C28x VCU Library

v1.10

Module User's Guide (SPRT606)

C28x Foundation Software

Texas Instruments

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components. In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of 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. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible or liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products:

www.ti.com/sc/docs/stdterms.htm

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

Table of Contents

Trademarks

TMS320 is the trademark of Texas Instruments Incorporated.

Code Composer Studio is a trademark of Texas Instruments Incorporated.

All other trademark mentioned herein is property of their respective companies.

1 Introduction

The Texas Instruments TMS320C28x Viterbi, Complex Math and CRC Unit (VCU) Library is a collection of highly optimized application functions written for the C28x + VCU. These functions enable the programmer to accelerate the performance of communications-based algorithms by up to a factor of 8X over C28x alone. This document provides a description of each function included within the library

2 Installing the Library

2.1 Directory Structure (Where the files are located)

The C28x VCU Library is partitioned into a well-defined directory structure. By default, the library and source code is installed into the **C:\TI\controlSUITE\libs\dsp\VCU\v110** directory. The following table describes the contents of the main directories used by the library:

Directory	Description
<base>	Base Install directory. By default this is **C:\TI\controlSUITE\libs\dsp\VCU\v110** For the rest of the document <base> will be omitted from the directory names
<base>\cmd	Linker command files that are specific to the library
<base>\doc	Documentation including the revision history from any previous release.
<base>\examples_ccsv4	Code Composer Studio V4 based examples
<base>\include	Header files that include the library function prototypes and structure definitions
<base>\lib	The built library.
<base>\source\C28x_VCU_LIB	Source files for the library. This also includes a Code Composer Studio project that can be used to re-build the library if required.

Table 1. C28x VCU Library Directory Structure

2.2 Library Build Options

The library was built using C28x codegen tools **v6.0.1** in CCS v4.2.2 with the following options:

-v28 -mt -ml -g --keep_unneeded_statics --vcu_support=vcu0

You can specify VCU support in the Runtime Model Options menu of the compiler's build property page (see red highlighted boxes in figure below)

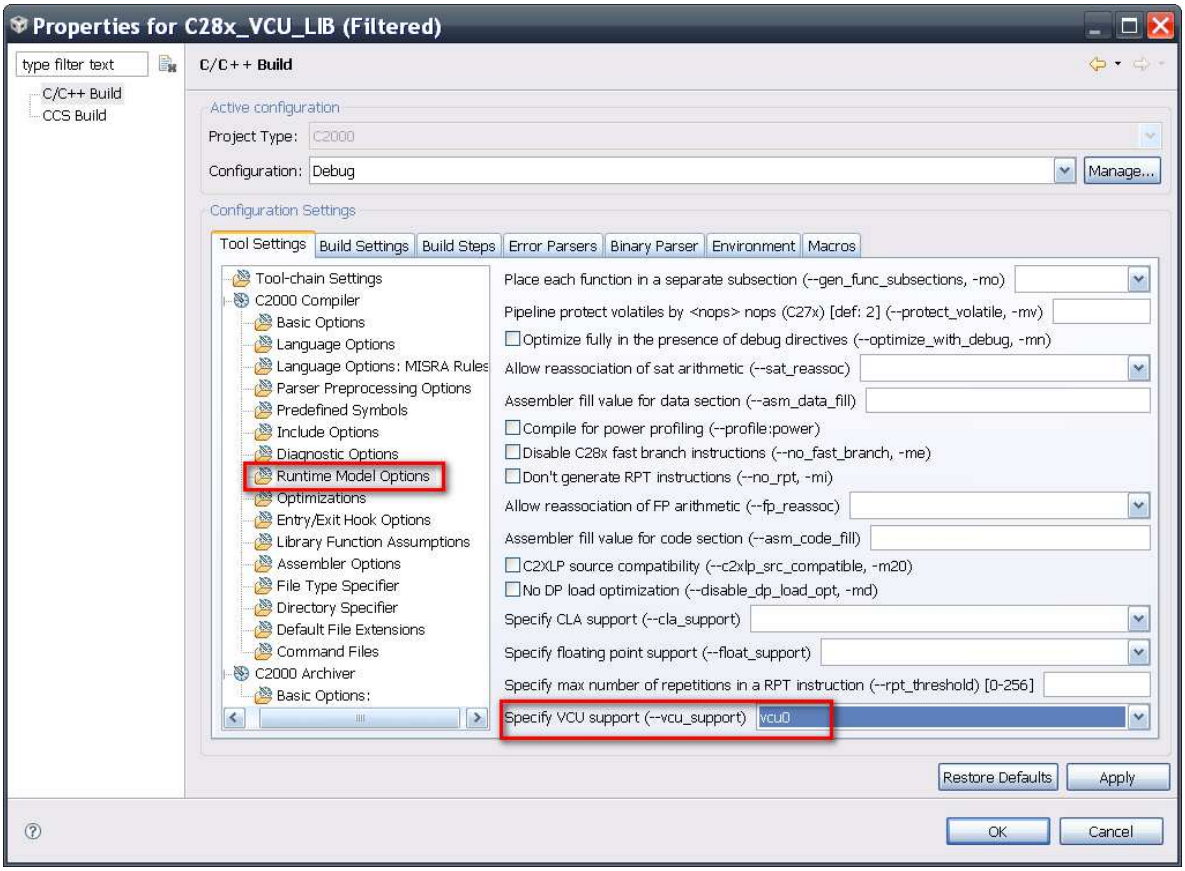

Figure 1. Enabling VCU Support

NOTE: Codegen v6.0.1 (and above) is available through the Software Updates option in the help menu. **Older versions of CCS (4.2.1 and older) will not display the vcu_support option.**

2.3 Header Files

The library header files are located in the **<base>\include** folder. Each Application module (FFT,CRC and Viterbi) has a separate header.

FFT - **<base>\include\fft.h**

CRC - **<base>\include\crc.h**

Viterbi - **<base>\include\viterbi.h**

A common header file **typedef.h** includes C28x data types and application-specific structures. The C28x data types include the following:

```
#ifndef DSP28_DATA_TYPES
#define DSP28_DATA_TYPES
typedef int                int16;
typedef long               int32;
typedef long long          int64;
#ifndef STD_
typedef unsigned int       Uint16;
typedef unsigned long      Uint32;
#endif //STD_
typedef unsigned long long Uint64;
typedef float              float32;
typedef long double        float64;
#endif //DSP28_DATA_TYPES
```

3 Function Summary

This release demonstrates the VCU's capability to accelerate several functions like the FFT butterfly, Viterbi decoding process and 8-bit CRC. A summary of the available library functions are listed in the table below.

