (CFFT\_Handle hndCFFT)

Runs the Complex FFT routine.

**Parameters:**

← *hndCFFT* handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 256 point complex FFT requires an input buffer of size 512 words (16-bit), therefore it must be aligned to a boundary of 512. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMLS3, ALIGN = 512, PAGE = 1
```

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **pInBuffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOutBuffer** and **pInBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.12 void CFFT\_run32Pt (CFFT\_Handle hndCFFT)

Runs the Complex FFT routine.

**Parameters:**

← *hndCFFT* handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 32 point complex FFT requires an input buffer of size 64 words (16-bit), therefore it must be aligned to a boundary of 64. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMLS3, ALIGN = 64, PAGE = 1
```

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **plnBuffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOutBuffer** and **plnBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.13 void CFFT\_run512Pt (CFFT\_Handle hndCFFT)

Runs the Complex FFT routine.

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 512 point complex FFT requires an input buffer of size 1024 words (16-bit), therefore it must be aligned to a boundary of 1024. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMGS4, ALIGN = 1024, PAGE = 1
```

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **plnBuffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOutBuffer** and **plnBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.14 void CFFT\_run64Pt (CFFT\_Handle hndCFFT)

Runs the Complex FFT routine.

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 64 point complex FFT requires an input buffer of size 128 words (16-bit), therefore it must be aligned to a boundary of 128. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMLS3, ALIGN = 128, PAGE = 1
```

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **pInBuffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOutBuffer** and **pInBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.15 void CFFT\_unpack (CFFT\_Handle hndCFFT)

Unpack the complex FFT output to get the FFT of two interleaved real sequences.

In order to get the FFT of a real N-pt sequences, we treat the input as an N/2 pt complex sequence, take its complex FFT, use the following properties to get the N-pt Fourier transform of the real sequence

$$FFT_n(k, f) = FFT_{N/2}(k, f_e) + e^{-\frac{j2\pi k}{N}} FFT_{N/2}(k, f_o)$$

where  $f_e$  is the even elements,  $f_o$  the odd elements and

$$F_e(k) = \frac{Z(k) + Z(\frac{N}{2} - k)^*}{2}$$

$$F_o(k) = -j \frac{Z(k) - Z(\frac{N}{2} - k)^*}{2}$$

We get the first N/2 points of the FFT by combining the above two equations

$$F(k) = F_e(k) + e^{-\frac{j2\pi k}{N}} F_o(k)$$

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Note:**

This is an in-place algorithm; the routine writes the output to the input buffer itself

**See also:**

<http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM> for the entire derivation

#### 6.2.2.16 void ICFFT\_run1024Pt (CFFT\_Handle hndCFFT)

Runs the Complex Inverse FFT routine.

Run the forward FFT on the input and rearrange the output as follows:

$$x(0) = x'(0)$$

$$x(n) = x'(N - n), n \in \{1, N - 1\}$$

, where N is the sample size

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 1024 point complex FFT requires an input buffer of size 2048 words (16-bit), therefore it must be aligned to a boundary of 2048. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMGS4, ALIGN = 2048, PAGE = 1
```

If the output buffer of the forward FFT becomes the input to the IFFT, then it must be aligned to the same word (16-bit) boundary as well.

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **plnBuffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOutBuffer** and **plnBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.17 void ICFFT\_run128Pt (CFFT\_Handle hndCFFT)

Runs the Complex Inverse FFT routine.

Run the forward FFT on the input and rearrange the output as follows:

$$x(0) = x'(0)$$

$$x(n) = x'(N - n), n \in \{1, N - 1\}$$

, where N is the sample size

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 128 point complex FFT requires an input buffer of size 256 words (16-bit), therefore it must be aligned to a boundary of 256. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMLS3, ALIGN = 256, PAGE = 1
```

If the output buffer of the forward FFT becomes the input to the IFFT, then it must be aligned to the same word (16-bit) boundary as well.

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **plnBuffer** and **pOutBuffer**, at every stage. Depending on the number of stages the



output may be in either of the two buffers; the algorithm will switch the pointers **pOutBuffer** and **pInBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.18 void ICFFT\_run256Pt (CFFT\_Handle hndCFFT)

Runs the Complex Inverse FFT routine.

Run the forward FFT on the input and rearrange the output as follows:

$$x(0) = x'(0)$$

$$x(n) = x'(N - n), n \in \{1, N - 1\}$$

, where N is the sample size

##### Parameters:

← **hndCFFT** handle to the CFFT object

##### Attention:

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 256 point complex FFT requires an input buffer of size 512 words (16-bit), therefore it must be aligned to a boundary of 512. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMLS3, ALIGN = 512, PAGE = 1
```

If the output buffer of the forward FFT becomes the input to the IFFT, then it must be aligned to the same word (16-bit) boundary as well.

#### 6.2.2.19 void ICFFT\_run32Pt (CFFT\_Handle hndCFFT)

Runs the Complex Inverse FFT routine.

Run the forward FFT on the input and rearrange the output as follows:

$$x(0) = x'(0)$$

$$x(n) = x'(N - n), n \in \{1, N - 1\}$$

, where N is the sample size

##### Parameters:

← **hndCFFT** handle to the CFFT object

##### Attention:

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 32 point complex FFT requires an input buffer of size 64 words (16-bit), therefore it must be aligned to a boundary of 64. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMLS3, ALIGN = 64, PAGE = 1
```

If the output buffer of the forward FFT becomes the input to the IFFT, then it must be aligned to the same word (16-bit) boundary as well.

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **pln-Buffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOut-Buffer** and **plnBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.20 void ICFFT\_run512Pt (CFFT\_Handle hndCFFT)

Runs the Complex Inverse FFT routine.

Run the forward FFT on the input and rearrange the output as follows:

$$x(0) = x'(0)$$

$$x(n) = x'(N - n), n \in \{1, N - 1\}$$

, where N is the sample size

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 512 point complex FFT requires an input buffer of size 1024 words (16-bit), therefore it must be aligned to a boundary of 1024. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMGS4, ALIGN = 1024, PAGE = 1
```

If the output buffer of the forward FFT becomes the input to the IFFT, then it must be aligned to the same word (16-bit) boundary as well.

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **pln-Buffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOut-Buffer** and **plnBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

#### 6.2.2.21 void ICFFT\_run64Pt (CFFT\_Handle hndCFFT)

Runs the Complex Inverse FFT routine.

Run the forward FFT on the input and rearrange the output as follows:

$$x(0) = x'(0)$$

$$x(n) = x'(N - n), n \in \{1, N - 1\}$$

, where N is the sample size

**Parameters:**

← **hndCFFT** handle to the CFFT object

**Attention:**

For bit reverse addressing to work, the input buffer must be aligned to size of the buffer in words (16-bit). For example, the 64 point complex FFT requires an input buffer of size 128 words (16-bit), therefore it must be aligned to a boundary of 128. This can be done by assigning the array to a named section (buffer1) using compiler pragmas

```
#pragma DATA_SECTION(buffer1Q15, "buffer1")
```

and then either assigning this memory to the start of a RAM block in the linker command file or aligning it to a boundary using the align directive

