Porting and Optimizing DSP56800 Applications to DSP56800E

Application Note

by

Cristian Caciuloiu, Radu Preda, Radu Bacrau, and Costel Ilas

AN2095/D
Rev. 0, 04/2001

MOTOROLA

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Freescale Semiconductor, Inc.

Abstract and Contents

The DSP56800E's DSP core architecture represents the next step in the evolution of Motorola's 16-bit DSP56800 Family of digital signal processors. It maintains compatibility with the DSP56800 while improving performance and adding new features. The main purpose of this application note is to recommend a method for porting DSP56800 applications to the DSP56800E and for optimizing the applications, exploiting the advantages of the new architecture.

An important feature of the DSP56800E is its source code compatibility with the DSP56800. Code developed for the DSP56800 can be assembled for the DSP56800E and will run correctly if certain coding requirements are fulfilled. These requirements are identified, analyzed, fulfilled, and verified with regard to example code in this application note.

1 Introduction

The DSP56800E's DSP core architecture represents the next step in the evolution of Motorola's 16-bit DSP56800 Family of digital signal processors. It maintains compatibility with the DSP56800 while improving performance and adding new features.

Some of the new and useful features of the DSP56800E (as compared to the DSP56800) that can be exploited for optimization include:

- Additional registers (accumulators, pointers, and an offset register)
- Extended set of data ALU operations
- AGU arithmetic
- Support for nested DO looping
- New data types (byte and long)
- 24-bit data memory address space and 21-bit program memory address space
- Support for real-time debugging (Enhanced OnCE™)

An important feature of the DSP56800E is its source code compatibility with the DSP56800. Code developed for the DSP56800 can be assembled for the DSP56800E and will run correctly if certain coding requirements are fulfilled. These requirements are not very restrictive. They are identified, analyzed, fulfilled, and verified with regard to example code in this application note. If these requirements are not met in the initial application, the necessary changes are easy to implement.

The sample code that was ported to the DSP56800E executed in half the number of cycles required by the DSP56800, even without any optimization (without exploiting the new features of the DSP56800E). Whereas the DSP56800 typically completes execution of an instruction in 2 cycles, the DSP56800E performs the same job in 1 cycle. Moreover, code written for the DSP56800 can be further optimized because of the new features introduced in the DSP56800E.

Another difference between the DSP56800E and the DSP56800 is new pipeline behavior. Although this behavior does not affect code correctness, in some cases it can introduce stalls. The programmer can avoid these situations by rearranging instructions.

The main purpose of this application note is to recommend a method for porting DSP56800 applications to the DSP56800E and for optimizing the applications, exploiting the advantages of the new architecture. This document also details some aspects of the following issues:

- The relative benefit of rewriting a function from scratch compared to porting and optimizing the application
- How pipeline effects on the DSP56800E can affect the ported code
- How an application can be translated beyond the 64 kwords boundary for program and data memory

1.1 Case Study

The application chosen as an example to be ported was the implementation of the International Telecommunications Union (ITU) Recommendation V.22 bis. The original code was taken from Freescale Embedded Software Development Kit (SDK) version 2.1. This software development kit, which runs with Metrowerks CodeWarrior 3.5.1 for the DSP56800 Family, can be found at the following URL:

http://www.freescale.com

The initial application could be run in two modes: either using a digital or an analog loopback (the latter could be run only on DSP56824 EVM). Modifications were made to the original code to run it only on the simulator (using only a digital loopback). Also, all calls to the SDK libraries were eliminated. These modifications affected only the tester, not the modem library.

The original code was initially optimized for the DSP56800. The result was considered the reference code for the next round of optimization (employing only the new features introduced by DSP56800E). The performance improvement gained after DSP56800E optimization is measured against the performance of this reference code and is discussed in Section 3, “Optimizing the Ported Code.”

1.2 References and Tools

This application note refers to the *DSP56800E 16-Bit Digital Signal Processor Core Reference Manual* (Order number DSP56800ERM/D) as the *Core Reference Manual*.

The tools used for developing and testing the code discussed in this document were the following:

- Metrowerks CodeWarrior 3.5.1 for DSP56800
- Prototype tools for DSP56800E including the assembler and the simulator

The porting process requires knowledge of several DSP56800 topics that are not explained in this application note. The following documents provide necessary information on these topics:

- *DSP56800 16-Bit Digital Signal Processor Family Manual* (Rev. 1.00, order number DSP56800FM/D)
- Freescale Embedded SDK 2.1 Help and Documentation: information about SDK libraries and system calls

In addition, coding requirements and recommendations used in this application note derive from a forthcoming guide on porting applications from the DSP56800 platform to the DSP56800E platform.

