VCU-II Software Library

USER'S GUIDE

VCU2-SW-LIB-UG-V2.00.00.00

Copyright

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

Revision Information

This is version V2.00.00.00 of this document, last updated on May 1, 2013.

Table of Contents

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. Complex Math

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_ccs5* 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 fro 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 v6.2.0or later**

3 Library Structure

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

```
C:\TI\controlSUITE\libs\dsp\VCU\VERSION
```

VERSION indicates the current revision of the VCU library. Figure. 3.1 shows the directory structure while the subsequent table 3.1 provides a description for each folder.

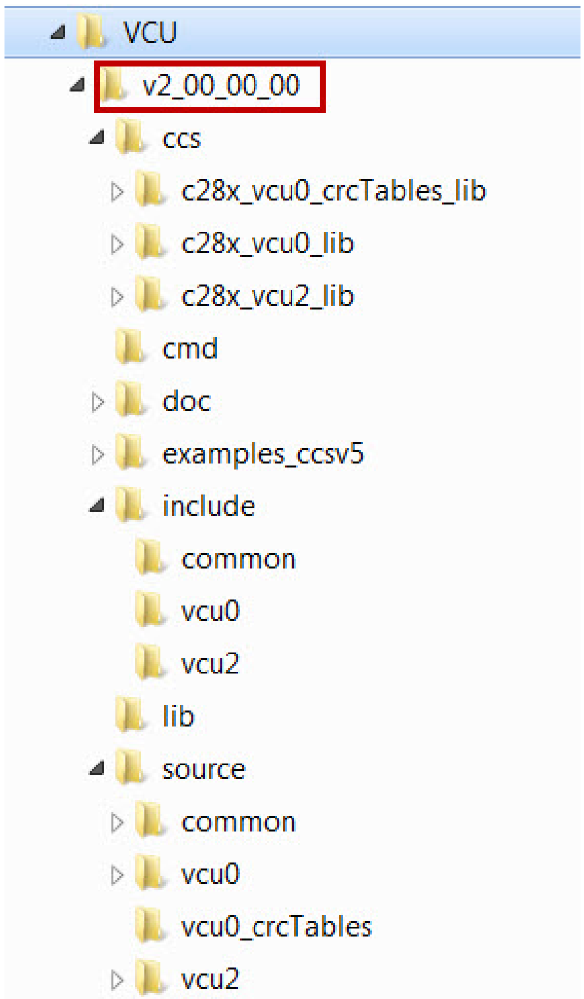

Figure 3.1: Directory Structure of the VCU Library

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 seperate 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/controlSUITE/libs/dsp/VCU/v2_00_00_00 For the rest of this document <base> will be omitted from the directory names
<base>/doc	Documentation for the current revision of the library including revision history
<base>/examples_ccsv5	Examples that illustrate the library functions. At the time of writing these examples were built for the F2837x device using the CCS5.5.0.00077 platform
<base>/include	Header files for the VCU library
<base>/lib	Pre-built VCU libraries
<base>/source	Source files and project for the library. Allows the user to reconfigure, modify and re-build the library to suit their particular needs

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 CCSv5(or later) you will be able to view and modify the source code for all routines and lookup tables (see Fig. 4.1)