Description	Prototype
FFT – Fast Fourier Transforms	
Initialization	`void cfft16_init(cfft16_t *);`
Bit Reverse	`void cfft16_brev(cfft16_t *);`
Table Bit Reverse	`void cfft16_tbl_brev(cfft16_t *);`
Flip Real-Imag	`void cfft16_flip_re_img(cfft16_t *);`
Flip Real-Imag + Conjugate	`void cfft16_flip_re_img_conj(cfft16_t *);`
Pack	`void cifft16_pack_asm(void *);`
Unpack	`void cfft16_unpack_asm(cfft16_t *);`
Calculation(**N** pt)	`void cfft16_Np_calc(cfft16_t *);`
CRC – Cyclic Redundancy Checks	

crc**X** calculation – C routine	`uintT getCRCX_cpu(uintT input_crcX_accum, uint16 *msg, parity_t parity, uint16 rxLen);`
	`if X =8,16P1,16P2 then T =16`
	`if X = 32 then T = 32`

crc**X** calculation – ASM routine	`uintT getCRCX_vcu(uint32 input_crcX_accum, uint16 *msg, parity_t parity, uint16 rxLen);`
	`if X =8,16P1,16P2 then T =16`
	`if X = 32 then T = 32`

crcX Table generation	`void genCRCXTable();` `X = 8,16P1,16P2,32`

VITERBI – Viterbi Decoding	
Initialization	`void cnvDecInit_asm(int nTranBits);`
Metric Rescale	`void cnvDecMetricRescale_asm();`
Convolutional Decode	`void cnvDec_asm(int nBits, int *in_p, int *out_p, int flag);`

Table 2. VCU Function Summary

4 Module Description

4.1 The Fast Fourier Transform

The library supports 64,128 and 256 point Complex FFTs,128, 256 and 512 point Real FFTs and 64, 128 and 256 point Inverse FFTs. They share a common data structure:

```
typedef struct {
 int *ipcbptr; //!< input buffer pointer
 int *workptr; //!< work buffer pointer
 int *tfptr;   //!< twiddle factor table pointer
 int size;     //!< Number of data points
 int nrstage;  //!< Number of FFT stages
 int step;     //!< Twiddle factor step size
 int *brevptr; //!< Bit Reversal Table Pointer
 void (*init)(void *); //!< Initialization routine
 void (*calc)(void *); //!< Calculation routine
 }cfft16_t;
```

NOTE: The 16 in **cfft16_t** is not be confused with the number of points in the FFT. It simply indicates the FFT operates on 16-bit data.

Member	Description	Comment
`ipcbptr`	Input Buffer Pointer	See [Memory Considerations]
`workptr`	Work buffer pointer	See [Memory Considerations]
`tfptr`	Twiddle factor table pointer	See [Lookup Tables]
`size`	Number of data points	Denoted as **N**
`nrstage`	Number of FFT stages	#stages = log2(**N**)
`step`	Twiddle factor search step	512/**N**
`brevptr`	Bit reversal table pointer	See [Lookup Tables]
`init`	Initialization routine pointer	See [Function Summary]
`calc`	Calculation routine pointer	See [Function Summary]

Table 3. FFT Data Structure

To understand how the VCU accelerates the FFT see the section titled [FFT Butterfly].

USEFUL NOTES (FFT & IFFT)

1. All FFT source code is placed under the folder **<base>\source\C28x_VCU_LIB\fft**
2. The user must include the header file **fft.h** ,under the **<base>\include** folder, in their project in order to call any of the FFT routines.
3. In the FFT *Function Description* section of the document the argument to the routines are listed as **cfft16_t ***, whereas in the code they are void pointers.
4. The header file, **fft.h**, defines a set of default initialization parameters for the cfft16_t structure for each N-point FFT. For example, to create a 128 point complex FFT structure you would initialize it as,

   ```
   cfft16_t fft = cfft16_128P_DEFAULTS;
   ```

   To create a 128 point real FFT you would initialize it as

   ```
   cfft16_t fft = rfft16_128P_DEFAULTS;
   ```

   The complete list of available initialization parameters are,

   ```
   cfft16_64P_DEFAULTS
   cfft16_64P_BREV_DEFAULTS
   cfft16_128P_DEFAULTS
   cfft16_256P_DEFAULTS
   rfft16_128P_DEFAULTS
   rfft16_256P_DEFAULTS
   rfft16_512P_DEFAULTS
   rifft16_64P_DEFAULTS
   rifft16_128P_DEFAULTS
   rifft16_256P_DEFAULTS
   ```

   Use the correct initialization parameter i.e. `cfft16_Np_DEFAULTS` for complex FFTs, `rfft16_Np_DEFAULTS` for real FFTs and `rifft16_Np_DEFAULTS` for inverse FFTs when declaring the fft structure (N = number of points in the FFT)

   NOTE: IFFTs make use of the same data structure as FFTs

5. When using the **complex** N-point FFT routine on an N point real data stream (e.g. N samples/words from the ADC) it is **up to the user to convert this data to complex** form (2N words) by adding in a zero imaginary term to each data point. For example, if we were to start with the real data set A = [1,3,5,6,8,2]; after conversion to complex we would end up with A =[0,1,0,3,0,5,0,6,0,8,0,2]. It is important to note here that the imaginary part is stored ahead of the real part so that the real part ends up being the high word of the 32-bit complex data. Each example has an assembly data file that shows you what the input data should look like. **You do not have to convert the data to complex form when using the real FFT**.
6. The input data buffer pointer (**ipcbptr**) and working buffer pointer (**workptr**) are used in **ping pong mode**. For example, in stage 1 ipcbptr points to the input data and wrkptr to the working buffer. In stage 2, they are swapped and the input data buffer becomes the new working buffer while the previous stage's working buffer becomes the new data set for the current stage. Since there are an odd number of stages, the final result of the FFT (pointed to by ipcbptr) ends up overwriting the original input data buffer.
7. Please note that in the examples the **input data gets overwritten** with the final output values, so re-running the test by issuing a soft reset will give incorrect values. The user must reload the example.
8. The real FFT examples make use of an **N/2**-pt complex FFT and its properties of symmetry to derive the **N**-pt real FFT
9. The inverse **N**-pt FFT makes use of the forward FFT. The data is first packed (see [3] for a detailed explanation on the process), then re-ordered in bit reversed format and fed through an **N-**pt FFT. Each output complex data point must then have its real and imaginary parts swapped in memory and conjugated to obtain the correct result.

4.1.1 Memory Considerations

All the FFT examples have the following linker command file: '**28069_CFFT16_RAM_lnk.cmd**' in the **<base>\cmd** folder**.** All data buffers are allocated to a section '**.shadow**' that the linker command file places in RAM Block L8. In all the examples **.shadow** is the only section to occupy RAML8 and therefore is automatically aligned to the start of the block (on an even address and on the same page). If it shares the block with other variables it would need to be aligned to 2N words. You can alter the command file as follows, (e.g. N =128, we need 256 words aligned)

```
.shadow : ALIGN = 256 > RAML8,  PAGE = 1
```

For any **N** point complex FFT we need **2N** words allocated to memory for both the input buffer and work buffer as the complex FFT expects complex input data. In any real world application, the input data would be an N-point stream of real data (16 bit data for the C28x). We convert the real data to complex data by adding an imaginary zero term to each data point prior to running the FFT, hence the 2N space requirement.