2 Application Porting

This section investigates porting a DSP56800 application to the DSP56800E architecture.

The entire application was tested and developed using a 16-bit address model for program and data memory space. Although the DSP56800E has larger addressing capabilities, they are not necessary for this application. Problems can result from using the extended DSP56800E program and data memory space for DSP56800 code. Section 6, “Converting Applications for Increased Data and Program Memory,” details these problems.

2.1 Porting Process

To set up the application to run on DSP56800E tools, the following steps were performed:

1. The original application code was tested, and test vectors were obtained. These steps are required for testing the ported code and for possible further optimization.
2. The original code's compliance with the coding requirements for porting was verified.

2.1.1 Running the Original Application Code and Obtaining Test Vectors

The original application was run in digital loopback mode under CodeWarrior 3.5.1 for DSP. The output of the digital loopback is a Boolean value resulting from a comparison between the transmitted and received data.

Using this version, we also obtained the test vectors: the data to be sent, the data received, and the information to be transmitted to the analog converter (data received as parameters by the transmitter's callback function, which should be sent by wire by the codec). Only the global test vectors were saved. Local test vectors, represented by the input and output data of a function, were saved later using DSP56800E tools, when they were required during the optimization process. The reason for this approach is that the local vectors were not required for porting, and it was simpler to save them from DSP56800E tools using simulator scripts (the CodeWarrior simulator does not accept scripts, and memory save and restore operations using I/O streams increase the simulation time).

2.1.2 Verifying the Coding Requirements

Coding practices are required to ensure that a DSP56800 program is compatible with the DSP56800E. The main code example that is featured in this application note met all requirements without being modified.

2.1.2.1 AGU Arithmetic Overflow and Underflow

Applications must be written so that there is no AGU overflow or underflow of 64K boundaries when data or program memory is accessed with the following DSP56800 addressing modes:

- (Rn)+
- (Rn)–
- (Rn)+N
- (SP–xx)
- (SP+xxxx)
- (R2+xx)

AGU overflow and underflow should not appear in normal conditions. They are not always easy to detect.

Consider a scenario showing the differences that appear when AGU overflow occurs. Assume that during the linking phase, a certain array or data structure is placed at the end of the addressable space, wrapping from a high address to a low address. To help detect such memory areas, an indication of the memory arrangement can be obtained from a memory utilization report generated by the assembler or from a linker map file.

One case is the access to this array using the (Rn)+ addressing mode. On DSP56800 architecture, the AGU overflows. When the application is ported to DSP56800E architecture, the wrapping does not occur and the array is placed in a contiguous space above the 64K boundary. Then, if the same addressing mode is used, the AGU calculates 24-bit addresses, overflow does not occur, and the array is accessed correctly.

Another case is when the (Rn) addressing mode is used and an LEA instruction updates the address register. As expected, the AGU overflows on the DSP56800. When the code is ported to DSP56800E, the results are different than in the preceding case. This addressing mode, which exists in the enhanced core to ensure DSP56800 compatibility, causes the AGU to produce 16-bit addresses by filling the upper 8 bits with 0, simulating an overflow. If the array is accessed with the sequence shown in Code Example 1, the next address will be forced to 16 bits, which is an error since the rest of the array is placed above 64K.

Code Example 1. Updating an Address Register (AGU Overflow or Underflow)

```
; accessing an array
    move    y0,x:(r2)                       ; writing Y0 at the address from R2
    . . .
; other code that might use the address from R2 register
    . . .
    lea     (r2)+n                          ; updating the R2 register with the increment from N
                                            ; the result is a 16-bit address on both architectures
```

The solution in this case is to replace the LEA instruction with ADDA, resulting in a 24-bit address.

However, there is no guaranteed method to detect these errors. Memory files can only give indications about data that might cause problems. Only a careful inspection of the code can reveal incompatibilities. The code chosen as the main example in this application note had no problems caused by AGU overflow or underflow.

2.1.2.2 MAC Output Limiter

Be careful with applications that enable the MAC output limiter (by setting the SA bit in the OMR). There are three instructions—ADC, SBC, and DIV—that are not affected by the state of the SA bit on the DSP56800E architecture but are affected on the DSP56800. When the DIV, ADC, or SBC instructions are executed, the accumulator extension registers must contain only sign extension, not significant bits.

Consider how the SA bit affects an ADC instruction on the DSP56800. The arithmetic instruction could be executed on a whole 36-bit accumulator, returning the correct 36-bit result when the SA bit is 0 or a 32-bit result when the SA bit is set. This feature was introduced so that the algorithms keep bit exactness on the DSP56800 (as compared to other DSPs that do not support high-precision arithmetic using extension bits). The MAC output limiter converts a 36-bit number to a 32-bit number. If it is a positive number and larger than the maximum value represented on 32 bits, it will be limited at $07FFFFFF; if it is a negative number that cannot be represented on 32 bits, it will be limited at $F80000000. On the DSP56800E, the SA bit does not affect an ADC instruction; the result has 36 bits.