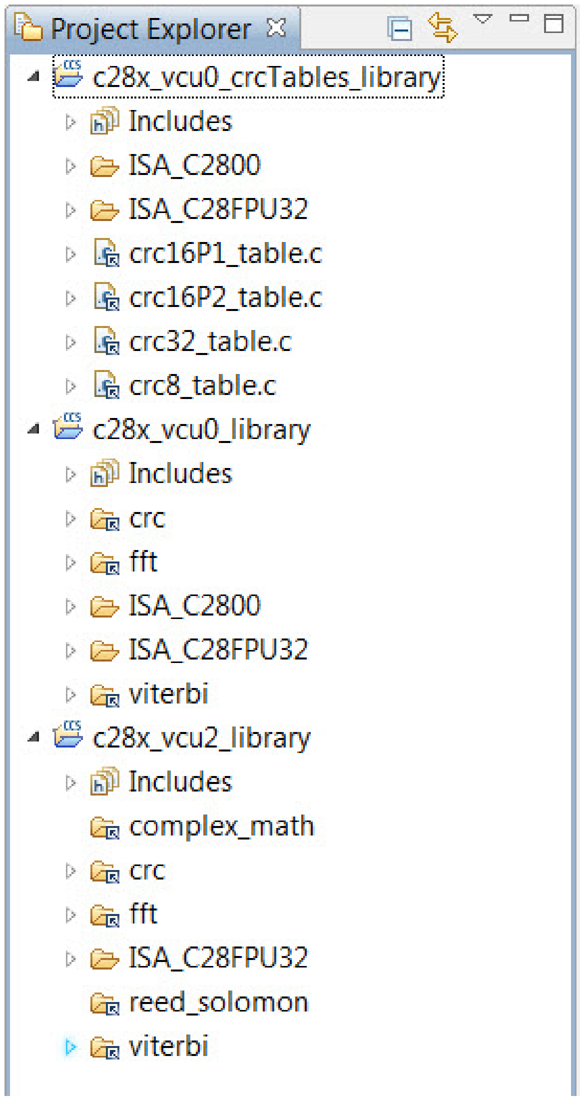

Figure 4.1: VCU Library Project View

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 config, 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

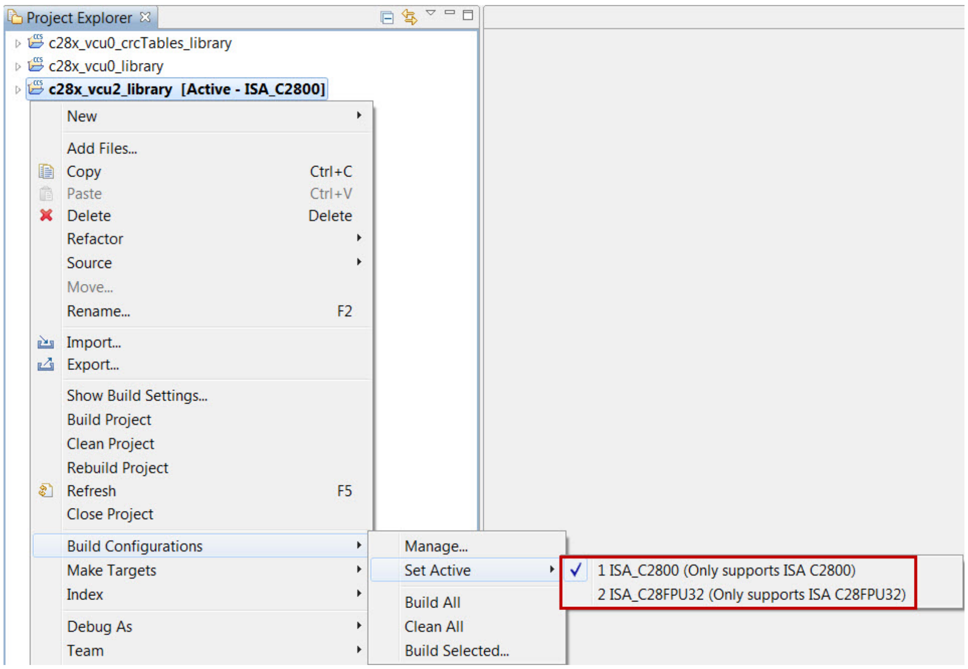

Figure 4.2: Library Build Configurations

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 controlsuite, this is usually the version folder.