An **N** point real FFT uses an **N/2** point complex FFT to calculate the real FFT. It does this by assuming every odd data point as the imaginary part of a complex number(even data point being the real part). Thus we only need N words allocated to the input and working buffers respectively. Refer to [1] for efficient computation of the DFT of an N point real sequence.

4.1.2 Lookup Tables

The twiddle factors i.e. the W^k_N are stored in an array **cfft16_tf** in the file **cfft16_tf.asm**. There are 512 twiddle factors currently allocated to the **.econst** section. This number may increase in the future with larger point FFTs and as such may need to be allocated to a larger RAM block. For each stage of the FFT we access only a set number of twiddle factors from the table spaced at regular intervals, given by the **twiddle search step** factor. We access the table using the structure member **tfptr**.

The other lookup table is the bit reversal table. At the time of writing the **2806x_CFFT_64p_brev** example is the only one that uses bit reversal tables in the `cfft16_tbl_brev` function. Please note that the table is part of the file, **cfft_64_brev_data.asm,** under the example project, **2806x_CFFT_64p_brev,** and not part of the library itself.

4.1.3 FFT Butterfly

The VCU operates on complex data stored in fixed point (Q15, see [2] for details on IQmath notation) format so all the rounding and saturation effects come into play when operating on this data. The VCU registers VR0-VR8 are 32 bits wide and can hold complex data with the Real and Imaginary parts each occupying the high and low words of the register respectively.

For a single FFT butterfly we have two complex inputs, R_a:I_a and R_b:I_b (I – Imaginary, R – Real) and two complex outputs, R_c:I_c and R_d:I_d . In the assembly code for the FFT butterfly the following structure is used repeatedly.

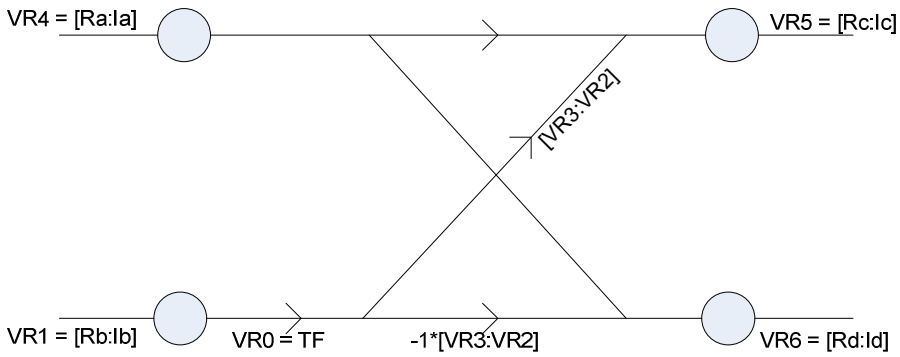

Figure 2. FFT Butterfly

To begin we load the two input complex data into VR4 and VR1 and the twiddle factor for the butterfly in VR0. The sequence of operations is as follows,

1. VR3 = Real(VR1 * VR0) & VR2 = Imaginary(VR1 * VR0) (2cycles)
2. Save off previous VR5 and VR6
3. VR6L = SAT16(((VR4L<<15)-VR2)>>16)

   VR6H = SAT16(((VR4H<<15)-VR3)>>16)
4. VR5L = SAT16(((VR4L<<15)+VR2)>>16)

   VR5H = SAT16(((VR4H<<15)+VR3)>>16)

It takes 5 cycles to do the butterfly with the VCU. The code snippet below shows the butterfly

```
VCMPY       VR3, VR2, VR1, VR0
||VMOV32    *XAR3++,VR5      ; Ic:Rc(1) = VR5
VMOV32      *XAR3++,VR6      ; Id:Rd(1) = VR6
VCDSUB16    VR6, VR4, VR3, VR2
VCDADD16    VR5, VR4, VR3, VR2
```

NOTE: Since we are operating in Q15 fixed point math, you can see that in steps 3 and 4 we shift the high and low word of VR4 (the second input) by 15, and then do the add/sub but then right shift back to 16. This means we are also dividing by 2 each stage of the FFT. This scaling factor is needed in fixed point calculations to prevent overflow from stage to stage.

4.1.4 Benchmark Information

The following table lists the cycle count for the Complex FFTs of sizes 64,128 and 256.

CFFT Size	Cycles	
	Bit Reversal	Calculation
64	343	1400
64(table based bit reversal)	480	1400
128	361	2990
256	1207	6483

Table 4. Complex FFT benchmark

The table below lists the cycle count for the Real FFTs.

RFFT Size	Cycles			
	Flip Real Imaginary	Bit Reversal	Calculation	Unpack
128	221	343	1400	1217
256	413	361	2990	2401
512	797	1207	6483	4769

Table 5. Real FFT benchmark

The following table lists the cycle count for Inverse FFTs.

IFFT Size	Cycles			
	Pack	Bit Reversal	Calculation	Flip Real Imaginary, then conjugate
64	1181	343	1400	530
128	2333	361	2990	1042
256	4637	1207	6483	2066

Table 6. Inverse FFT Benchmark

4.1.5 Function Description

cfft16_init	*Initialization function for the FFT*

This function will initialize the twiddle factor pointer (**tfptr**) of the FFT structure to the base address of the twiddle factor table (**cfft16_tf.asm**) in memory. This function is applicable to both complex and real FFTs.

Source File cfft16_utils.asm

Header File fft.h

Declaration `void cfft16_init(cfft16_t *)`

Arguments Handle to cfft_16 structure

Usage For a 128 point complex FFT,

```
cfft16_t fft = cfft16_128P_DEFAULTS;
fft.init(&fft);
```

Notes See [USEFUL NOTES], point 4

cfft16_brev	*Bit Reversing Utility*

This function will rearrange the input data pointed to by the fft structure member, **ipcbptr**, by bit reversing the index of each data point. This function is applicable to both complex and real FFTs.

Source File cfft16_utils.asm

Header File fft.h

Declaration `void cfft16_brev(cfft16_t *)`

Arguments Handle to cfft_16 structure

Usage For a 128 point complex FFT,

```
cfft16_t fft = cfft16_128P_DEFAULTS;
fft.init(&fft);
cfft16_brev(&fft);
```

Notes See [USEFUL NOTES], points 4,5

cfft16_tbl_brev	*Table based Bit Reversing Utility*

This function will rearrange the input data pointed to by the fft structure member, **ipcbptr**, in bit reversed index format. Instead of manipulating the indices directly, it uses a mapping table to determine in which order to rearrange the data points. At this point the **2806x_CFFT_64p_brev** example is the only one to use a bit reversal table.

Source File cfft16_utils.asm

Header File fft.h

Declaration `void cfft16_tbl_brev(cfft16_t *)`

Arguments Handle to cfft_16 structure

Usage For a 64 point bit reversed (table) complex FFT,

```
cfft16_t fft = cfft16_64P_BREV_DEFAULTS;
fft.init(&fft);
cfft16_tbl_brev(&fft);
```

Notes See [USEFUL NOTES], points 4,5

cfft16_flip_re_img	*Flip Real and Imaginary Terms of Complex Number*

This function will flip the real and imaginary parts of the complex input data pointed to by the fft structure member, **ipcbptr**. It must be called prior to running a real FFT on the data.