```
buffer1 : > RAMLS3, ALIGN = 128, PAGE = 1
```

If the output buffer of the forward FFT becomes the input to the IFFT, then it must be aligned to the same word (16-bit) boundary as well.

**Note:**

The algorithm ping-pongs between the two buffers, i.e. the buffers pointed to by **pInBuffer** and **pOutBuffer**, at every stage. Depending on the number of stages the output may be in either of the two buffers; the algorithm will switch the pointers **pOutBuffer** and **pInBuffer** in the event that the output ends up in the original input buffer, with the end result that **pOutBuffer** always points to the output.

## 6.2.3 Variable Documentation

### 6.2.3.1 const int16\_t\* [vcu0\\_twiddleFactors](#)

VCU0 twiddle factors.

### 6.2.3.2 const int16\_t\* [vcu2\\_twiddleFactors](#)

VCU2 twiddle factors.

## 6.2.4 Real Fast Fourier Transform

It is possible to run the Fast Fourier Transform on a sequence of real data using the complex FFT. For a 2N point real sequence, the user would treat the data as N-pt complex (no rearrangement required) and run it through an N point complex FFT. In order to derive the correct spectrum, you would have to “unpack” the output. The derivations can be found here:

<http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM>

Similarly, to run an inverse Real FFT, the user would “pack” the data and run it through an N-point Forward Complex FFT and then conjugate its complex output to get the original signal.

**Note 1** When running an inverse real FFT after the forward real FFT, the user must take care to first switch the **Input (pInBuffer)** and **Output (pOutBuffer)** pointers in the FFT object before calling the FFT routine again

**Note 2** Because the buffers are switched for the inverse FFT, they must both be aligned to a 2N word boundary. **buffer1Q15** must be aligned since it is the input to the forward real FFT, while **buffer2Q15** is the input to the inverse real FFT; it must also be aligned

**Note 3** The **pack**, **unpack**, and **FFT** routines scale down the input data to prevent overflows. Therefore, the output of the real inverse FFT process will be a scaled down version of the original. The user may choose to scale the output of intermediate operations to prevent small values being zeroed out

**Note 4** Refer to the project, **2837x\_vcu\_rfft\_128**, in the examples folder for a demonstration of the entire process

**See also:**

[CFFT\\_pack](#), [CFFT\\_unpack](#), [CFFT\\_conjugate](#)

## 6.3 Cyclic Redundancy Check (VCU2)

### Data Structures

- [\\_CRC\\_Obj\\_](#)

### Defines

- [INIT\\_CRC16](#)
- [INIT\\_CRC24](#)
- [INIT\\_CRC32](#)
- [INIT\\_CRC8](#)

### Enumerations

- [CRC\\_parity\\_e](#)

### Functions

- [uint32\\_t CRC\\_bitReflect](#) ([uint32\\_t](#) valToReverse, [int16\\_t](#) bitWidth)
- [void CRC\\_init16Bit](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_init24Bit](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_init32Bit](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_init8Bit](#) ([CRC\\_Handle](#) hndCRC)
- [uint16\\_t CRC\\_pow2](#) ([uint16\\_t](#) power)
- [void CRC\\_reset](#) ([void](#))
- [void CRC\\_run16BitPoly1](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run16BitPoly1Reflected](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run16BitPoly2](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run16BitPoly2Reflected](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run16BitReflectedTableLookupC](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run16BitTableLookupC](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run24Bit](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run24BitReflected](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run24BitReflectedTableLookupC](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run24BitTableLookupC](#) ([CRC\\_Handle](#) hndCRC)
- [void CRC\\_run32BitPoly1](#) ([CRC\\_Handle](#) hndCRC)

- void [CRC\\_run32BitPoly1Reflected](#) (CRC\_Handle hndCRC)
- void [CRC\\_run32BitPoly2](#) (CRC\_Handle hndCRC)
- void [CRC\\_run32BitPoly2Reflected](#) (CRC\_Handle hndCRC)
- void [CRC\\_run32BitReflectedTableLookupC](#) (CRC\_Handle hndCRC)
- void [CRC\\_run32BitTableLookupC](#) (CRC\_Handle hndCRC)
- void [CRC\\_run8Bit](#) (CRC\_Handle hndCRC)
- void [CRC\\_run8BitReflected](#) (CRC\_Handle hndCRC)
- void [CRC\\_run8BitTableLookupC](#) (CRC\_Handle hndCRC)

## 6.3.1 Data Structure Documentation

### 6.3.1.1 \_CRC\_Obj\_

#### Definition:

```
typedef struct
{
    uint32_t seedValue;
    uint16_t nMsgBytes;
    CRC\_parity\_e parity;
    uint32_t crcResult;
    void *pMsgBuffer;
    void *pCrcTable;
    void (*init)(void *);
    void (*run)(void *);
}
_CRC_Obj_
```

#### Members:

**seedValue** Initial value of the CRC calculation.

**nMsgBytes** the number of bytes in the message buffer

**parity** start the CRC from the low byte (CRC\_parity\_even) or high byte (CRC\_parity\_odd) of the first word (16-bit)

**crcResult** the calculated CRC

**pMsgBuffer** Pointer to the message buffer.

**pCrcTable** Pointer to the CRC lookup table.

**init** Function pointer to CRC initialization routine.

**run** Function pointer to CRC computation routine.

#### Description:

CRC structure.

## 6.3.2 Define Documentation

### 6.3.2.1 INIT\_CRC16

**Definition:**

```
#define INIT_CRC16
```

**Description:**

Initial CRC Register Value.

### 6.3.2.2 INIT\_CRC24

**Definition:**

```
#define INIT_CRC24
```

**Description:**

Initial CRC Register Value.

### 6.3.2.3 INIT\_CRC32

**Definition:**

```
#define INIT_CRC32
```

**Description:**

Initial CRC Register Value.

### 6.3.2.4 INIT\_CRC8

**Definition:**

```
#define INIT_CRC8
```

**Description:**

Initial CRC Register Value.

## 6.3.3 Typedef Documentation

### 6.3.3.1 CRC\_Handle

**Definition:**

```
typedef CRC_Obj *CRC_Handle
```

**Description:**

Handle to the CRC structure.

### 6.3.3.2 CRC\_Obj

**Definition:**

```
typedef struct _CRC_Obj_ CRC_Obj
```

**Description:**  
CRC structure.

## 6.3.4 Enumeration Documentation

### 6.3.4.1 CRC\_parity\_e

**Description:**  
Parity enumeration.

The parity is used by the CRC algorithm to determine whether to begin calculations from the low byte (EVEN) or from the high byte (ODD) of the first word (16-bit) in the message.

For example, if your message had 10 bytes and started at the address 0x8000 but the first byte was at the high byte position of the first 16-bit word, the user would call the CRC function with odd parity i.e. CRC\_parity\_odd

Address: HI LO

0x8000 : B0 XX

0x8001 : B2 B1

0x8002 : B4 B3

0x8003 : B6 B5

0x8004 : B8 B7

0x8005 : XX B9

However, if the first byte was at the low byte position of the first 16-bit word, the user would call the CRC function with even parity i.e. CRC\_parity\_even

Address: HI LO

0x8000 : B1 B0

0x8001 : B3 B2

0x8002 : B5 B4

0x8003 : B7 B6

0x8004 : B9 B8

**Enumerators:**  
**CRC\_parity\_even** Even parity, CRC starts at the low byte of the first word (16-bit).  
**CRC\_parity\_odd** Odd parity, CRC starts at the high byte of the first word (16-bit).

## 6.3.5 Function Documentation

### 6.3.5.1 CRC\_bitReflect

Bit-reverse a value.

**Prototype:**  
uint32\_t