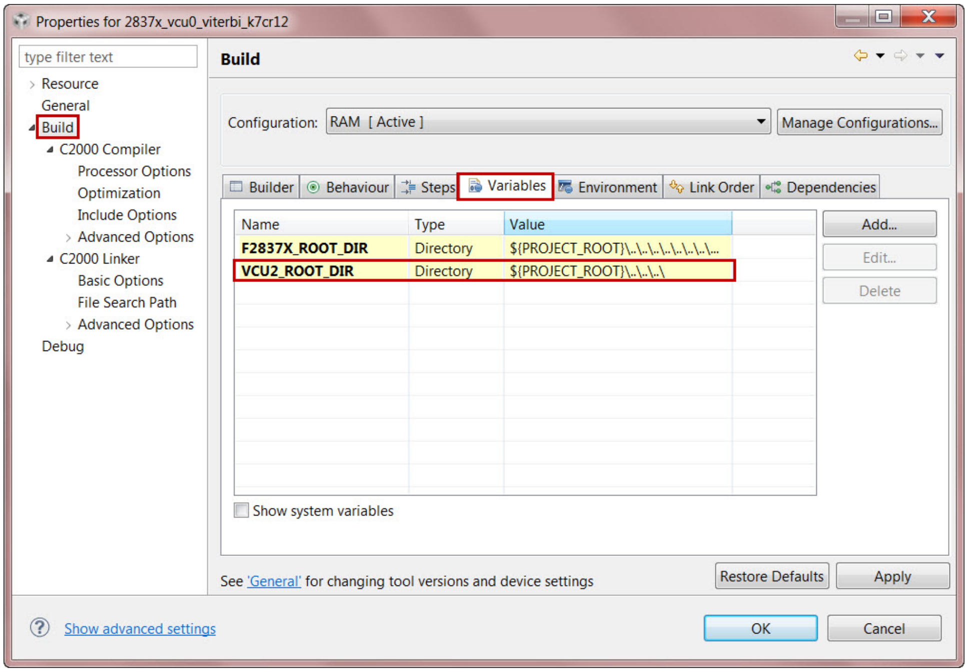

Figure 4.3: Creating a new build variable

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.

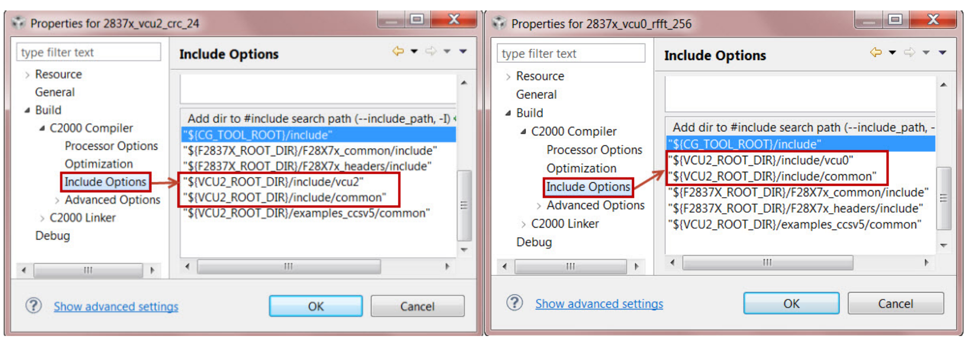

Figure 4.4: Adding the Include Search Path for the Library

2. Enable the **–vcu_support** option in the **Runtime Model Options** to either **vcu0** or **vcu2** depending on the library used (Fig. 4.5).