Due to this effect, DSP56800 code that uses the MAC output limiter feature and that includes ADC, DIV, and SBC instructions that are affected by it should be rewritten so that these instructions are used only with 32-bit sign-extended accumulators. Code Example 2 illustrates the consequence of not following this recommendation.

Code Example 2. Using ADC with Non-Signed Operands (Different Results)

```
; example of the effect of the SA bit from OMR
    bfset    #$10,omr                   ; setting SA bit
    move     #$1000,y1
    move     #$F000,a1                  ; no sign extension in A register
    adc      y,a
;         y1= $1000 y0= $0000
; a2= $0 a1= $f000 a0= $0000 <= prior execution
;         y1= $1000 y0= $0000
; a2= $0 a1= $7fff a0= $ffff <= execution on DSP56800: limitation occurs
;         y1= $1000 y0= $0000
; a2= $1 a1= $0000 a0= $0000 <= execution on DSP56800E: no limitation
```

To avoid this problem, scan the code for ADC, SBC, or DIV instructions that are run with the SA bit set. If the operands are not always 32-bit and sign extended or if the result of the operation having 32-bit sign-extended operands is larger than 32 bits, the solution is the SAT instruction. This instruction, available only on the DSP56800E, forces the saturation (see Code Example 3). It is safe to place a SAT instruction after ADC, SBC, or DIV, explicitly saturating the result.

Code Example 3. Using ADC with Non-Signed Operands (with Correction)

```
; example of the effect of the SA bit from OMR
    bfset    #$10,omr                ; setting SA bit
    move     #$1000,y1
    move     #$F000,a1               ; no sign extension in A register
    adc      y,a
    sat      a,a                     ; this instruction manually forces saturation
;        y1= $1000 y0= $0000
; a2= $f a1= $f000 a0= $0000 <= prior execution
;        y1= $1000 y0= $0000
; a2= $0 a1= $7fff a0= $ffff <= execution on DSP56800E: limitation caused by SAT instruction
```

The code used as the main example in this application note did not face this type of limiting problem.

2.2 Application Performance Comparison

Test vectors were generated and coding requirements were verified, and the main test application ran correctly on both DSPs. Immediately afterward, all DSP56800 code optimization methods were implemented. The correctness of the results was preserved on both platforms. The code obtained after this step was considered the reference code for all subsequent optimization allowed by the new features of the DSP56800E.

The reference application's different speeds on the two platforms indicate the improvement was achieved only by porting. The speeds were measured with the function testLoopback (representing the entire application, including not only the V.22 bis library but also the main function, which realizes the digital loopback). On the DSP56800, this function took **61,918,898** cycles (the code was run using the CodeWarrior simulator). The ported code on the DSP56800E was expected to run about two times faster. Indeed, the function testLoopback on the DSP56800E took **31,694,501** cycles.

The DSP56800E measurement is slightly more than half the number of cycles on the DSP56800. The difference can be explained in that some instructions that take more than half the cycles to execute unlike similar instructions on the DSP56800 (such as those for change of flow: Bcc, BRA, Jcc, JMP). Also, the new pipeline effects on the DSP56800E (not present on the DSP56800) slightly increase on the number of cycles.

More interesting is the study of performance for the V.22 bis library alone. For this reason, in the remainder of this document, size measurements refer only to the size of the library, and speed measurements refer to the worst case for a symbol transmitting and receiving (these two tasks are performed by the functions TXBAUD and RXBAUDPROC, respectively).

Table 1 presents a comparison between the size of the V.22 bis library built and run on both the DSP56800 and DSP56800E (the same source code, after minor modifications required for porting).

Table 1. Size Comparison Between DSP56800 and DSP56800E Ported Code

Platform	Program Memory (Words)	Data Memory (Words)	
		Constants/Tables	Variables
DSP56800	4735	1092	909
DSP56800E	4973	1092	909

NOTE:

Throughout this application note, 1 word equals 16 bits.

The size of the data memory does not include the gaps that the circular buffers introduce.

The size of data is the same on both platforms (as expected), but the size of the code is slightly (5 percent) larger on the ported application. One explanation is that some instructions (such as Bcc) are coded with more words on the DSP56800E than on the DSP56800. Program space also increases somewhat due to the different coding of some addressing modes. For example, a MOVE instruction with an immediate operand and an indexed operand, addressed using R2, is coded on 2 words on the DSP56800E and 1 word on the DSP56800.

