Freescale Semiconductor
Application Note

Document Number: AN4945
Rev. 0, 4/2014

Accelerating the CFFT with Freescale's 32-bit DSC Instruction Set

With Processor-Expert–generated code modification method

by John L. Winters

Improvement in the performance of the CFFT on Freescale's 32-bit DSCs is gained by use of the bit-reverse indexing instruction. Using the method explained in Freescale document AN4837, “Porting Legacy DSC Applications to Freescale’s 32-bit DSC Family,” the test harness for the CFFT is:

1. Ported up to the 32-bit DSC
2. Modified to use the bit-reverse indexing instruction
3. Run through the test harness

Performance is measured before and after. The intended audience for this application note consists of those with interest in the DSC, regardless of Integrated Development Environment (IDE) experience.

Contents

1 Overview of project work

In a previous application note (AN4837, “Porting Legacy DSC Applications to Freescale’s 32-bit DSC Family”), the method for converting the Processor

Expert test harness software examples from the earlier DSC, DSP56858, to the present DSC, MC56F84xxx, was revealed. In that application note, a similar method was first used to convert the CW8.3 for DSC Processor Expert Example code for the Complex Fast Fourier Transform (CFFT) test harness from the 56858 device to the MC56F84789. That resulting project is provided as the starting point for this application note.

From that starting point project, this document will further develop the project by replacing one of the generated functions with one tailored to the new instruction set. The routine to be replaced is that which does bit-reverse indexing. Bit-reverse indexing differs from bit-reverse addressing: in bit-reverse indexing, the index is bit-reversed, incremented by one, and then bit-reversed again, instead of counting to the next integer value. To increment by "one," first the "one" is bit reversed, so that it occupies the most significant bit of the log2N bit right-justified index field; then it is added to the index with reverse carries in effect. This gives the bit reversed next index.

Much CPU time (up to 35 percent) is saved by having an instruction that can do this "reverse-carry add" in one instruction. Yet, the logic required to do this is very modest, being no different than a normal adder, with the only difference being the reversed frame of reference. Only one control bit is needed to turn reverse-carry on and off. The radix of the reverse-carry is determined by the value of the bit that is added to the index, and need not be adjusted as a separate step. Of course, this approach is limited, as is the example code already, to cases where N is an integral power of two.

For example, if $\log_2 N$ is 4, (N=16, or a 16 point CFFT), then the bit number set for the addend would be $\log_2 N$ -1, or bit 3. In hexidecimal this would be 0x8.

Adding this 0x8 to zero with reverse carry gives the sequence 0,8.

Adding this 0x8 to 8 with reverse carry gives 0,8,4, for the first three values in the sequence.

The complete sequence is 0,8,4,C,2,A,6,E,1,9,5,D,3,B,7,F.

This is exactly the kind of sequence needed by the CFFT algorithm.

Note that since bit-reversed pairs are swapped during the course of the function execution, care is needed to avoid re-swapping pairs that were already swapped. That safeguard is built into the function.

1.1 Materials needed

For this procedure, Freescale's CodeWarrior 8.3 for DSC (CW 8.3) is optional, required only if you wish to derive the project from the original source. The cost of the Special Edition of this integrated development environment is free to those who register at www.freescale.com.

Steps outlined below with the CW 8.3 for DSC IDE will not call for actual hardware. This is because the DP56858 is supported by the simulator included with CW 8.3. No debug pods are needed, and neither are boards.

Only a PC would be needed upon which CW 8.3 would be installed. Operating systems supported include Windows XP and Windows 7.

Also of course, CodeWarrior for MCU 10.5 (CW 10.5) is needed, and it is also available from www.freescale.com. In addition, the product TWR-56F8400 is needed to run the completed project. The TWR-56F8400 module is a small board that may be run standalone without the Tower System, or in the

Tower System. It will host the finished, ported application. At the time of writing, the simulator is not available for the MC56F84789 (device used on the TWR-56F8400).

1.2 Starting project – final project

It is advised for this project to start with the converted project. It will run the original bit-reverse code (which also runs on the 56800E cored devices) on the MC56F84789 under the CodeWarrior for MCU version 10.5 IDE. The zip file provided with this note contains one project, the final project.

I actually only supply the final project with the new bit reverse function. This is not a problem because, to fall back to the starting point, it is only necessary to delete dfr16bitrev.asm from the User Modules of CW10.5 and regenerate the code using Processor Expert. The old less-efficient module will be back, installed in the generated code directory, from which it is missing in the final project, as delivered in the zip file of associated software for this application note.