```
CRC_bitReflect(uint32_t valToReverse,  
               int16_t bitWidth)
```

**Description:**

Bit reverse a given hex value, The number of bits must be a power of 2

**Parameters:**

***valToReverse*** Value to reverse

***bitWidth*** Bit-width of the input, must be a power of 2

**Returns:**

bit-reversed value

### 6.3.5.2 CRC\_init16Bit

Initializes the CRC object.

**Prototype:**

```
void  
CRC_init16Bit(CRC_Handle hndCRC)
```

**Description:**

Clears the CRCMSGFLIP bit is cleared ensuring the input is interpreted in normal bit-order

**Parameters:**

← ***hndCRC*** handle to the CRC object

### 6.3.5.3 CRC\_init24Bit

Initializes the CRC object.

**Prototype:**

```
void  
CRC_init24Bit(CRC_Handle hndCRC)
```

**Description:**

Clears the CRCMSGFLIP bit is cleared ensuring the input is interpreted in normal bit-order

**Parameters:**

← ***hndCRC*** handle to the CRC object

### 6.3.5.4 CRC\_init32Bit

Initializes the CRC object.

**Prototype:**

```
void  
CRC_init32Bit(CRC_Handle hndCRC)
```

**Description:**

Clears the CRCMSGFLIP bit is cleared ensuring the input is interpreted in normal bit-order

**Parameters:**

← ***hndCRC*** handle to the CRC object

#### 6.3.5.5 CRC\_init8Bit

Initializes the CRC object.

**Prototype:**

```
void  
CRC_init8Bit (CRC_Handle hndCRC)
```

**Description:**

Clears the CRCMSGFLIP bit is cleared ensuring the input is interpreted in normal bit-order

**Parameters:**

← ***hndCRC*** handle to the CRC object

#### 6.3.5.6 CRC\_pow2

power of 2

**Prototype:**

```
uint16_t  
CRC_pow2(uint16_t power)
```

**Description:**

recursive function to calculate a positive integer that is a power of two

**Parameters:**

***power*** The exponent of two

**Returns:**

an integer that is a power of two

#### 6.3.5.7 CRC\_reset

Workaround to the silicon issue of first VCU calculation on power up being erroneous.

**Prototype:**

```
void  
CRC_reset(void)
```

**Description:**

Details Due to the internal power-up state of the VCU module, it is possible that the first CRC result will be incorrect. This condition applies to the first result from each of the eight CRC instructions. This rare condition can only occur after a power-on reset, but will not necessarily occur on every power on. A warm reset will not cause this condition to reappear. Workaround(s): The application can reset the internal VCU CRC logic by performing a CRC calculation of a single byte in the initialization routine. This routine only needs to perform one CRC calculation and can use any of the CRC instructions

#### 6.3.5.8 void CRC\_run16BitPoly1 (CRC\_Handle hndCRC)

Runs the CRC routine using polynomial 0x8005.

Calculates the 16-bit CRC using polynomial 0x8005 on the VCU. Depending on the parity chosen the CRC begins at either the low byte (PARITY\_LOWBYTE) or the high byte (PARITY\_HIGHBYTE) of the first word (16-bit).

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

#### 6.3.5.9 CRC\_run16BitPoly1Reflected

Runs the 16-bit CRC routine using polynomial 0x8005 with the input bits reversed.

**Prototype:**

```
void  
CRC_run16BitPoly1Reflected(CRC_Handle hndCRC)
```

**Description:**

By setting the CRCMSGFLIP bit, the input is fed through the VCU 16-bit CRC calculator (polynomial 0x8005) in reverse bit order

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

#### 6.3.5.10 CRC\_run16BitPoly2

Runs the CRC routine using polynomial 0x1021.

**Prototype:**

```
void  
CRC_run16BitPoly2(CRC_Handle hndCRC)
```

**Description:**

Calculates the 16-bit CRC using polynomial 0x1021 on the VCU. Depending on the parity chosen the CRC begins at either the low byte (PARITY\_LOWBYTE) or the high byte (PARITY\_HIGHBYTE) of the first word (16-bit).

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

#### 6.3.5.11 CRC\_run16BitPoly2Reflected

Runs the 16-bit CRC routine using polynomial 0x1021 with the input bits reversed.

**Prototype:**

```
void  
CRC_run16BitPoly2Reflected(CRC_Handle hndCRC)
```

**Description:**

By setting the CRCMSGFLIP bit, the input is fed through the VCU 16-bit CRC calculator (polynomial 0x1021) in reverse bit order

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

### 6.3.5.12 CRC\_run16BitReflectedTableLookupC

C table-lookup 16-bit CRC calculation(reflected algorithm).

**Prototype:**

```
void  
CRC_run16BitReflectedTableLookupC(CRC_Handle hndCRC)
```

**Description:**

The CRC is calculated using a table lookup method, where each byte of the input is an index into the table. The value at that index is XOR'd into a variable called the accumulator. Once the final byte's CRC is looked up and accumulated we get the CRC for the entire message block

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

**See also:**

[http://www.ross.net/crc/download/crc\\_v3.txt](http://www.ross.net/crc/download/crc_v3.txt)

### 6.3.5.13 CRC\_run16BitTableLookupC

C table-lookup 16-bit CRC calculation.

**Prototype:**

```
void  
CRC_run16BitTableLookupC(CRC_Handle hndCRC)
```

**Description:**

The CRC is calculated using a table lookup method, where each byte of the input is an index into the table. The value at that index is XOR'd into a variable called the accumulator. Once the final byte's CRC is looked up and accumulated we get the CRC for the entire message block

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

**See also:**

[http://www.ross.net/crc/download/crc\\_v3.txt](http://www.ross.net/crc/download/crc_v3.txt)

#### 6.3.5.14 CRC\_run24Bit

Runs the CRC routine.

**Prototype:**

```
void  
CRC_run24Bit (CRC_Handle hndCRC)
```

**Description:**

Calculates the 24-bit CRC using polynomial 0x5d6dcb on the VCU. Depending on the parity chosen the CRC begins at either the low byte (PARITY\_LOWBYTE) or the high byte (PARITY\_HIGHBYTE) of the first word (16-bit).

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

#### 6.3.5.15 CRC\_run24BitReflected

Runs the 24-bit CRC routine using polynomial 0x5d6dcb with the input bits reversed.

**Prototype:**

```
void  
CRC_run24BitReflected (CRC_Handle hndCRC)
```

**Description:**

By setting the CRCMSGFLIP bit, the input is fed through the VCU 24-bit CRC calculator (polynomial 0x5d6dcb) in reverse bit order

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

#### 6.3.5.16 CRC\_run24BitReflectedTableLookupC

C table-lookup 24-bit CRC calculation(reflected algorithm).

**Prototype:**