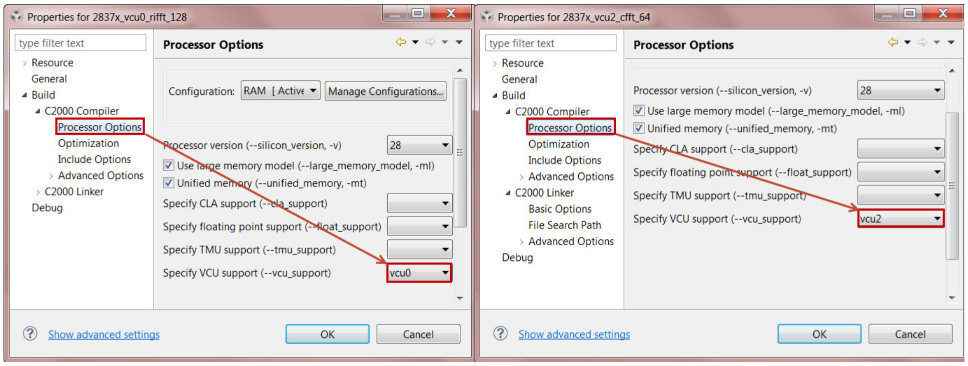

Figure 4.5: Turning on CLA support

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

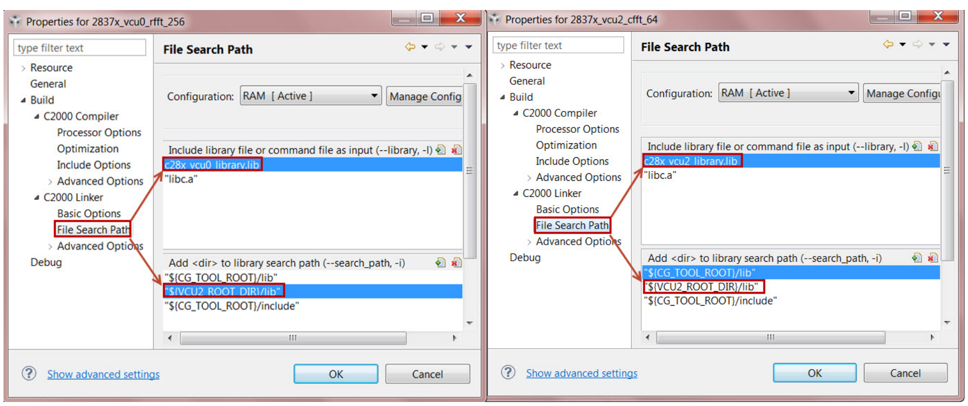

Figure 4.6: Adding the library and location to the file search path

5 Application Programming Interface (VCU0)

5.1 VCU0 Type Defintions

Data Structures

- cplx16

Enumerations

- parity_t

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 parity_t

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 in the message.

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

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 cifft16_pack_asm (cfft16_t *cfft16_handle_s)

Variables

- SINT16 BIT_REV_64_TBL[64]
- SINT16 CFFT16_TF[ ]

5.2.1 Data Structure Documentation

5.2.1.1 cfft16_t

Definition:

```
typedef struct
{
    int *ipcbptr;
    int *workptr;
    int *tfptr;
    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.

5.2.3 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-revereseed 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. For example, the 128 point complex FFT requires an input buffer of size 256 words, 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 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 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 cifft16_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 Variable Documentation

5.2.5.1 SINT16 BIT_REV_64_TBL[64]

Bit Reversal Lookup Table (deprecated)

5.2.5.2 SINT16 CFFT16_TF[ ]

Twiddle Factor Table.

5.3 Cyclic Redundancy Check (VCU0)

Defines

Description:

- 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, parity_t parity, uint16 rxLen)
- uint16 getCRC16P1_vcu (uint32 input_crc16_accum, uint16 *msg, parity_t parity, uint16 rxLen)
- uint16 getCRC16P2_cpu (uint16 input_crc16_accum, uint16 *msg, parity_t parity, uint16 rxLen)
- uint16 getCRC16P2_vcu (uint32 input_crc16_accum, uint16 *msg, parity_t parity, uint16 rxLen)
- uint32 getCRC32_cpu (uint32 input_crc32_accum, uint16 *msg, parity_t parity, uint16 rxLen)
- uint32 getCRC32_vcu (uint32 input_crc32_accum, uint16 *msg, parity_t parity, uint16 rxLen)
- uint16 getCRC8_cpu (uint16 input_crc8_accum, uint16 *msg, parity_t parity, uint16 rxLen)
- uint16 getCRC8_vcu (uint32 input_crc8_accum, uint16 *msg, parity_t 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*, parity_t *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 byte. The parity 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

5.3.2.7 getCRC16P1_vcu

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

Prototype:

```
uint16
getCRC16P1_vcu(uint32 input_crc16_accum,
               uint16 *msg,
               parity_t 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 byte. The parity 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

5.3.2.8 getCRC16P2_cpu

C- function to get the 16-bit CRC.