It is the purpose of this document to show how the final project is derived from the starting point. The final project saves cycles with the new bit-reverse function and measures its own cycle count in real time.

This will equip the reader to tackle cycle count reduction projects using the advanced instructions of this 32-bit DSC. To get started, get all the required materials and examine the bit-reverse functions, old and new.

To go to the final solution, just unzip the file again and import it.

In fact, if you rename one of the imported projects, you can have both projects in your work-area at the same time!

2 Modification of the bit-reverse assembly code

The code which was used to perform a bit-reverse on the legacy DSC devices with the V2 core is shown in section 2.1, with some discussion. The same code, revised to take advantage of the reverse-carry add instruction in the new 32-bit DSC core, is shown in section 2.2 with some discussion.

The modifications were fairly simple. In fact, the modification *was* a simplification of a rather small routine. Small routines are typical for DSP code, since not many instruction cycles are afforded per data point.

But how is the assembly language code generated by Processor Expert to be modified? It is really quite simple. Remove the module from the generated code folder and place it in the same folder as your main program. The IDE then treats it as if you own it. Just take care not to cause Processor Expert to regenerate the code. But don't worry – if it does regenerate it, you will only need to delete the file from the generated code directory.

For this project, we will be working with, or looking at, the file named dfr16bitrev.asm.

2.1 Original legacy V2 bit-reverse code

```
;*****************************************************************
;
; (c) Freescale Semiconductor
```

```
; 2004 All Rights Reserved
;
;
;*******************************************************************
;
; File Name:  dfr16bitrev.asm
;
; Description: Assembly module for Bit Reverse
;
; Modules
;  Included: Fdfr16Cbitrev_
;
; Author(s):  Sandeep S
;        Alwin Anbu.D
;
; Date:     3 Dec 2001
;
;*******************************************************************

    SECTION rtlib

  include "portasm.h"

  GLOBAL Fdfr16Cbitrev_

;*******************************************************************
;
; Module Name: Fdfr16Cbitrev_
;
; Description: Bit Reverses the Input Array
;
; Functions
;   Called: None
;
; Calling
; Requirements: 1. r2 - Pointer to Input Buffer.
;        2. r3 - Pointer to Output Buffer.
;        3. y0 - Length of the input/output buffer
;
; C Callable:  Yes
;
; Reentrant:  Yes
;
; Globals:   None
;
; Statics:   None
;
; Registers
;   Changed: All except r1 and r5
;
; DO loops:   1
;
; REP loops:  None
;
; Environment: MetroWerks on PC
;
; Special
```

```
;   Issues:  1.r2 and r3 MUST have even boundaries
;
;******************************Change History*************************
;
;  DD/MM/YY    Code Ver   Description   Author(s)
;  --------    --------   -----------   ------
;  18/01/2001   0.1     Module created  Sandeep S
;  18/01/2001   1.0     Baselined    Sandeep S
;  30/11/2001   1.1     Modified     Alwin Anbu.D
;
;*********************************************************************

Fdfr16Cbitrev_

  adda   #2,sp
  move.l  c2,x:(sp)+
  move.l  c10,x:(sp)+
  move.l  d2,x:(sp)+
  move.l  d10,x:(sp)

  clr.w   d               ; d is the normal index
  move.w  #0,x0             ; x0 is the bit reversed index
  tfra   r2,r0              ; r0 points to input
  tfra   r3,r4              ; r4 points to output
  move.w  y0,c
  asr    c                ; a1=(No.of points)/2

  dec.w   y0                ; y0=n-1

  if CODEWARRIOR_WORKAROUND==1
  do     y0,>>End_Do
  else
  do     y0,End_Do
  endif

  move.w  x:(r0)+,a0           ; Move real part to a0
  move.w  x:(r0)+,a1           ; Move imaginary part to a1
  cmp.w   x0,d              ; Check if d < x0
  blt    elsepart            ; if yes,bit reversal
                      ; already done.Jump to elsepart
  moveu.w  d,n              ; Move bit reversed index to n
  asla   n
  move.w  x:(r2+n),b0        ; Move real parts at bit
                     ; reversed locations to
                     ; normal location
  adda   #1,n
  move.w  x:(r2+n),b1          ; Move imaginary parts at bit
                     ; reversed locations to
                     ; normal location
  move.w  a1,x:(r4+n)
  adda   #-1,n
  move.w  a0,x:(r4+n)

  move.w  b0,x:(r3)+          ; Move imaginary part
  move.w  b1,x:(r3)-          ; Move imaginary part at bit
                     ; reversed location to
                      ; normal location
```

```
elsepart

  move.w  c1,y0            ; y0=N/2
  cmp    y0,d             ; Check if d < N/2 .Update r3
  move.w  x:(r3)+,y1
  blt    skip_change         ; If yes,skip change

chk_again            ; this loop is quite costly and is not needed with the V3 instruction set.
  sub    y0,d             ; d=d-y0
  asr    y0              ; y0=y0/2
  cmp.w   y0,d             ; Check if d>=y0
  bge    chk_again          ; If yes,check again

skip_change

  add    y0,d             ; Update r3
  move.w  x:(r3)+,y1
  inc.w   x0              ; Increment normal index
End_Do

  move.w  x:(r0)+,a0          ; Move last pair to a
  move.w  x:(r0)-,a1
  move.w  a0,x:(r3)+          ; Move a to last output location
  move.w  a1,x:(r3)-
  move.l  x:(sp)-,d
  move.l  x:(sp)-,d2
  move.l  x:(sp)-,c
  move.l  x:(sp)-,c2

  rts

  ENDSEC

;************************* End of file ********************************
```