When using an N-point real FFT routine, the real input data (N words) is interpreted as an N/2 complex data array. For example, if we started off with the 10 point real data (16-bits) set,

A = [1,2,3,4,5,6,7,8,9,10]

The routine will interpret this as a 5 point complex data (32-bit) set with the high word as the real part and the low word as the imaginary part.

A = [(1,2),(3,4),(5,6),(7,8),(9,10)], (I,R)

We need to flip the high and low words to ensure that the real part of the complex data (32 bit long) is always at the high word. After calling the **cfft16_flip_re_img()** the data should look like this,

A = [(2,1),(4,3),(6,5),(8,7),(10,9)], (I,R)

Source File	cfft16_utils.asm
Header File	fft.h
Declaration	`void cfft16_flip_re_img(cfft16_t *)`
Arguments	Handle to cfft_16 structure

Usage	For a 128 point real FFT,
	`cfft16_t fft = rfft16_128P_DEFAULTS;`
	`fft.init(&fft);`
	`cfft16_flip_re_img(&fft);`

Notes	See [USEFUL NOTES], point 5

cfft16_flip_re_img_conj	*Flip Real and Imaginary Terms of Complex Number*

This function is identical to the previous one, with the exception that it flips the real and imaginary parts of the complex data point and negates the imaginary term to give its conjugate.

Source File	cfft16_utils.asm
Header File	fft.h
Declaration	`void cfft16_flip_re_img_conj(cfft16_t *)`
Arguments	Handle to cfft_16 structure

Usage	For a 128 point real FFT,
	`cfft16_t fft = rfft16_128P_DEFAULTS;`
	`fft.init(&fft);`
	`cfft16_flip_re_img_conj(&fft);`

Notes	See [USEFUL NOTES], point 5

cfft16_pack_asm	*Inverse FFT Packing Function*

The N-pt inverse Fourier Transform can be computed using a forward N-pt FFT. The input data, however, needs to be “packed” prior to bit-reversed ordering and then feeding it to the FFT.

Source File cfft16_utils.asm

Header File fft.h

Declaration `void cfft16_pack_asm(cfft16_t *)`

Alias `cfft16_pack(fft_hnd)`

Arguments Handle to cfft_16 structure

Usage For a 128 point IFFT,

```
cfft16_t fft = rifft16_128P_DEFAULTS;
fft.init(&fft);
cifft16_pack_asm(fft_hnd);
cfft16_brev(&fft);
fft.calc(&fft);
cfft16_flip_re_img_conj(fft_hnd);
```

Notes See [USEFUL NOTES], point 9 and reference [3]

cfft16_unpack_asm	*Derive real FFT from output of an N/2 point complex FFT*

The real FFT re-interprets the N real input data as N/2 complex data and then proceeds with the decimation in time algorithm. This approach does not give us the intended output directly. This function will apply the symmetry properties of the Fourier transform on the output to get the correct data.

Source File cfft16_utils.asm

Header File fft.h

Declaration `void cfft16_unpack_asm(cfft16_t *)`

Alias `cfft16_unpack(fft_hnd)`

Arguments Handle to cfft_16 structure

Usage For a 128 point real FFT,

```
cfft16_t fft = rfft16_128P_DEFAULTS;
fft.init(&fft);
```

```
cfft16_flip_re_img(&fft);
cfft16_brev(&fft);
fft.calc(&fft);
cfft16_unpack(&fft);
```

Notes See [USEFUL NOTES], point 5 and reference [3]

cfft16_Np_calc	*Complex FFT calculation*

This function calculates the N point (N = 64,128,256) complex FFT of the input data pointed to by the structure member, **ipcbptr**. It is also used to calculate the real FFT and the inverse FFT as well.

Source File cfft_N.asm (cfft_64.asm,cfft_128.asm, cfft_256.asm)

Header File fft.h

Declaration `void cfft16_Np_asm(cfft16_t *)`

Arguments Handle to cfft_16 structure

Usage For a 128 point complex FFT,

```
cfft16_t fft = cfft16_128P_DEFAULTS;
fft.init(&fft);
cfft16_brev(&fft);
fft.calc(&fft);
```

For a 128 point real FFT,

```
cfft16_t fft = rfft16_128P_DEFAULTS;
fft.init(&fft);
cfft16_flip_re_img(&fft);
cfft16_brev(&fft);
fft.calc(&fft);
cfft16_unpack_asm(&fft);
```

For a 128 point inverse FFT,

```
cfft16_t fft = rifft16_128P_DEFAULTS;
fft.init(&fft);
cifft16_pack_asm(&fft);
cfft16_brev(&fft);
fft.calc(&fft);
cfft16_flip_re_img_conj(&fft);
```

Notes See [USEFUL NOTES], points 5-9 and reference [1]

4.2 Cyclic Redundancy Check

The VCU can perform 8-bit, 16-bit and 32-bit CRC calculation of data stored in C28x ROM, RAMs and Flash to check their integrity during application runtime. The VCU is capable of calculating the CRC for a byte of data in a single cycle, thereby providing a significant speed boost over conventional software routines.

This library provides both C (C28x) and assembly routines to perform the CRC with the assembly routines using VCU instructions. The following polynomials are used for each of the CRCs,

CRC	Type	Polynomial
CRC8_PRIME	8-bit	0x07
CRC_ALT	16-bit	0x8005
CRC16_802_15_4	16-bit	0x1021
CRC32_PRIME	32-bit	0x04C11DB7

Table 7. CRC Polynomials

A new feature of codegen v6.0.1(and above) is the ability to calculate the CRC of a given section at link time. Please refer to the section, [Linker Generated CRC] for more details.

USEFUL NOTES (CRC)

1. All CRC source code is placed under the folder **<base>\source\C28x_VCU_LIB\crc**
2. All assembly and C routines pertaining to the CRC include the header file **crc.h** which is under the **<base>\include** folder. The user must include this header file in their project in order to call any of the CRC routines.
3. Each CRC routine takes as its first argument the accumulated crc. In the examples we set this value to 0 but it could also be the crc of a message fragment from a previous run. The user may choose to start off with a non zero value as this is quite useful in detecting errors in a long string of leading zeros.

4. The C and ASM routines to calculate the crc expect the data length argument in **bytes** not words. Special care needs to be taken when using the linker to generate the CRC. The linker saves the number of words operated on in the CRC_RECORD and the user is expected to convert this to bytes before using the library routines.