```
void  
CRC_run24BitReflectedTableLookupC (CRC_Handle hndCRC)
```

**Description:**

The CRC is calculated using a table lookup method, where each byte of the input is an index into the table. The value at that index is XOR'd into a variable called the accumulator. Once the final byte's CRC is looked up and accumulated we get the CRC for the entire message block

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

**See also:**

[http://www.ross.net/crc/download/crc\\_v3.txt](http://www.ross.net/crc/download/crc_v3.txt)

### 6.3.5.17 CRC\_run24BitTableLookupC

C table-lookup 24-bit CRC calculation.

**Prototype:**

```
void  
CRC_run24BitTableLookupC(CRC_Handle hndCRC)
```

**Description:**

The CRC is calculated using a table lookup method, where each byte of the input is an index into the table. The value at that index is XOR'd into a variable called the accumulator. Once the final byte's CRC is looked up and accumulated we get the CRC for the entire message block

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

**See also:**

[http://www.ross.net/crc/download/crc\\_v3.txt](http://www.ross.net/crc/download/crc_v3.txt)

### 6.3.5.18 CRC\_run32BitPoly1

Runs the 32-bit CRC routine using polynomial 0x04c11db7.

**Prototype:**

```
void  
CRC_run32BitPoly1(CRC_Handle hndCRC)
```

**Description:**

Calculates the 32-bit CRC using polynomial 0x04c11db7 on the VCU. Depending on the parity chosen the CRC begins at either the low byte (PARITY\_LOWBYTE) or the high byte (PARITY\_HIGHBYTE) of the first word (16-bit).

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

### 6.3.5.19 CRC\_run32BitPoly1Reflected

Runs the 32-bit CRC routine using polynomial 0x04c11db7 with the input bits reversed.

**Prototype:**

```
void  
CRC_run32BitPoly1Reflected(CRC_Handle hndCRC)
```

**Description:**

By setting the CRCMSGFLIP bit, the input is fed through the VCU 32-bit CRC calculator (polynomial 0x04c11db7) in reverse bit order

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

### 6.3.5.20 CRC\_run32BitPoly2

Runs the 32-bit CRC routine using polynomial 0x1edc6f41.

**Prototype:**

```
void  
CRC_run32BitPoly2(CRC_Handle hndCRC)
```

**Description:**

Calculates the 32-bit CRC using polynomial 0x1edc6f41 on the VCU. Depending on the parity chosen the CRC begins at either the low byte (PARITY\_LOWBYTE) or the high byte (PARITY\_HIGHBYTE) of the first word (16-bit).

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

### 6.3.5.21 CRC\_run32BitPoly2Reflected

Runs the 32-bit CRC routine using polynomial 0x1edc6f41 with the input bits reversed.

**Prototype:**

```
void  
CRC_run32BitPoly2Reflected(CRC_Handle hndCRC)
```

**Description:**

By setting the CRCMSGFLIP bit, the input is fed through the VCU 32-bit CRC calculator (polynomial 0x1edc6f41) in reverse bit order

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

### 6.3.5.22 CRC\_run32BitReflectedTableLookupC

C table-lookup 32-bit CRC calculation(reflected algorithm).

**Prototype:**

```
void  
CRC_run32BitReflectedTableLookupC(CRC_Handle hndCRC)
```

**Description:**

The CRC is calculated using a table lookup method, where each byte of the input is an index into the table. The value at that index is XOR'd into a variable called the accumulator. Once the final byte's CRC is looked up and accumulated we get the CRC for the entire message block

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

**See also:**

[http://www.ross.net/crc/download/crc\\_v3.txt](http://www.ross.net/crc/download/crc_v3.txt)

### 6.3.5.23 CRC\_run32BitTableLookupC

C table-lookup 32-bit CRC calculation.

**Prototype:**

```
void  
CRC_run32BitTableLookupC(CRC_Handle hndCRC)
```

**Description:**

The CRC is calculated using a table lookup method, where each byte of the input is an index into the table. The value at that index is XOR'd into a variable called the accumulator. Once the final byte's CRC is looked up and accumulated we get the CRC for the entire message block

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

**See also:**

[http://www.ross.net/crc/download/crc\\_v3.txt](http://www.ross.net/crc/download/crc_v3.txt)



#### 6.3.5.24 CRC\_run8Bit

Calculate the 8-bit CRC using polynomial 0x7.

**Prototype:**

```
void  
CRC_run8Bit (CRC_Handle hndCRC)
```

**Description:**

Calculates the 8-bit CRC using polynomial 0x7 on the VCU. Depending on the parity chosen the CRC begins at either the low byte (PARITY\_LOWBYTE) or the high byte (PARITY\_HIGHBYTE) of the first word (16-bit).

**Note:**

the size of the message (bytes) is limited to 65535 bytes. If attempting to process a larger message, the user must break it into pieces of size 65535 or smaller, and successively run the CRC on each block, with the CRC result of one block becoming the seed value for the next block. An example of this is shown in the FLASH build configuration of the example **2837x\_vcu2\_crc\_8**.

**Parameters:**

← **hndCRC** handle to the CRC object

#### 6.3.5.25 CRC\_run8BitReflected

Runs the 8-bit CRC routine using polynomial 0x7 with the input bits reversed.

**Prototype:**

```
void  
CRC_run8BitReflected (CRC_Handle hndCRC)
```

**Description:**

By setting the CRCMSGFLIP bit, the input is fed through the VCU 8-bit CRC calculator (polynomial 0x7) in reverse bit order

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

#### 6.3.5.26 CRC\_run8BitTableLookupC

C table-lookup 8-bit CRC calculation.

**Prototype:**

```
void  
CRC_run8BitTableLookupC (CRC_Handle hndCRC)
```

**Description:**

The CRC is calculated using a table lookup method, where each byte of the input is an index into the table. The value at that index is XOR'd into a variable called the accumulator. Once the final byte's CRC is looked up and accumulated we get the CRC for the entire message block

**Note:**

the size of the message (bytes) is limited to 65535 bytes Please see the notes for the function **CRC\_run8Bit** for details

**Parameters:**

← **hndCRC** handle to the CRC object

**See also:**

[http://www.ross.net/crc/download/crc\\_v3.txt](http://www.ross.net/crc/download/crc_v3.txt)

## 6.4 Viterbi Decoding (VCU2)

### Data Structures

- [\\_VITERBI\\_DECODER\\_Obj\\_](#)

### Enumerations

- [VITERBIMODE\\_e](#)

### Functions

- void [VITERBI\\_DECODER\\_initK4CR12](#) ([VITERBI\\_DECODER\\_Handle](#) hndVITDe-coder)
- void [VITERBI\\_DECODER\\_initK7CR12](#) ([VITERBI\\_DECODER\\_Handle](#) hndVITDe-coder)
- void [VITERBI\\_DECODER\\_rescaleK4CR12](#) ([VITERBI\\_DECODER\\_Handle](#) hndVITDe-coder)
- void [VITERBI\\_DECODER\\_rescaleK7CR12](#) ([VITERBI\\_DECODER\\_Handle](#) hndVITDe-coder)
- void [VITERBI\\_DECODER\\_runK4CR12](#) ([VITERBI\\_DECODER\\_Handle](#) hndVITDe-coder)
- void [VITERBI\\_DECODER\\_runK7CR12](#) ([VITERBI\\_DECODER\\_Handle](#) hndVITDe-coder)

### 6.4.1 Data Structure Documentation

#### 6.4.1.1 [\\_VITERBI\\_DECODER\\_Obj\\_](#)

**Definition:**