Prototype:

```
uint16
getCRC16P2_cpu(uint16 input_crc16_accum,
               uint16 *msg,
               parity_t 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 byte. The parity 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

5.3.2.9 getCRC16P2_vcu

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

Prototype:

```
uint16
getCRC16P2_vcu(uint32 input_crc16_accum,
               uint16 *msg,
               parity_t 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 byte. The parity 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

5.3.2.10 getCRC32_cpu

C- function to get the 32-bit CRC.

Prototype:

```
uint32
getCRC32_cpu(uint32 input_crc32_accum,
             uint16 *msg,
             parity_t 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 byte. The parity 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

5.3.2.11 getCRC32_vcu

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

Prototype:

```
uint32
getCRC32_vcu(uint32 input_crc32_accum,
             uint16 *msg,
             parity_t 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 byte. The parity 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

5.3.2.12 getCRC8_cpu

C- function to get the 8-bit CRC.

Prototype:

```
uint16
getCRC8_cpu(uint16 input_crc8_accum,
            uint16 *msg,
            parity_t 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 byte. The parity 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

5.3.2.13 getCRC8_vcu

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

Prototype:

```
uint16
getCRC8_vcu(uint32 input_crc8_accum,
            uint16 *msg,
            parity_t 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 byte. The parity 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 previos block giving nBits/2 bits
- 3: Update metrics and transition history, trace back through the current and previous blocks, decodes current and previos 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 Defintions

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

- CFFT_Obj CFFT
- 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;
    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(0x2937268)
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,

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

$$F_o(k) = \frac{F(k) - F(\frac{N}{2} - k)^*}{2} e^{\frac{j2\pi k}{N}}$$

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

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

Parameters:
$\leftarrow$ ***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 seqeunce 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:
$\leftarrow$ ***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. For example, the 1024 point complex FFT requires an input buffer of size 2048 words, 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
```

6.2.2.10 void CFFT_run128Pt (CFFT_Handle *hndCFFT*)

Runs the Complex FFT routine.

Parameters:
$\leftarrow$ ***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. For example, the 128 point complex FFT requires an input buffer of size 256 words, 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. For example, the 256 point complex FFT requires an input buffer of size 512 words, 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
```

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. For example, the 32 point complex FFT requires an input buffer of size 64 words, 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
```

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. For example, the 512 point complex FFT requires an input buffer of size 1024 words, 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
```

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. For example, the 64 point complex FFT requires an input buffer of size 128 words, 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
```

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. For example, the 1024 point complex FFT requires an input buffer of size 2048 words, 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 boundary as well.

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. For example, the 128 point complex FFT requires an input buffer of size 256 words, 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 boundary as well.

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. For example, the 256 point complex FFT requires an input buffer of size 512 words, 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 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:
$\leftarrow$ ***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. For example, the 32 point complex FFT requires an input buffer of size 64 words, 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 boundary as well.

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:
$\leftarrow$ ***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. For example, the 512 point complex FFT requires an input buffer of size 1024 words, 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 boundary as well.

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:

$\leftarrow$ ***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. For example, the 64 point complex FFT requires an input buffer of size 128 words, 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 boundary as well.

6.2.3 Variable Documentation

6.2.3.1 CFFT_Obj CFFT

CFFT Object.

6.2.3.2 const int16_t∗ vcu0_twiddleFactors

VCU0 twiddle factors.

6.2.3.3 const int16_t∗ vcu2_twiddleFactors

VCU2 twiddle factors.

6.3 Cyclic Redundancy Check (VCU2)

Data Structures

- _CRC_Obj_

Defines

- INIT_CRC16
- INIT_CRC24
- INIT_CRC32
- INIT_CRC8

Enumerations

- 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)

Variables

- CRC_Obj CRC

6.3.1 Data Structure Documentation

6.3.1.1 _CRC_Obj_

Definition:

```
typedef struct
{
    uint32_t seedValue;
    uint16_t nMsgBytes;
    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 the location, in a word, of the first byte for the CRC calculation
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 Parity_e

Description:
Parity: the location, in a word, of the first byte of the CRC calculation.