5. The CRC table for each polynomial has been compiled into a separate library: **C28x_VCU_LIB_Tables.lib**. The user needs to include this library in their project if they wish to use pre-built CRC lookup tables. You can add this library to your project by:

   i. Go to *Project Properties->C/C++ Build->C2000 Linker->File Search Path*

   ii. Include the library: "**C28x_VCU_LIB_Tables.lib**"

   iii. In the search path enter: "**${INSTALLROOT_TO_VCU_VERSION}/lib**"

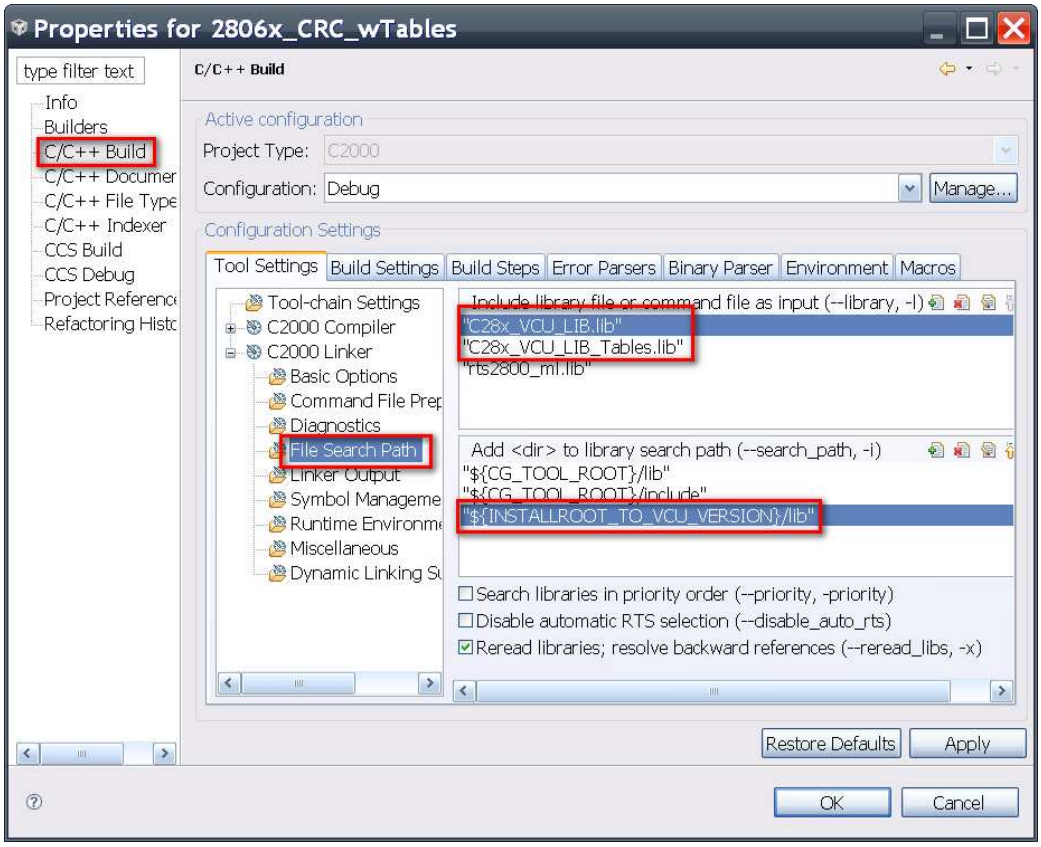

Figure 3. Adding a Library in project properties

6. To generate the lookup tables in software **DO NOT** include the library, **C28x_VCU_LIB_Tables.lib,** in your project. You must, however, declare the following variables and their memory placements in code:

```
#pragma DATA_SECTION(crc8_table,"CRC8_TABLE");
#pragma DATA_SECTION(crc16p1_table,"CRC16_TABLE");
#pragma DATA_SECTION(crc16p2_table,"CRC16_TABLE");
#pragma DATA_SECTION(crc32_table,"CRC32_TABLE");

uint16 crc8_table[256];
uint16 crc16p1_table[256];
uint16 crc16p2_table[256];
uint32 crc32_table[256];
```

4.2.1 Memory Considerations

The CRC example uses the following linker command file: '**28069_CRC_RAM_lnk.cmd**' in the **<base>\cmd** folder**.** The table below lists the three memory sections created

Section	Memory	Type(bits)	Space(words)
CRC8_TABLE	RAML5	16	256
CRC16_TABLE	RAML6	16	256*2
CRC32_TABLE	RAML7	32	512

Table 8. Memory Sections

The section CRC16_TABLE must have enough space for two tables namely `crc16p1_table` and `crc16p2_table`. It might be necessary to align these sections if they share the RAM block with other variables.

A new section `.TI.crctab` must be defined in order to be able to use the link time CRC calculation feature of the linker. This section will store all the CRC tables generated at link time.

4.2.2 Linker Generated CRC

The linker has the ability to calculate the CRC value for a given memory region at link time, and store that value in target memory such that it is accessible at boot/run time. This is a useful diagnostic tool as it allows an application program to check the integrity of a data region at run time.

The linker supports the four CRC algorithms mentioned earlier in the document. **The CRC may only be applied to initialized sections.**

To generate the CRC for a given section in memory we must use the `crc_table` command on that section in the linker command file. The linker will create a record of the type **CRC_RECORD** and store it in a designated memory. The record saves the CRC value and what polynomial was used in the calculation. The user must include the header file **crc_tbl.h**, which defines the structures **CRC_TABLE** and **CRC_RECORD**, in their application code to be able to access these records.

In the example project we calculate the 8-bit CRC for a section `CRC_CMD_TEST_VECTOR` with the following command:

```
CRC_CMD_TEST_VECTOR  : > RAML3,     PAGE = 1
crc_table(_CRCTestVector,algorithm = CRC8_PRIME)
```

What this does is tell the linker to allocate the section to RAML3 and calculate its 8-bit CRC using the polynomial specified by CRC8_PRIME and saving the record in a linker generated variable of the type **CRC_TABLE** called `_CRCTestVector.` This variable is automatically placed in the section `'.TI.crctab'.`

All linker generated variables including `_CRCTestVector` must be declared as extern in the user application to be able to access them.

```
extern CRC_TABLE CRCTestVector
```

Each CRC Table can have multiple records. To access these records we create a pointer of the type CRC_TABLE and a variable of the type CRC_RECORD

```
CRC_TABLE *psTable;
CRC_RECORD sRecord;
```

We then associate the pointer with our linker generated table and grab a record from it using our CRC_RECORD variable

```
psTable = &(CRCTestVector);
sRecord = psTable->recs[kk]; //kk = 0 - #records
```

Each record stores the polynomial type used in the CRC, page number of data, starting address and size (bytes, **NOT** words) of the memory region used in the calculation and finally the crc value itself.

The user can then proceed to use the C-routine, `getCRCX` or the assembly routine `getCRCX_vcu_asm` (X = 8,16P1,16P2,32) to get the CRC value at run-time.

Please refer to **LINKER_GENERATED_CRC_README.txt** [4] in the codegen folder for more details on the link-time CRC generation feature. The codegen, by default, is installed to the following folder on the local machine:

C:\Program Files\Texas Instruments\ccsv4\tools\compiler\C2000 Code Generation Tools 6.0.1

4.2.3 Benchmark Information

The following table shows the cycle count for the different n-bit CRCs operating on a 128 word data set.