2.2 Bit-reverse code optimized for V3 core

The loop, colored orange above (or a lighter shade of black), represents the bulk of the instructions that may be targeted for reduction in the new coding scheme. There is no need to calculate the bit reversal in a loop below, since one instruction does it. See the blue comments below.

```
;*********************************************************************
;
; (c) Freescale Semiconductor
; 2013 All Rights Reserved
;
;
;*********************************************************************
;
; File Name:  dfr16bitrev.asm
;
; Description: Assembly module for Bit Reverse of Complex number vector
;        Ported from 858 PEx test projects using CW10.2 project converter
;
; Modules
;  Included: Fdfr16Cbitrev_
;
```

```
; Target Processor
;  HawkV3 family or greater (Nevis2, Anguilla Silver..)
;  Will fail to function on earlier DSC parts!
;
; IDE
;  CodeWarrior for MCU 10.4 with test harness
;
; Author(s):  John L. Winters
;
; Date:     01 Aug 2013
;
;*********************************************************************

    SECTION user

  include "portasm.h"

  GLOBAL Fdfr16Cbitrev_

;*********************************************************************
;
; Module Name: Fdfr16Cbitrev_
;
; Description: Bit Reverses the Input Array
;
; Functions
;   Called: None
;
; Calling
; Requirements: 1. r2 - Pointer to Input Buffer of complex 16 bit entries
;       2. r3 - Pointer to Output Buffer of complex 16 bit entries, may be same as input buffer
;        3. y0 - Length of the input/output buffer. Input and output buffers are same length.
;
; C Callable:  Yes
;
; Reentrant:  Yes
;
; Globals:   None
;
; Statics:   None
;
; Registers
;   Changed: All except r1 and r5
;
; DO loops:   1
;
; REP loops:  None
;
; Environment: MetroWerks on PC
;
; Special
;   Issues:  1. r2 and r3 MUST have even boundaries
;        2. core prior to V3 did not have the reverse carry add feature
;         so code must detect if V3 instructions are present prior using it.
;         if the core is not V3 or newer, the orginal code should assemble.
;         Even with the reverse carry add, it is still required to use the
;         bit reverse routine. It is just faster with V3 instructions.
```

```
;        3. This routine sets the M01 register to 0x4000 which affects how addressing with R1
functions.
;         Any ISR interrupting this routine must consider M01 treatment.
;         It is set back to all ones on return.
;
;******************************Change History************************
;
;  DD/MM/YY   Code Ver   Description   Author(s)
;  --------   --------   -----------   ------
;  18/01/2001  0.1     Module created  Sandeep S
;  18/01/2001  1.0     Baselined    Sandeep S
;  30/11/2001  1.1     Modified     Alwin Anbu.D
;  08/01/2013  2.0     for V3 core   John L. Winters
;
;********************************************************************
         SUBROUTINE "Fdfr16Cbitrev_",Fdfr16Cbitrev_,Fdfr16Cbitrev_END-Fdfr16Cbitrev_
Fdfr16Cbitrev_:              ; tag with colon needed for debugger information generation, as
well as above SUBROUTINE statement
  adda   #2,sp            ; step past call information in stack
  move.l  R1,x:(sp)+          ; push r1
  move.l  c2,x:(sp)+          ; puch c2
  move.l  c10,x:(sp)+          ; push c10
  move.l  d2,x:(sp)+          ; push d2
  move.l  d10,x:(sp)          ; push d10
  move.w  #$4000,x0  ; this value, when placed in M01, will activate R0 for bit reversal mode
  moveu.w  x0,m01            ; set the m01 register to reverse carry for R0 only addressing
  move.w  #0,r0   ; for caluclation of the bit reversed index, zero starting position