```
typedef struct
{
    int16_t *pInBuffer;
    uint16_t *pOutBuffer;
    uint16_t *pTransitionHistory;
    const int32_t *pBMSELInit;
    int16_t stateMetricInit;
    int16_t nBits;
    int16_t constraintLength;
    int16_t nStates;
    int16_t codeRate;
    VITERBIMODE\_e mode;
    uint16_t *pTransitionStart1;
    uint16_t *pTransitionStart2;
```

```

uint16_t *pTransitionWrap1;
uint16_t *pTransitionWrap2;
uint16_t *pTransitionTemp;
void (*init)(void *);
void (*run)(void *);
void (*rescale)(void *);
}
_VITERBI_DECODER_Obj_

```

**Members:**

***pInBuffer*** Input buffer pointer.

***pOutBuffer*** Output buffer pointer.

***pTransitionHistory*** Transition History pointer.

***pBMSELInit*** Initialization value for the BMSEL register.

***stateMetricInit*** Initialization value for the state metrics.

***nBits*** Total number of bits to be decoded.

***constraintLength*** Constraint Length, i.e. K.

***nStates*** HASH(0x26fb8a8)

***codeRate*** The symbol code rate.

***mode*** Viterbi mode enumerator.

***pTransitionStart1*** Points to the start of the tranistion history buffer.

***pTransitionStart2*** Points to the mid of the tranistion history buffer.

***pTransitionWrap1*** Points to the mid of the tranistion history buffer.

***pTransitionWrap2*** Points to the end of the tranistion history buffer.

***pTransitionTemp*** Points to a temporary(scratch) tranistion history buffer.

***init*** Function pointer to VITERBI initialization routine.

***run*** Function pointer to VITERBI computation routine.

***rescale*** Function pointer to VITERBI rescale routine.

**Description:**

VITERBI Decoder Structure.

## 6.4.2 Enumeration Documentation

### 6.4.2.1 VITERBIMODE\_e

**Description:**

The Viterbi mode enumerator.

**Enumerators:**

***VITERBIMODE\_DECODEALL*** Decodes all output bits, upto a max of 256, at once.

***VITERBIMODE\_OVERLAPINIT*** no traceback is performed

Use window overlap method, This is used for the first block where state metrics and transition history is updated but

***VITERBIMODE\_OVERLAPDECODE*** Use window overlap method, update transitions/metrics for the current block (ith block), run a traceback using the ith and (i-1)st block's transition history but only decode the (i-1)st block

**VITERBIMODE\_OVERLAPLAST** Trace back and decode the last block in overlap window method.

## 6.4.3 Function Documentation

### 6.4.3.1 VITERBI\_DECODER\_initK4CR12

Initializes the VITERBI object (constraint length 4, code rate 1/2).

**Prototype:**

```
void  
VITERBI_DECODER_initK4CR12 (VITERBI_DECODER_Handle  
hndVITDecoder)
```

**Description:**

Sets the constraint length of the viterbi object and initialized the state metrics to the object element, stateMetricInit

**Parameters:**

← **hndVITDecoder** handle to the VITERBI object

### 6.4.3.2 VITERBI\_DECODER\_initK7CR12

Initializes the VITERBI object (constraint length 7, code rate 1/2).

**Prototype:**

```
void  
VITERBI_DECODER_initK7CR12 (VITERBI_DECODER_Handle  
hndVITDecoder)
```

**Description:**

Sets the constraint length of the viterbi object and initialized the state metrics to the object element, stateMetricInit

**Note:**

This function uses a global variable to save off the metric registers and is, therefore, non re-entrant

**Parameters:**

← **hndVITDecoder** handle to the VITERBI object

### 6.4.3.3 VITERBI\_DECODER\_rescaleK4CR12

Rescales the viterbi state metrics (constraint length 4, code rate 1/2).

**Prototype:**

```
void  
VITERBI_DECODER_rescaleK4CR12 (VITERBI_DECODER_Handle  
hndVITDecoder)
```

**Description:**

Rescale the state metrics by finding the lowest metric and dividing the rest by it. This prevents overflow between successive decoder stages.

**Parameters:**

← *hndVITDecoder* handle to the VITERBI object

#### 6.4.3.4 VITERBI\_DECODER\_rescaleK7CR12

Rescales the viterbi state metrics (constraint length 7, code rate 1/2).

**Prototype:**

```
void  
VITERBI_DECODER_rescaleK7CR12 (VITERBI_DECODER_Handle  
hndVITDecoder)
```

**Description:**

Rescale the state metrics by finding the lowest metric and dividing the rest by it. This prevents overflow between successive decoder stages.

**Parameters:**

← *hndVITDecoder* handle to the VITERBI object

#### 6.4.3.5 VITERBI\_DECODER\_runK4CR12

Runs the VITERBI decoder for constraint length 4, code rate 1/2.

**Prototype:**