NOTE: The example projects were compiled with no optimization, so it is possible to obtain better cycle counts with the C functions on higher optimization builds.

CRC (n-bit)	Cycles	
	C- Function	ASM-Function
CRC-8	6177	323
CRC-16P1	8228	323
CRC-16P2	8228	323
CRC-32	7202	323

Table 9. n-bit CRC Benchmark

4.2.4 Function Description

getCRCX_cpu	*C-routine to calculate CRC*

This function uses lookup tables, which the user can either pre-load or generate during execution, to calculate the CRC of a given array or section or memory. There are 4 variations of the function, each operating on a different data size and using a different generator polynomial, they are

Function	Operates on (bits)	POLY	Lookup table
getCRC8_cpu	8	0x07	crc8_table
getCRC16P1_cpu	16	0x8005	crc16p1_table
getCRC16P2_cpu	16	0x1021	crc16p2_table
getCRC32_cpu	32	0x04C11DB7	crc32_table

Table 10. Variations of getCRCX_cpu

Source File crcX.c (crc8.c, crc16.c and crc32.c)

Header File crc.h

Declaration

```
uintT getCRCX_cpu(uintT input_crcX_accum,
uint16 *msg, parity_t parity, uint16
rxLen);
X =8,16P1,16P2 ,T =16
X = 32, T = 32
```

Arguments

`Input_crcX_accum`	Accumulated CRC
`Msg`	Address of message buffer
`Parity`	Enumerated value (EVEN,ODD) of the first byte to determine if its on an even or odd address
`rxLen`	Number of **bytes** in the message

Usage For an 8-bit CRC,

```
//! Test Packet Length in number of words
#define CRC_TEST_PACKET_LEN     128
```

```
//! Test Packet Length in number of BYTES
#define CRC_TEST_PACKET_BYTES   256

UINT16 crc_c;
//INIT_CRC8 = 0, defined in crc.h
crc_c = getCRC8_cpu(INIT_CRC8, test_vector,
0,CRC_TEST_PACKET_BYTES);
```

Notes See [USEFUL NOTES (CRC)], points 3,4,5

getCRCX_vcu	*ASM-routine to calculate CRC*

This function uses the VCU instructions to calculate the CRC of a given array or section in memory. There are 4 variations of the function, each operating on a different data size and using a different generator polynomial, they are

Function	Operates on (bits)	POLY
getCRC8_vcu	8	0x07
getCRC16P1_vcu	16	0x8005
getCRC16P2_vcu	16	0x1021
getCRC32_vcu	32	0x04C11DB7

Table 11. Variations of getCRCX_vcu

Source File crcX_vcu.asm (crc8_vcu.asm, crc16_vcu.asm and crc32_vcu.asm)

Header File crc.h

Declaration

```
uintT getCRCX_vcu(uint32
input_crcX_accum, uint16 *msg, parity_t
parity, uint16 rxLen);
X =8,16P1,16P2 ,T =16
X = 32, T = 32
```

Arguments

`Input_crcX_accum`	Accumulated CRC
`Msg`	Address of message buffer
`Parity`	Enumerated value (EVEN,ODD) of the first byte to determine if its on an even or odd address

`rxLen`	Number of **bytes** in the message

Usage For an 8-bit CRC,

```
//! Test Packet Length in number of words
#define CRC_TEST_PACKET_LEN     128
//! Test Packet Length in number of BYTES
#define CRC_TEST_PACKET_BYTES   256

UINT16 crc_vcu;
//INIT_CRC8 = 0, defined in crc.h
crc_vcu = getCRC8_vcu(INIT_CRC8, test_vector,
0,CRC_TEST_PACKET_BYTES);
```

Notes See [USEFUL NOTES (CRC)], points 3,4

genCRCXTable	*C-routine to generate CRC lookup tables*

This function will generate the tables used for fast lookups in the different CRC algorithms. There are 4 variations of the function, each operating on a different data size and using a different generator polynomial, they are

Function	POLY	Table generated
genCRC8Table	0x07	crc8_table
genCRC16P1Table	0x8005	crc16p1_table
genCRC16P2Table	0x1021	crc16p2_table
genCRC32Table	0x04C11DB7	crc32_table

Table 12. Variations of genCRCXTable

Source File crcX.c (crc8.c, crc16.c and crc32.c)

Header File crc.h

Declaration
```
void genCRCXTable();
X = 8,16P1,16P2,32
```

Usage For an 8-bit CRC,

```
#pragma DATA_SECTION(crc8_table,"CRC8_TABLE");
#pragma DATA_SECTION(crc16p1_table,"CRC16_TABLE");
#pragma DATA_SECTION(crc16p2_table,"CRC16_TABLE");
#pragma DATA_SECTION(crc32_table,"CRC32_TABLE");

uint16 crc8_table[256];
uint16 crc16p1_table[256];
uint16 crc16p2_table[256];
uint32 crc32_table[256];
//! Generate the tables
genCRC8Table();
genCRC16P1Table();
genCRC16P2Table();
genCRC32Table();
```

Notes See [USEFUL NOTES (CRC)], points 3, 4, 6 and reference [5] on how to generate lookup tables

4.3 Viterbi Decoding

The VCU provides instructions to support fast software implementation of the Viterbi Add-Compare-Select and traceback operation in hardware. The library provides an optimized assembly implementation of a Viterbi Decoder of constraint length 7 and code rate of 1/2. For an introduction to the Viterbi decoding process see [6].

The following enumeration determines the mode of operation of the decoder:

```
typedef enum
{
CNV_DEC_MODE_DEC_ALL =0, //!< Decodes all output bits
CNV_DEC_MODE_OVLP_INIT = 1,//!< Use window overlap
       method, only metrics and transitions update
CNV_DEC_MODE_OVLP_DEC  = 2, //!< Use window overlap
       method,update transitions/metrics/trace
       curr blk+prv block, decodes prv blk
CNV_DEC_MODE_OVLP_LAST = 3 //!< last block in overlap
}vitMode_t;
```

Mode	Member	Description
M0	CNV_DEC_MODE_DEC_ALL	Decode all output bits
M1	CNV_DEC_MODE_OVLP_INIT	Use window overlap method. Update all state metrics and transitions. Do not decode any bits
M2	CNV_DEC_MODE_OVLP_DEC	Use window overlap method. Update all state metrics and transitions. Begin tracing the current and previous blocks while only decoding the previous block
M3	CNV_DEC_MODE_OVLP_LAST	Last block in the overlap method. Perfrom final update and decode all bits

Table 13. Viterbi Decoding Modes

For more information on decoding modes see [Decoding Process]

USEFUL NOTES (VITERBI)

1. All VITERBI source code is placed under the folder **<base>\source\C28x_VCU_LIB\viterbi**
2. All assembly routines pertaining to the VITERBI decoder include the header file **viterbi.h** which is under the **<base>\include** folder. The user must include this header file in their project in order to use the decoder.
3. Output data bits from the decoder are packed into words in the output buffer. The input, however, is not. Each incoming bit takes up a whole word. This is important in determining the memory space needed for the input buffer.
4. The input data bits are processed in a frame\block of length 256 **coded** bits (symbols from a convolutional encoder of code rate 1/2) or 128 **decoded** bits. The functions `cnvDecInit` and `cnvDec` accept the number of coded bits as their first argument.
5. The state metrics are based on Q15 fixed point math and can overflow. To prevent overflow they are rescaled every 4 frames\blocks.
6. The output buffers should be aligned to 32-bit boundaries. See [Memory Consideration] for more details on the need and alternatives of this requirement

4.3.1 Memory Consideration

The Viterbi example project uses the linker command file, **28069_VITERBI_RAM_lnk.cmd** which is located in the **<base>\cmd** folder. RAM blocks L0,L1 and L2 have been lumped together to be able to store the program code which is sizeable and tends to grow larger as the constraint length of the decoder increases beyond 7.