  clr.w   d        ; d1 is the bit reversed index, zero initially. Will be copied from r0.
  move.w  #0,y1            ; y1 is the straight index, also zero initially
  tfra   r2,r1            ; r2 and r1 point to input
  tfra   r3,r4            ; r3 and r4 point to output
  move.w  y0,c            ; C1 number of complex input numbers
  asr    c              ; C1= number of complex numbers/2
; which is the addend into the reverse carry function to generate the bit reversed counter
  dec.w   y0             ; y0=n-1
  if CODEWARRIOR_WORKAROUND==1     ; see definition of this tag for explanation
  do    y0,>>End_Do         ;This loop is executed once for each of the complex numbers in the
input
;(or in the output).
  else
  do    y0,End_Do
  endif
  move.w  x:(r1)+,a0       ; Move real part of the input complex number to a0
  move.w  x:(r1)+,a1          ; Move imaginary part of the input complex number to a1
  cmp.w   y1,d            ; Check if d < y1, (check if the index is less than its bit reverse
that is to say)
  blt    elsepart          ; if yes,bit reversal already done, so jump to elsepart.
  moveu.w  d,n             ; Move bit reversed index to n
  asla   n             ; n is multiplied by two since complex numbers have two word entries
  move.w  x:(r2+n),b0         ; Move real part at bit reversed locations to: b0
  adda   #1,n            ; address the imaginary part at the bit reversed location
  move.w  x:(r2+n),b1         ; Move imaginary part at bit reversed location to b1
  move.w  a1,x:(r4+n)         ; store imaginary part of the input complex number in bit reversed
location (n always positive)
  adda   #-1,n            ; toggle down to the real part of the bit reversed loaction
  move.w  a0,x:(r4+n)         ; store the real part of the input complex number in the bit
reversed location (n always positive)
```

```
  move.w  b0,x:(r3)+         ; Move real part and
  move.w  b1,x:(r3)-         ; imaginary part at bit reversed location to normal location

; Above section moves the numbers to effect the bit reversal
elsepart   ; bit reversal already done (used bit reverse indices are always greater than the
index to avoid re-reversing!)
; this is where the next index and next bit reversed index are generated
; this code below adds one to the bit reversed index using the reverse carry feature of the V3
Hawk core.
; increment the bit reversed counter and the straight counter
  moveu.w  c,n             ; this is the number of complex numbers divided by two,
; or the msb of the complex number index needed to bit-reverse count-up
  move.w  x:(r3)+,x0         ; update r3; x0 is don't care.
  move.w  x:(r0)+n,x0 ; increment reverse carry index
  move.w  x:(r3)+,x0          ; update r3; x0 is don't care (can be done anywhere)
  inc.w   y1              ; Increment normal index
  move.w  r0,d1            ; copy reverse carry to d
End_Do                   ; end of zero overhead loop. Last item is moved below
  move.w  #-1,x0            ; all ones is reset configuration for M01
  moveu.w  x0,m01            ; set the m register to linear addressing
  move.w  x:(r1)+,a0          ; Move last pair to a0,a1. (all ones index)
  move.w  x:(r1)-,a1         ; Im part
  move.w  a0,x:(r3)+          ; Move a to last output location (all ones index)
  move.w  a1,x:(r3)-          ; Im part
  move.l  x:(sp)-,d          ; pop d10
  move.l  x:(sp)-,d2         ; pop d2
  move.l  x:(sp)-,c          ; pop c10
  move.l  x:(sp)-,c2          ; pop c2
  move.l  x:(sp)-,r1         ; pop r1
  rts                    ; return from subroutine, stack restored
         Fdfr16Cbitrev_END:
  ENDSEC

;************************* End of file ********************************a
```