```
void  
VITERBI_DECODER_runK4CR12 (VITERBI_DECODER_Handle  
hndVITDecoder)
```

**Description:**

The viterbi decode is done using a window overlap method with 4 modes of operation :

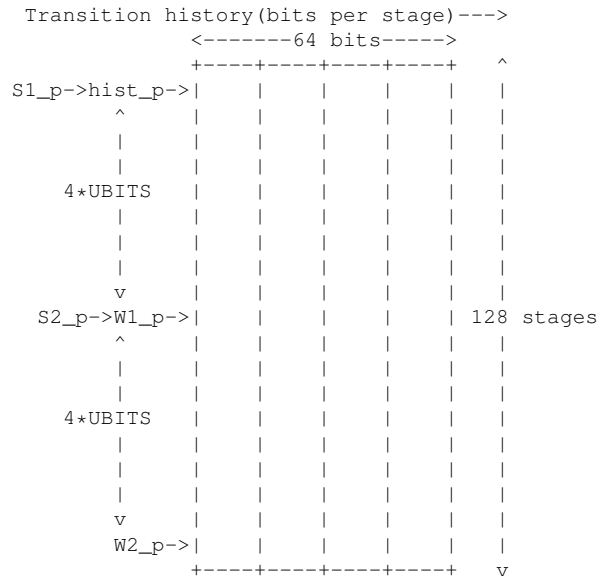
1. VITERBIMODE\_DECODEALL, a one-shot decode mode typically used for header information where the entire block of data is processed through the trellis and decoded
2. VITERBIMODE\_OVERLAPINIT, window overlap method – this is used for the first block where state metrics and transition history is updated but no traceback is performed
3. VITERBIMODE\_OVERLAPDECODE, window overlap method – update transitions/metrics for the current block (ith block), run a traceback using the ith and (i-1)st block's transition history but only decode the (i-1)st block
4. VITERBIMODE\_OVERLAPLAST, window overlap method– trace back and decode the last block

The window overlap method requires the transition history of two successive blocks to be recorded. The transition history buffer is used in a circular fashion and requires 5 pointers:

- pTransitionHistory(hist\_p): start of the transition history buffer
- pTransitionStart1(S1\_p): points to where the transition update should start
- pTransitionStart2(S2\_p: points to the mid point of the overlap(S1\_p + 4\*nUnencodedBits)
- pTransitionWrap1(W1\_p): points to where trace overlap 2 should go (wrap, S1\_p + 4\*nUnencodedBits)

- `pTransitionWrap2(W2_p)`: points to the end of the overlap( $S1\_p + 2*4*nUnencodedBits$ )

$CBITS = 128$  (coded bits per block)  
 $UBITS = CBITS/2 = 64$  (uncoded bits per block)  
 $UWORDS = 4$  (4 words (16-bits) required to store UBITS)



**Parameters:**

← ***hndVITDecoder*** handle to the VITERBI object

#### 6.4.3.6 void VITERBI\_DECODER\_runK7CR12 ([VITERBI\\_DECODER\\_Handle](#) *hndVITDecoder*)

Runs the VITERBI decoder for constraint length 7, code rate 1/2.

**Parameters:**

← ***hndVITDecoder*** handle to the VITERBI object

**See also:**

[VITERBI\\_DECODER\\_runK4CR12](#) for a description of the window overlap method

## 6.5 Reed Solomon Decoder (VCU2)

### Data Structures

- [\\_REEDSOLOMON\\_DECODER\\_Obj\\_](#)

### Defines

- [RS\\_BLOCK\\_K](#)
- [RS\\_BLOCK\\_N](#)
- [RS\\_BLOCK\\_T](#)
- [RS\\_NROOTS](#)

### Functions

- void [REEDSOLOMON\\_DECODER\\_berlekampMassey](#) (REEDSOLOMON\_DECODER\_Handle hndRSDecoder)
- void [REEDSOLOMON\\_DECODER\\_calcSyndrome](#) (REEDSOLOMON\_DECODER\_Handle hndRSDecoder, int16\_t \*pData, int16\_t nBytes)
- void [REEDSOLOMON\\_DECODER\\_chienForney](#) (REEDSOLOMON\_DECODER\_Handle hndRSDecoder, int16\_t nBytes)
- void [REEDSOLOMON\\_DECODER\\_initN255K239](#) (REEDSOLOMON\_DECODER\_Handle hndRSDecoder, int16\_t \*pSyndrome, int16\_t \*pLambda, int16\_t \*pOmega, int16\_t \*pPackedAlpha, int16\_t \*pPackedBeta, int16\_t \*pRS\_expTable, int16\_t \*pRS\_logTable, ERROR\_LOCVAL\_Obj \*pErrorLoc)
- void [REEDSOLOMON\\_DECODER\\_runN255K239](#) (REEDSOLOMON\_DECODER\_Handle hndRSDecoder, int16\_t \*pData, int16\_t nBytes)

### 6.5.1 Data Structure Documentation

#### 6.5.1.1 [\\_REEDSOLOMON\\_DECODER\\_Obj\\_](#)

##### Definition:

```
typedef struct
{
    uint16_t _n;
    uint16_t _k;
    uint16_t _t;
    uint16_t nRoots;
    int16_t *pSyndrome;
    int16_t *pLambda;
    int16_t *pOmega;
    int16_t *pPackedAlpha;
    int16_t *pPackedBeta;
    int16_t *pRS_expTable;
    int16_t *pRS_logTable;
    ERROR_LOCVAL_Obj *pErrorLoc;
}
```



```

        void (*init)(void *,
                      int16_t *,
                      int16_t *,
                      int16_t *,
                      int16_t *,
                      int16_t *,
                      int16_t *,
                      int16_t *,
                      ERROR_LOCVAL_Obj *);
        void (*run)(void *,
                    int16_t *,
                    int16_t);
    }
    _REEDSOLOMON_DECODER_Obj_

```

**Members:**

**\_n** number of codeword symbols (bytes) in a block

**\_k** number of message symbols (bytes) in a block

**\_t** number of correctable errors in the block

**nRoots** number of roots for the code generator polynomial

**pSyndrome** pointer to the syndromes

**pLambda** pointer to the Lambdas

**pOmega** pointer to the Omega

**pPackedAlpha** Pointer to the roots of the code generator polynomial.

**pPackedBeta** Pointer to the first 2t elements of the Galois Field.

**pRS\_expTable** Pointer to the lookup table (roots of the extension Galois Field) that converts index to decimal form.

**pRS\_logTable** Pointer to the lookup table (roots of the extension Galois Field) that converts decimal to index form.

**pErrorLoc** Pointer to the error (location, value) pairs.

**init** Function pointer to Reed Solomon Decoder initialization routine.

**run** Function pointer to Reed Solomon Decoder computation routine.

**Description:**

Reed-Solomon Decoder structure.

## 6.5.2 Define Documentation

### 6.5.2.1 RS\_BLOCK\_K

**Definition:**

```
#define RS_BLOCK_K
```

**Description:**

Message size.

### 6.5.2.2 RS\_BLOCK\_N

**Definition:**

```
#define RS_BLOCK_N
```

**Description:**

Encoded block size.

### 6.5.2.3 RS\_BLOCK\_T

**Definition:**

```
#define RS_BLOCK_T
```

**Description:**

number of correctable errors

### 6.5.2.4 RS\_NROOTS

**Definition:**

```
#define RS_NROOTS
```

**Description:**

Number of code generator polynomial roots.

## 6.5.3 Typedef Documentation

### 6.5.3.1 REEDSOLOMON\_DECODER\_Handle

**Definition:**

```
typedef REEDSOLOMON_DECODER_Obj *REEDSOLOMON_DECODER_Handle
```

**Description:**

Handle to the Reed-Solomon Decoder structure.

### 6.5.3.2 REEDSOLOMON\_DECODER\_Obj

**Definition:**

```
typedef struct _REEDSOLOMON_DECODER_Obj_  
    REEDSOLOMON_DECODER_Obj
```

**Description:**

Reed-Solomon Decoder structure.

## 6.5.4 Function Documentation

### 6.5.4.1 REEDSOLOMON\_DECODER\_berlekampMassey

Error locator polynomial calculation (inversionless Berlekamp Massey Method).

**Prototype:**