For the purpose of the example we have included **viterbi_data.asm** which defines the test input data and expected output data. This data is stored in the **.econst** section of memory in RAM blocks L3,L4 and L5.

The following table shows the different array variables in the example that need to have memory allocated in an uninitialized sector by the linker file

Variable	Section	Size (words)	Description
`tran_hist`	**.ebss**	512	Transition history array

dold	**.ebss**	64	Old State Metric
dnew	**.shadow**	64	New State Metric
trn_tmp	**.ebss**	256	Temp transition array
trn_s1_p	**.ebss**	2	Pointer to transition update start
trn_s2_p	**.ebss**	2	Pointer to transition update start
trn_w1_p	**.ebss**	2	Pointer to where trace overlap should go (wrap position)
trn_w2_p	**.ebss**	2	Pointer to where trace overlap should go (wrap position)

Table 14. Memory assignment for decoder variables

The old and new state metrics must be saved onto separate memory banks. In the library implementation the new state metric array, pointed to by dnew, is stored in the section **.shadow(RAML8)** while the old metric is stored in the **.ebss(RAML3_L5)** section

The output buffer needs to be aligned to 32-bits to allow for proper decoding during the traceback step. This can be specified in the linker command. In the example the output buffer is allocated to the section **"buffer_out"** at a 2 word(32-bit) boundary as follows:

```
buffer_out   : ALIGN = 2 > RAML8,      PAGE = 1
```

As an alternative to alignment the user can set the macro, **VIT_OUT_ODD**, to 1 in the library source file **Viterbi_K7CRhalf.asm** and recompile the library if the output buffers are not aligned to 32-bits. This results in slightly longer code but ensures the right result is obtained from the traceback step.

4.3.2 Decoding Process

The example project implements a Viterbi decoder of constraint length 7 and code rate 1/2. The sample input data and expected output are declared in **viterbi_data.asm**. A total of 4096 **coded** bits make up the receiver input. They have quantized values to simulate the soft decision process in most receivers. Since this is a 1/2 code rate decoder we expect 2048 output/**decoded** bits. The data stream is processed in blocks of 256 coded bits to allow simultaneous reception and decoding to take place. The variable `nBits` indicates the number of unprocessed coded bits at any stage and `vblk` the number of processed blocks. We can split the algorithm into 16 (4096/256) stages:

Stage 1

1. Run the routine in Mode 1 (`CNV_DEC_MODE_OVLP_INIT`). The first 256 bits will not be decoded at this stage but only be used to update all the state metrics and transition history.

```
cnvDec(CNV_DEC_BLK_CBITS,
       dataIn_p,
       dataOut_p,
       CNV_DEC_MODE_OVLP_INIT);
```

2. Update input pointer to point to the next block
3. Increment `vblk` by 1 and decrement `nBits` by the block size

Stage 2-15

1. For the intermediate 14 stages the routine is run in Mode 2 (`CNV_DEC_MODE_OVLP_DEC`) which will use the current block to update all the state metrics, transition updates and decode the previous block of data using the transition history that was updated in the previous stage.

```
cnvDec(CNV_DEC_BLK_CBITS,
       dataIn_p,
       dataOut_p,
       CNV_DEC_MODE_OVLP_DEC);
```

2. Update input pointer to point to the next block
3. Increment `vblk` by 1 and decrement `nBits` by the block size
4. For every 4th block (`vblk & 0x3`) we must rescale the metrics to prevent overflow between stages. We use the function, `cnvDecMetricRescale` to accomplish this.

Stage 16

1. Decode the final block (size <= 256 bits) using Mode 3 (CNV_DEC_MODE_OVLP_LAST).

4.3.3 The Viterbi Butterfly

The VCU operates on complex data stored in fixed point (Q15, see [2] for details on IQmath notation) format and so all the rounding and saturation effects come into play when operating on this data. The VCU registers VR0-VR8 are 32 bits wide and can hold complex data with the Real and Imaginary parts each occupying the high and low words of the register respectively.

For a single Viterbi butterfly the inputs and outputs are the state metrics. The current state's metrics i.e. SM(n) and SM(n+1) and future state's metrics, SM(m) and SM(m+1), are stored in high and low words of different VRn registers. In the assembly code for the Viterbi butterfly the following structure is used repeatedly, albeit, with different VRn registers on each run.

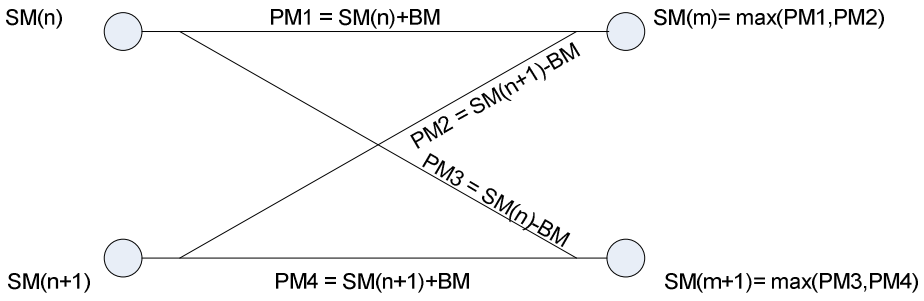

Figure 4. The Viterbi Butterfly

Assume that the current state metrics are in VR2, i.e. VR2 = [SM(n):SM(n+1)] and the branch metrics in VR0, VR0=BM=[Q(j)-Q(j+1):Q(j)+Q(j+1)] where Q(j) signifies the quantized received data. The butterfly has to complete the following sequence of operations to get the next/new state's metrics

1. Calculate the path metrics from the old state metrics and branch metrics

   VR3 = [PM1:PM2] =[VR2H+VR0L:VR2L-VR0L]

   VR4 = [PM3:PM4] =[VR2H-VR0L:VR2L+VR0L]

2. Select the value for the new state metrics

   VR5L = SM(m)= max(VR3L,VR3H)

   VR6L = SM(m+1)= max(VR4L,VR4H)

The VCU takes two cycles to complete a Viterbi butterfly. The following code snippet is an example of the butterfly

```
VITDLADDSUB VR4,VR3,VR2,VR0
VITLSEL VR6,VR5,VR4,VR3 || VMOV32 VR2, *XAR1++
```

4.3.4 Benchmark Information

The following table shows the cycle count for the viterbi decoder routine in modes 1,2 and 3 for a frame length of 256 coded bits. Since mode 3 is used on the last block of data its cycle count would depend on the number of remaining input bits after all blocks (of size 256) were processed in the previous stages

Viterbi Decoder Mode	Cycles
1	5176
2	5920
3	1197

Table 15. Viterbi Decoder K=7 CR=1/2 Benchmark

4.3.5 Function Description

cnvDecInit_asm	*ASM-routine to initialize the Viterbi Decoder*

This function will initialize the old state metric array to an initial value given by the macro, `CNV_DEC_METRIC_INIT` and also set the state transition pointers and wrap transition pointers to the correct locations in the transition history array. It expects the number of coded bits to be passed as an argument.