3 Conclusion

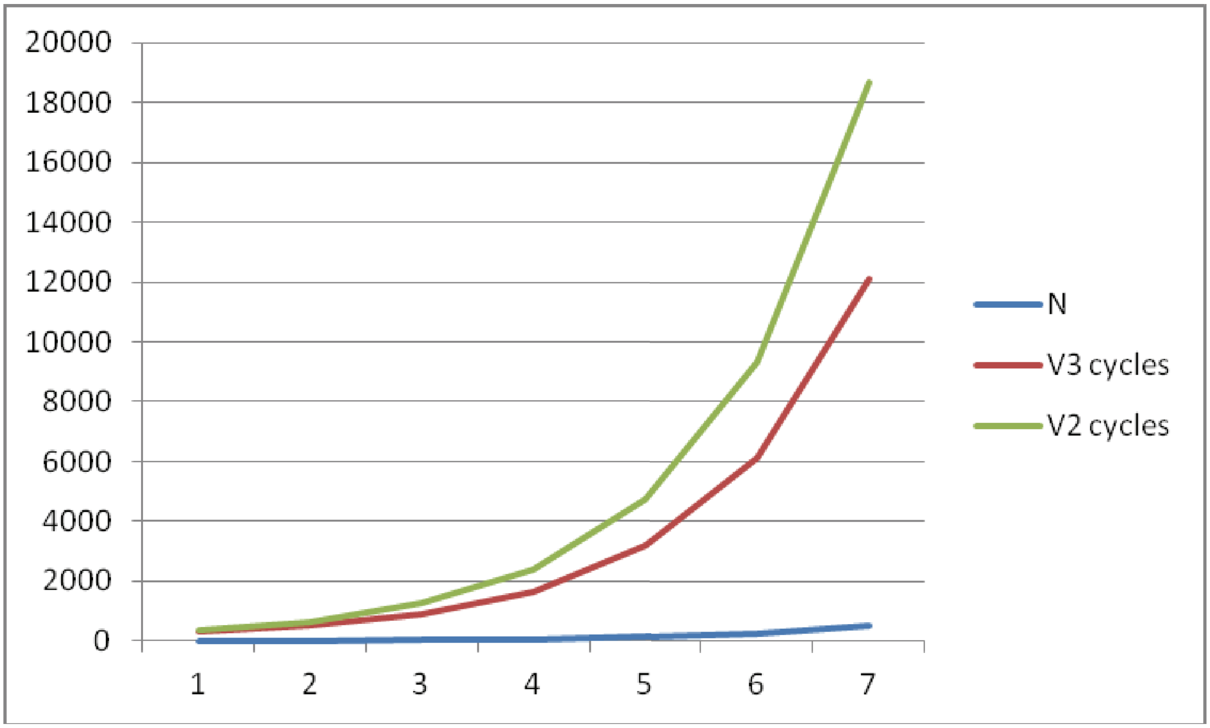
Using the method explained in this application note, DSC projects that consist of core/memory routines (not using peripherals) may be ported in a direct manner from older DSC products, starting with the first V2 DSC, the DSP56858, onwards. This includes the entire line of V2-core–based DSC products.

New core features from the MC56F84789 may be utilized directly in the resulting source code generated automatically by Processor Expert. This can enhance performance: in addition to utilizing the faster clock speeds of the newer devices, fewer machine cycles can be utilized.

The large collection of example code available to users of the DSP56858 is now readily portable and improvable to all DSC users. This code includes source code generation and included test harnesses.

The method for modification of generated assembly language routines is disclosed herein. The new bit-reverse function saves significant cycles in the execution of all the complex FFT cases tested.

How much was performance improved by using this new instruction application?



As can be seen above, the number of cycles saved increases with the size of the CFFT. In tabular form it appears as below.

Table 1. Cycle comparison

N	V3 cycles	V2 cycles	percent saved
8	315	371	15.09433962
16	499	651	23.34869432
32	899	1251	28.13749001
64	1635	2395	31.73277662
128	3171	4755	33.31230284
256	6115	9355	34.63388562
512	12131	18691	35.09710556

In the cases tested, up to 35 percent of the cycles needed for bit-reversal can be saved with the technique measured.

4 Testing and validation

The Processor Expert example project considered in this application note consists of a test harness that runs on the DSP56858. Once it was ported, running that test harness self-verified the port to the new target. Cycle time measurement utilized the onboard timer for cycle accurate measurement, including any bus or core stalls, for realistic numbers.

Were any flaw to be found in the bit-reverse function, the test would fail. During debugging of my code I actually saw this a few times.

The final project supplied with this application note includes software components that measure the cycle count using timers on the system on a chip, MC56F84789. The reader may then duplicate the results given the final project, CodeWarrior 10.5, and the TWR-MC56F8400 module which contains the MC56F84789 System on a Chip.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, CodeWarrior, and Processor Expert are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Tower is a trademark of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.
© 2014 Freescale Semiconductor, Inc.

Document Number: AN4945
Rev. 0
4/2014