6.3.5 Function Documentation

6.3.5.1 uint32_t CRC_bitReflect (uint32_t *valToReverse*, int16_t *bitWidth*)

Bit-reverse a value.

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 void CRC_init16Bit (CRC_Handle *hndCRC*)

Initializes the CRC object.

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 void CRC_init24Bit (CRC_Handle *hndCRC*)

Initializes the CRC object.

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 void CRC_init32Bit (CRC_Handle *hndCRC*)

Initializes the CRC object.

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 void CRC_init8Bit (CRC_Handle *hndCRC*)

Initializes the CRC object.

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 uint16_t CRC_pow2 (uint16_t *power*)

power of 2

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 void CRC_reset (void)

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

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.

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

6.3.5.9 void CRC_run16BitPoly1Reflected (CRC_Handle *hndCRC*)

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

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

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

6.3.5.10 void CRC_run16BitPoly2 (CRC_Handle *hndCRC*)

Runs the CRC routine using polynomial 0x1021.

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.

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

6.3.5.11 void CRC_run16BitPoly2Reflected (CRC_Handle *hndCRC*)

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

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

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

6.3.5.12 void CRC_run16BitReflectedTableLookupC (CRC_Handle *hndCRC*)

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

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

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

See also:
http://www.ross.net/crc/download/crc_v3.txt

6.3.5.13 void CRC_run16BitTableLookupC (CRC_Handle *hndCRC*)

C table-lookup 16-bit CRC calculation.

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

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

See also:
http://www.ross.net/crc/download/crc_v3.txt

6.3.5.14 void CRC_run24Bit (CRC_Handle *hndCRC*)

Runs the CRC routine.

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.

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

6.3.5.15 void CRC_run24BitReflected (CRC_Handle *hndCRC*)

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

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

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

6.3.5.16 void CRC_run24BitReflectedTableLookupC (CRC_Handle *hndCRC*)

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

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

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

See also:
http://www.ross.net/crc/download/crc_v3.txt

6.3.5.17 void CRC_run24BitTableLookupC (CRC_Handle *hndCRC*)

C table-lookup 24-bit CRC calculation.

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

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

See also:
http://www.ross.net/crc/download/crc_v3.txt

6.3.5.18 void CRC_run32BitPoly1 (CRC_Handle *hndCRC*)

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

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.

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

6.3.5.19 void CRC_run32BitPoly1Reflected (CRC_Handle *hndCRC*)

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

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

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

6.3.5.20 void CRC_run32BitPoly2 (CRC_Handle *hndCRC*)

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

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.

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

6.3.5.21 void CRC_run32BitPoly2Reflected (CRC_Handle *hndCRC*)

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

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

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

6.3.5.22 void CRC_run32BitReflectedTableLookupC (CRC_Handle *hndCRC*)

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

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

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

See also:
http://www.ross.net/crc/download/crc_v3.txt

6.3.5.23 void CRC_run32BitTableLookupC (CRC_Handle *hndCRC*)

C table-lookup 32-bit CRC calculation.

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

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

See also:
http://www.ross.net/crc/download/crc_v3.txt

6.3.5.24 void CRC_run8Bit (CRC_Handle *hndCRC*)

Calculate the 8-bit CRC using polynomial 0x7.

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.

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

6.3.5.25 void CRC_run8BitReflected (CRC_Handle *hndCRC*)

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

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

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

6.3.5.26 void CRC_run8BitTableLookupC (CRC_Handle *hndCRC*)

C table-lookup 8-bit CRC calculation.