```
void  
REEDSOLOMON_DECODER_berlekampMassey (REEDSOLOMON_DECODER_Handle  
hndRSDecoder)
```

**Parameters:**

← **hndRSDecoder** handle to the Reed Solomon Decoder object

**Note:**

Requires the lambda array to be even aligned

6.5.4.2 void REEDSOLOMON\_DECODER\_calcSyndrome  
(REEDSOLOMON\_DECODER\_Handle hndRSDecoder, int16\_t \* pData,  
int16\_t nBytes)

Syndrome calculation function (Horner's Method).

**Parameters:**

← **hndRSDecoder** handle to the Reed Solomon Decoder object  
← **pData** pointer to the data  
← **nBytes** number of bytes in the message block

**Note:**

Requires the syndrome array to be even aligned

6.5.4.3 void REEDSOLOMON\_DECODER\_chienForney  
(REEDSOLOMON\_DECODER\_Handle hndRSDecoder, int16\_t nBytes)

calculate error locations using Chien search and magnitude using Forney's algorithm

**Parameters:**

← **hndRSDecoder** handle to the Reed Solomon Decoder object  
← **nBytes** number of bytes in the message block

**Note:**

Requires the omega and error location arrays to be even aligned

6.5.4.4 void REEDSOLOMON\_DECODER\_initN255K239  
(REEDSOLOMON\_DECODER\_Handle hndRSDecoder, int16\_t \*  
pSyndrome, int16\_t \* pLambda, int16\_t \* pOmega, int16\_t \* pPackedAlpha,  
int16\_t \* pPackedBeta, int16\_t \* pRS\_expTable, int16\_t \* pRS\_logTable,  
ERROR\_LOCVAL\_Obj \* pErrorLoc)

Initializes the Reed Solomon Decoder object (n,k = 255, 239).

**Parameters:**

← **hndRSDecoder** handle to the Reed Solomon Decoder object  
← **pSyndrome** Pointer to the syndromes  
← **pLambda** Pointer to the error locator polynomial coefficients  
← **pOmega** Pointer to the error magnitude polynomial coefficients

- ← ***pPackedAlpha*** Pointer to the roots of the generator polynomial  $x + \alpha^i$
- ← ***pPackedBeta*** Pointer to the roots of the generator polynomial  $x + \beta^i$
- ← ***pRS\_expTable*** Pointer to the lookup table that converts index to decimal form
- ← ***pRS\_logTable*** Pointer to the lookup table that converts decimal to index form
- ← ***pErrorLoc*** Pointer to the error (location, value) pairs

**Note:**

Requires the data array to be even aligned

6.5.4.5 void REEDSOLOMON\_DECODER\_runN255K239  
([REEDSOLOMON\\_DECODER\\_Handle](#) *hndRSDecoder*, int16\_t \* *pData*,  
int16\_t *nBytes*)

Runs the Reed Solomon Decoder (n,k = 255, 239).

**Parameters:**

- ← ***hndRSDecoder*** handle to the Reed Solomon Decoder object
- ← ***pData*** pointer to the received message block
- ← ***nBytes*** number of bytes in the message block

## 6.6 De-Interleaver (VCU2)

### Data Structures

- `_DEINTERLEAVER_Obj_`

### Functions

- void `DEINTERLEAVER_run` (`DEINTERLEAVER_Handle` hndDEINTERLEAVER)

#### 6.6.1 Data Structure Documentation

##### 6.6.1.1 `_DEINTERLEAVER_Obj_`

**Definition:**

```
typedef struct
{
    uint16_t *pInBuffer;
    uint16_t *pOutBuffer;
    uint16_t *pSymbol;
    uint16_t n;
    uint16_t m;
    uint16_t b;
    uint16_t v;
    uint16_t a;
    uint16_t u;
    uint16_t n_i;
    uint16_t n_j;
    uint16_t m_i;
    uint16_t m_j;
    void (*init)(void *);
    void (*run)(void *);
}
_DEINTERLEAVER_Obj_
```

**Members:**

***pInBuffer*** Pointer to the input buffer.  
***pOutBuffer*** Pointer to the input buffer.  
***pSymbol*** Pointer to symbol storage.  
***n*** number of OFDM symbols in each interleaving block  
***m*** number of sub-carriers in each OFDM symbol  
***b*** beta  
***v*** mu  
***a*** alpha  
***u*** upsilon  
***n\_i*** Circular shift of the rows.  
***n\_j*** Circular shift of the rows.

***m\_i*** Circular shift of the columns.

***m\_j*** Circular shift of the columns.

***init*** Function pointer to DEINTERLEAVER initialization routine (NULL as of current release).

***run*** Function pointer to DEINTERLEAVER computation routine.

**Description:**

De-interleaver structure.

## 6.6.2 Function Documentation

### 6.6.2.1 DEINTERLEAVER\_run

Runs the DEINTERLEAVER routine.

**Prototype:**

```
void
DEINTERLEAVER_run (DEINTERLEAVER_Handle hndDEINTERLEAVER)
```

**Description:**

The de-interleaver equations are:

$$J = (j \times n_j + i \times n_i) \% n$$

$$I = (i \times m_i + J \times m_j) \% m$$

The interleaver equations are:

$$i = (a \times I - u \times J) \% m$$

$$j = (b \times J - v \times i) \% n$$

$$b = \beta_j$$

$$v = \mu_{ij} = \beta_j \times n_i$$

$$a = \alpha_i$$

$$u = v_{ij} = \alpha_i \times m_j$$

(i,j) - original bit position (I,J) - interleaved position

**Parameters:**

← ***hndDEINTERLEAVER*** handle to the DEINTERLEAVER object

## 7 Benchmarks

The benchmarks were obtained with the following compiler settings for the libraries:

VCU Type 0 (ISA\_C2800)

```
-v28 -ml -mt --vcu_support=vcu0 -g --verbose_diagnostics  
--diag_warning=225 --display_error_number --issue_remarks
```

VCU Type 2 (ISA\_C2800)