The following macro has been defined as an alias of this routine:

```
#define cnvDecInit(nBits) \
        cnvDecInit_asm(nBits)
```

Source File viterbi_K7CRhalf.asm

Header File viterbi.h

Declaration `void cnvDecInit_asm(int nTranBits);`

Alias `cnvDecInit(nBits)`

Arguments Number of coded bits in a frame\block

Usage In the viterbi example,

```
//Uncoded Bits
#define CNV_DEC_BLK_UBITS  64
//Coded bits
#define CNV_DEC_BLK_CBITS (CNV_DEC_BLK_UBITS << 1)
cnvDecInit((int)CNV_DEC_BLK_CBITS);
```

Notes See [USEFUL NOTES (VITERBI)], point 4

cnvDecMetricRescale_asm	*ASM-routine to rescale the Viterbi metrics*

This function will rescale the old branch metrics by finding the minimum metric and then subtracting it from the rest of the metrics. Since the VCU uses fixed point math, this repeated scaling ensures that there isn't any overflow between successive iterations of the decoding process

The following macro has been defined as an alias of this routine:

```
#define cnvDecMetricRescale() \
```

`cnvDecMetricRescale_asm()`

Source File viterbi_K7CRhalf.asm

Header File viterbi.h

Declaration `void cnvDecMetricRescale_asm();`

Alias `cnvDecMetricRescale()`

Usage In the viterbi example,

`cnvDecMetricRescale();`

Notes See [USEFUL NOTES (VITERBI)], point 5

cnvDec_asm	*ASM-routine to perform the decoding process*

This routine implements the decoder. It expects four arguments: the number of input bits to the decoder, the address of the input bits, the address of the output buffer to store the decoded bits and finally the decoder mode. The four modes of operation for the decoder are:

```
CNV_DEC_MODE_DEC_ALL
CNV_DEC_MODE_OVLP_INIT
CNV_DEC_MODE_OVLP_DEC
CNV_DEC_MODE_OVLP_LAST
```

The following macro has been defined as an alias of this routine:

```
#define cnvDec(nBits, in_p, out_p, flag) \
        cnvDec_asm(nBits, in_p, out_p, flag)
```

The operation of the decoder and how the modes affect the process is explained in the section [Decoding Process]

Source File viterbi_K7CRhalf.asm

Header File viterbi.h

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

Alias `cnvDec(nBits, in_p, out_p, flag)`

Arguments

`nBits`	Number of coded bits/frame
`in_p`	Address of input buffer

out_p	Address of output buffer
flag	Decoder mode

Usage In the viterbi example (assuming all variables were declared previously),

```
//Uncoded Bits
#define CNV_DEC_BLK_UBITS  64
//Coded bits
#define CNV_DEC_BLK_CBITS (CNV_DEC_BLK_UBITS << 1)
//Initialize the decoder
cnvDecInit((int)CNV_DEC_BLK_CBITS);
//Initialize input/output pointers
dataIn_p = VIT_quant_data;
dataOut_p = data_out;
/* Process the first blk in mode M1*/
nBits = DATA_BITS_LEN << 1;
cnvDec(CNV_DEC_BLK_CBITS,
       dataIn_p,
       dataOut_p,
       CNV_DEC_MODE_OVLP_INIT);
vblk = 1;
dataIn_p += CNV_DEC_BLK_CBITS;
nBits -= CNV_DEC_BLK_CBITS;
/*Proceed to decode subsequent blocks in mode M2 */
while (nBits >= CNV_DEC_BLK_CBITS){
   cnvDec(CNV_DEC_BLK_CBITS,
          dataIn_p,
          dataOut_p,
          CNV_DEC_MODE_OVLP_DEC);

   dataIn_p    += CNV_DEC_BLK_CBITS;
   dataOut_p   += (CNV_DEC_BLK_UBITS >> 4);
   nBits       -= CNV_DEC_BLK_CBITS;
   vblk++;
/* rescale every 4 blks  */
   if ((vblk & 0x3) == 0)
     cnvDecMetricRescale();
}
/* Decode final block in mode M3 */
cnvDec(nBits,
       dataIn_p,
       dataOut_p+((CNV_DEC_BLK_UBITS+nBits)>>4),
       CNV_DEC_MODE_OVLP_LAST);
```

Notes See [USEFUL NOTES (VITERBI)], points 3, 4

5 References

1. J. Proakis, D. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, sec 6.2.2
2. IQmath Wiki for the C28x, http://processors.wiki.ti.com/index.php/IQmath_Library_for_C28x
3. Engineering Productivity Tools Ltd, *The FFT Demystified*, http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM#Head521
4. LINKER_GENERATED_CRC_README.txt, C:\Program Files\Texas Instruments\ccsv4\tools\compiler\C2000 Code Generation Tools 6.0.1
5. *A Painless guide to CRC error detection algorithms v3*, http://www.repairfaq.org/filipg/LINK/F_crc_v3.html
6. C. Fleming, *A Tutorial on Convolutional Encoding with Viterbi Decoding,* http://home.netcom.com/~chip.f/viterbi/tutorial.html#specapps

6 Revision History

v1.00 – April 07, 2011

- First Release

V1.10 – October 01, 2011

- Revised benchmark information
- FFT
  - Added Context saves of registers VR0-6 in several routines
  - Newer, faster bit reversal routine
  - Added pack routine
- CRC
  - Branched out the CRC tables into a new static library: C28x_VCU_LIB_Tables

- Removed macro GEN_CRC_TABLES
- Added context save/restore of XAR0-3 to routines
- All VCU(assembly) routines accept a 32-bit initial CRC value
- Renamed getCRCx to getCRCx_cpu and getCRCx_vcu_asm to getCRCx_vcu

- Viterbi
  - Non 32-bit alignment of the output buffer requires additional steps to store the output
- Examples
  - Added 3 new examples for 64,128 and 256 pt IFFTs
  - Split CRC example into 2; one utilizing pre-loaded CRC lookup tables and the other generating the tables in software