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

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

See also:
http://www.ross.net/crc/download/crc_v3.txt

6.3.6 Variable Documentation

6.3.6.1 CRC_Obj CRC

Object of the structure CRC_Obj.

6.4 Viterbi Decoding (VCU2)

Data Structures

- _VITERBI_DECODER_Obj_

Enumerations

- VITERBIMODE_e

Functions

- void VITERBI_DECODER_initK4CR12 (VITERBI_DECODER_Handle hndVITDecoder)
- void VITERBI_DECODER_initK7CR12 (VITERBI_DECODER_Handle hndVITDecoder)
- void VITERBI_DECODER_rescaleK4CR12 (VITERBI_DECODER_Handle hndVITDecoder)
- void VITERBI_DECODER_rescaleK7CR12 (VITERBI_DECODER_Handle hndVITDecoder)
- void VITERBI_DECODER_runK4CR12 (VITERBI_DECODER_Handle hndVITDecoder)
- void VITERBI_DECODER_runK7CR12 (VITERBI_DECODER_Handle hndVITDecoder)

Variables

- VITERBI_DECODER_Obj VITERBI_DECODER

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(0x28636a8)

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 metrcs 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 metrcs 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 required to store UBITS)

 Transition history(bits per stage)--->
              <-------64 bits----->
              +----+----+----+----+   ^
S1_p->hist_p->|    |    |    |    |   |
        ^     |    |    |    |    |   |
        |     |    |    |    |    |   |
        |     |    |    |    |    |   |
    4*UBITS   |    |    |    |    |   |
        |     |    |    |    |    |   |
        |     |    |    |    |    |   |
        |     |    |    |    |    |   |
        v     |    |    |    |    |   |
  S2_p->W1_p->|    |    |    |    | 128 stages
        ^     |    |    |    |    |   |
        |     |    |    |    |    |   |
        |     |    |    |    |    |   |
    4*UBITS   |    |    |    |    |   |
        |     |    |    |    |    |   |
        |     |    |    |    |    |   |
        |     |    |    |    |    |   |
        v     |    |    |    |    |   |
        W2_p->|    |    |    |    |   |
              +----+----+----+----+   v
```

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

6.4.4 Variable Documentation

6.4.4.1 VITERBI_DECODER

Definition:

```
VITERBI_DECODER_Obj VITERBI_DECODER
```

Description:

VITERBI Decoder Object.

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)

Variables

- REEDSOLOMON_DECODER_Obj REEDSOLOMON_DECODER

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*)

caculate 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

6.5.5 Variable Documentation

6.5.5.1 REEDSOLOMON_DECODER_Obj REEDSOLOMON_DECODER

Reed Solomon Decoder Object.

7 Benchmarks

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

VCU Type 0 (ISA_C2800)

Description:

```
-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. Table. 7.1 lists the performance metrics for all the library routines. These numbers were obtained by profiling the code in the examples directory

Library	Function	Cycles[1]
VCU Type 0	CRC_reset	11
	getCRC8_vcu	1.515 [2]
	getCRC32_vcu	1.515 [2]
	getCRC16P2_vcu	1.515 [2]
	getCRC16P1_vcu	1.515 [2]
	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
	cifft16_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
	cfft16_64p_calc	1402
	cfft16_128p_calc	3681
	cfft16_256p_calc	8135
	cnvDec_asm	5921 [4]
	cnvDecInit_asm	92
	cnvDecMetricRescale_asm	212
VCU Type 2	CRC_reset	11
	CRC_init8Bit	12
	CRC_run8Bit	1.476 [2]