```
-v28 -ml -mt --vcu_support=vcu2 -g --verbose_diagnostics  
--diag_warning=225 --display_error_number --issue_remarks
```

The ISA\_C28FPU32 build configuration adds the `-float_support=fpu32` in addition to those specified above. The tables below list the performance metrics for all the library routines. These numbers were obtained by profiling the code in the examples directory

Module	Function	Cycles <sup>1</sup>
<b>CRC</b>	CRC_reset	11
	getCRC8_vcu	1.515 <sup>2</sup>
	getCRC32_vcu	1.515 <sup>2</sup>
	getCRC16P2_vcu	1.515 <sup>2</sup>
	getCRC16P1_vcu	1.515 <sup>2</sup>
<b>FFT</b>	cfft16_init	13
	cfft16_flip_re_img	223, N = 128
		414, N = 256
		798, N = 512
	cfft16_flip_re_img_conj	532, N = 64
		1043, N = 128
		2067, N = 256
	cfft16_pack_asm	1182, N = 64
		2271, N = 128
		4511, N = 256
	cfft16_brev	348, N = 64
		459, N = 128
		1655, N = 256
	cfft16_unpack_asm	1218, N = 128
		2339, N = 256
		4643, N = 512
<b>Viterbi</b>	cfft16_64p_calc	1402
	cfft16_128p_calc	3681
	cfft16_256p_calc	8135
	cnvDec_asm	5921 <sup>3</sup>
	cnvDecInit_asm	92
	cnvDecMetricRescale_asm	212

Table 7.1: Benchmark for the VCU Type 0 Library Routines

<sup>1</sup>include call, return and store (if required) instructions<sup>2</sup>average count per byte for a message size of 128 bytes<sup>3</sup>Viterbi decoder block size is 128 coded bits, mode: overlap decode



Module	Function	Cycles <sup>1</sup>
<b>CRC</b>	CRC_reset	11
	CRC_init8Bit	11
	CRC_run8Bit	1.437 <sup>2</sup>
	CRC_run8BitReflected	1.515 <sup>2</sup>
	CRC_init16Bit	11
	CRC_run16BitPoly1	1.437 <sup>2</sup>
	CRC_run16BitPoly2	1.437 <sup>2</sup>
	CRC_run16BitPoly1Reflected	1.515 <sup>2</sup>
	CRC_run16BitPoly2Reflected	1.515 <sup>2</sup>
	CRC_init24Bit	11
	CRC_run24Bit	1.437 <sup>2</sup>
	CRC_run24BitReflected	1.515 <sup>2</sup>
	CRC_init32Bit	11
	CRC_run32BitPoly1	1.414 <sup>2</sup>
	CRC_run32BitPoly2	1.414 <sup>2</sup>
	CRC_run32BitPoly1Reflected	1.492 <sup>2</sup>
	CRC_run32BitPoly2Reflected	1.492 <sup>2</sup>
<b>FFT</b>	CFFT_init32Pt	32
	CFFT_run32Pt	330 <sup>3</sup>
	ICFFT_run32Pt	333 <sup>3</sup>
	CFFT_init64Pt	32
	CFFT_run64Pt	608 <sup>3</sup>
	ICFFT_run64Pt	641 <sup>3</sup>
	CFFT_init128Pt	32
	CFFT_run128Pt	1494 <sup>3</sup>
	ICFFT_run128Pt	1495 <sup>3</sup>
	CFFT_init256Pt	32
	CFFT_run256Pt	2908 <sup>3</sup>
	ICFFT_run256Pt	3036 <sup>3</sup>
	CFFT_init512Pt	32
	CFFT_run512Pt	7011 <sup>3</sup>
	ICFFT_run512Pt	7012 <sup>3</sup>
	CFFT_init1024Pt	32
	CFFT_run1024Pt	13920 <sup>3</sup>
	ICFFT_run1024Pt	14435 <sup>3</sup>
	CFFT_conjugate	293, N = 64
		549, N = 128
		1061, N = 256
	CFFT_pack	733, N = 64
		1437, N = 128
		2845, N = 256
	CFFT_unpack	740, N = 128
		1443, N = 256
		2851, N = 512
<b>Viterbi</b>	VITERBI_DECODER_initK4CR12	15
	VITERBI_DECODER_runK4CR12	954 <sup>4</sup>
	VITERBI_DECODER_rescaleK4CR12	54
	VITERBI_DECODER_initK7CR12	15
	VITERBI_DECODER_runK7CR12	949

Continued on next page

Table 7.2 – continued from previous page

Module	Function	Cycles
	VITERBI_DECODER_rescaleK7CR12	285
<b>Reed-Solomon</b>	REEDSOLOMON_DECODER_initN255K239	78
	REEDSOLOMON_DECODER_runN255K239	10372
	REEDSOLOMON_DECODER_calcSyndrome	1426
	REEDSOLOMON_DECODER_berlekampMassey	1311
	REEDSOLOMON_DECODER_chienForney	7610
<b>Deinterleaver</b>	DEINTERLEAVER_run	773 <sup>5</sup>

Table 7.2: Benchmark for the VCU Type 2 Library Routines

<sup>1</sup>include call, return and store (if required) instructions<sup>2</sup>average count per byte for a message size of 128 bytes<sup>3</sup>VCU Type 2 FFT is more efficient when  $N_{stages} = 2k + 6$ ,  $k \in \{0, 1, 2\}$ <sup>4</sup>Viterbi decoder block size is 128 coded bits, mode: overlap decode<sup>5</sup>72 sub-carriers (G3 Powerline Communications FCC band)

Module	Function	Cycles <sup>1</sup>
<b>CRC</b>	genCRC8Table	47116 <sup>3</sup>
	genCRC16P1Table	57189 <sup>3</sup>
	genCRC16P2Table	57444 <sup>3</sup>
	genCRC32Table	51468 <sup>3</sup>
	getCRC8_cpu	24.234 <sup>2 3</sup>
	getCRC16P1_cpu	31.273 <sup>2 3</sup>
	getCRC16P2_cpu	31.273 <sup>2 3</sup>
	getCRC32_cpu	28.25 <sup>2 3</sup>
	CRC_bitReflect	30.968(max avg) <sup>3 2</sup>
	CRC_run8BitTableLookupC	35.453 <sup>3 2</sup>
	CRC_run32BitTableLookupC	39.375 <sup>3 2</sup>
	CRC_run32BitReflectedTableLookupC	40.351 <sup>3 2</sup>
	CRC_run24BitTableLookupC	40.398 <sup>3 2</sup>
	CRC_run24BitReflectedTableLookupC	40.375 <sup>3 2</sup>
	CRC_run16BitTableLookupC	31.406 <sup>3 2</sup>
	CRC_run16BitReflectedTableLookupC	31.406 <sup>3 2</sup>
<b>Viterbi</b>	VITERBI_ENCODER_init	114 <sup>3</sup>
	VITERBI_ENCODER_blockUnpack2Bits	16157 <sup>3 6</sup>
	VITERBI_ENCODER_quantizeBits	104496 <sup>3 6</sup>
	VITERBI_ENCODER_runK4CR12	49303 <sup>3 6</sup>
	VITERBI_ENCODER_runK7CR12	47194 <sup>3 6</sup>
<b>Reed-Solomon</b>	REEDSOLOMON_ENCODER_init	54 <sup>3 7</sup>
	REEDSOLOMON_ENCODER_run	412755 <sup>3 7</sup>
<b>Interleaver</b>	INTERLEAVER_findParams	132 <sup>4</sup>
	INTERLEAVER_run	3999 <sup>4</sup>

Table 7.3: Benchmark for the Library 'C' Routines

<sup>1</sup>include call, return and store (if required) instructions<sup>2</sup>average count per byte for a message size of 128 bytes<sup>3</sup>C routines compiled with default optimization<sup>4</sup>72 sub-carriers (G3 Powerline Communications FCC band)

## 8 Revision History

### **V2.10.00.00: Moderate Revision**

- Shortened, and eliminated when unnecessary, the context save/restores for all functions
- Changed the linker command file and example for the crc\_8 example to show how to run the crc on blocks larger than 65535 bytes
- Viterbi Decode - shortened the traceback loops (within the RPTB) from 6 instructions to 4
- Added De-interleaver assembly source code and example
- Added Interleaver 'C' source code
- Added VCU2 Real Inverse FFT source code and examples
- Corrected documentation for the RIFFT routines
- Eliminated global object definitions ('extern' qualifier) from the vcu2 header files
- Fixed bug in the rescale routine for Viterbi decode K7CR12 that was causing an overwrite
- Fixed bug with Inverse Berlekamp Massey routine, where size of the local frame was incorrect

### **V2.00.00.00: Initial Release**

- First release of the library to work with VCU types 0, 2
- Added legacy VCU0 routines

---

# 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	<a href="http://amplifier.ti.com">amplifier.ti.com</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>
DLP® Products	<a href="http://www.dlp.com">www.dlp.com</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>
Clocks and Timers	<a href="http://www.ti.com/clocks">www.ti.com/clocks</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>
RFID	<a href="http://www.ti-rfid.com">www.ti-rfid.com</a>
RF/IF and ZigBee® Solutions	<a href="http://www.ti.com/lprf">www.ti.com/lprf</a>

## Applications

Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Automotive	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
Broadband	<a href="http://www.ti.com/broadband">www.ti.com/broadband</a>
Digital Control	<a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a>
Medical	<a href="http://www.ti.com/medical">www.ti.com/medical</a>
Military	<a href="http://www.ti.com/military">www.ti.com/military</a>
Optical Networking	<a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a>
Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Telephony	<a href="http://www.ti.com/telephony">www.ti.com/telephony</a>
Video & Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>
Wireless	<a href="http://www.ti.com/wireless">www.ti.com/wireless</a>

---

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265  
Copyright © 2015, Texas Instruments Incorporated