Continued on next page

Table 7.1 – continued from previous page

Library	Function	Cycles
	CRC_run8BitReflected	1.554 [2]
	CRC_init16Bit	12
	CRC_run16BitPoly1	1.476 [2]
	CRC_run16BitPoly2	1.484 [2]
	CRC_run16BitPoly1Reflected	1.554 [2]
	CRC_run16BitPoly2Reflected	1.554 [2]
	CRC_init24Bit	12
	CRC_run24Bit	1.476[2]
	CRC_run24BitReflected	1.554[2]
	CRC_init32Bit	12
	CRC_run32BitPoly1	1.460 [2]
	CRC_run32BitPoly2	1.460 [2]
	CRC_run32BitPoly1Reflected	1.539 [2]
	CRC_run32BitPoly2Reflected	1.539 [2]
	CFFT_init32Pt	54
	CFFT_run32Pt	350 [5]
	ICFFT_run32Pt	354 [5]
	CFFT_init64Pt	54
	CFFT_run64Pt	629 [5]
	ICFFT_run64Pt	662 [5]
	CFFT_init128Pt	54
	CFFT_run128Pt	1515 [5]
	ICFFT_run128Pt	1516 [5]
	CFFT_init256Pt	53
	CFFT_run256Pt	2929 [5]
	ICFFT_run256Pt	3057 [5]
	CFFT_init512Pt	53
	CFFT_run512Pt	7032 [5]
	ICFFT_run512Pt	7033 [5]
	CFFT_init1024Pt	53
	CFFT_run1024Pt	13941 [5]
	ICFFT_run1024Pt	14456 [5]
	CFFT_unpack	759, N = 128
		1462, N = 256
		2871, N = 512
	VITERBI_DECODER_initK4CR12	43
	VITERBI_DECODER_runK4CR12	976
	VITERBI_DECODER_rescaleK4CR12	97
	VITERBI_DECODER_initK7CR12	43
	VITERBI_DECODER_runK7CR12	969
	VITERBI_DECODER_rescaleK7CR12	328
	REEDSOLOMON_DECODER_initN255K239	105
	REEDSOLOMON_DECODER_runN255K239	10824
	REEDSOLOMON_DECODER_calcSyndrome	1445
	REEDSOLOMON_DECODER_berlekampMassey	1464
	REEDSOLOMON_DECODER_chienForney	7870
Common	genCRC8Table	47116 [3]
	genCRC16P1Table	57189 [3]

Continued on next page

Table 7.1 – continued from previous page

Library	Function	Cycles
	genCRC16P2Table	57444 [3]
	genCRC32Table	51468 [3]
	getCRC8_cpu	24.234 [2] [3]
	getCRC16P1_cpu	31.273 [2] [3]
	getCRC16P2_cpu	31.273 [2] [3]
	getCRC32_cpu	28.25 [2] [3]
	CRC_bitReflect	30.968(max avg) [3] [2]
	CRC_run8BitTableLookupC	34.492 [3] [2]
	CRC_run32BitTableLookupC	41.5 [3] [2]
	CRC_run32BitReflectedTableLookupC	42.476 [3] [2]
	CRC_run24BitTableLookupC	42.5 [3] [2]
	CRC_run24BitReflectedTableLookupC	42.484 [3] [2]
	CRC_run16BitTableLookupC	34.492 [3] [2]
	CRC_run16BitReflectedTableLookupC	34.484 [3] [2]
	VITERBI_ENCODER_init	129 [3]
	VITERBI_ENCODER_blockUnpack2Bits	12700 [3] [6]
	VITERBI_ENCODER_quantizeBits	96298 [3] [6]
	VITERBI_ENCODER_runK4CR12	49398 [3] [6]
	VITERBI_ENCODER_runK7CR12	54455 [3] [6]
	REEDSOLOMON_ENCODER_init	56 [3] [7]
	REEDSOLOMON_ENCODER_run	442233 [3] [7]

Table 7.1: Benchmark for the VCU Library Routines

[1] include call, return and store (if required) instructions
[2] average count per byte for a message size of 128 bytes
[3] C routines compiled without optimization turned on
[4] Viterbi decoder block size is 128 coded bits, mode: overlap decode
[5] VCU Type 2 FFT is more efficient when $N_{stages} = 2k + 6, k \in \{0, 1, 2\}$
[6] Raw input data of 64 words(16-bit)
[7] 235 bytes in the message encoded to a block of 251 bytes

8 Revision History

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

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