Other interesting information is the processing load of the DSP, measured in million cycles per second (MCPS). It is computed as the worst-case number of cycles needed to transmit or receive a symbol, multiplied by 600 symbols per second (V.22 bis assumes that 600 symbols per second are transmitted at 2400 Hz or 1200 Hz), and divided by 1,000,000. Processing load for the full duplex mode includes processing loads for both transmission and reception. For example, if the worst cases for transmission and reception are 1352, respectively 9870 cycles, the worst case for full duplex mode is 1352 + 9870 = 11222 cycles. Processing load for full duplex is 11222 × 600 / 1000000 = 6.73 MCPS.

The comparison performed on worst case and processing load illustrates that, as with the speed measurements for the entire application, the number of cycles on the DSP56800E is slightly more than half the number of cycles on the DSP56800. See Table 2.

Table 2. Cycle Count Comparison of Ported DSP56800 Code to DSP56800E

Data Component or Mode	Worst Case (Cycles)		Processing Load (MCPS)		Speedup Factor
	DSP56800	DSP56800E	DSP56800	DSP56800E	
Transmitter	1352	571	0.81	0.34	2.33
Receiver	9870	5154	5.92	3.09	1.92
Full duplex	11222	5697	6.73	3.43	1.97

Note that the DSP56800E also runs at higher clock rates; the DSP56800E runs at a clock frequency of 120 MHz, while the DSP56800 runs at 35 MHz. Thus, the actual execution time is much shorter, and the total improvement in speed obtained by simply porting DSP56800 code to the DSP56800E is significantly higher.

3 Optimizing the Ported Code

This section summarizes some of the optimization practices that are allowed by the new features of the improved core architecture. The section also assesses the improvements achieved by implementing these practices.

The DSP56800E optimization methods that were used do not attempt to make any change in the algorithm or in the program flow. The modular structure defined by the functions in the reference code is maintained. The optimization methods were applied at function level, by replacing selected limited code sequences with optimized equivalents. All of the methods comply with recommended coding practices.

Methods based on almost all the new DSP56800E features, which are summarized in Section 1, “Introduction,” were used to improve the speed of the application. Table 3 and Table 4 present the overall results obtained after most of the time-consuming functions were optimized.

Table 3. Size Comparison

Code	Program Memory (Words)	Data Memory (Words)	
		Constants/Tables	Variables
DSP56800 reference	4735	1092	909
DSP56800E ported	4957	1092	909
DSP56800E optimized	4856	1100	911
Optimized vs. ported	–2.04%	+0.73%	+0.22%
Optimized vs. reference	+2.10%	+0.73%	+0.22%

The program memory size decreased by 2 percent compared to the ported code, nearing the size of the DSP56800 reference code. Better code density was achieved because more registers were used and some instructions were replaced with more flexible ones. The data memory area slightly increased because of the gaps introduced by alignment requirements.

Table 4 illustrates the speed improvements.

Table 4. Speed Comparison

Data Component or Mode	Code	Worst Case (Cycles)	Processing Load (MCPS)	Speedup Factor Relative to Reference
Transmitter	DSP56800 reference	1352	0.81	N/A
	DSP56800E ported	571	0.34	2.37
	DSP56800E optimized	498	0.30	2.71
Receiver	DSP56800 reference	9870	5.92	N/A
	DSP56800E ported	5154	3.09	1.91
	DSP56800E optimized	4692	2.81	2.10
Full duplex	DSP56800 reference	11222	6.73	N/A
	DSP56800E ported	5697	3.43	1.97
	DSP56800E optimized	5190	3.11	2.16

The DSP56800E optimization methods produced an increased speed of about 9 percent compared to the ported version. The optimized version runs in less than half the number of cycles compared to the DSP56800 reference version. Remember that the actual execution time is much shorter, since the DSP56800E is a faster processor.

Table 5 indicates the effects of optimizing the most time-consuming functions. The results are presented globally, not for each individual act of optimization.

Table 5. Optimization Gains on the Most Time-Consuming Functions

Function	Initial	Final	Gain
RXBPF	2158	2015	6.63%
RXDEMOD	717	641	10.60%
RXINTP	265	252	4.91%
RXCDAGC	152.67	142.54	6.64%
RXDECIM	118	116	1.69%
tx_fm	192	190	1.04%
RXEQUD	201	199	1.00%
TONEDETECT	318	286	10.06%
RXS1	110.25	106.53	3.37%
RXUSB1	71.74	69.73	2.80%
rx_dscr	125.03	104.47	16.44%
RXEQERR	99.54	95.54	4.02%
tx_scr	81.86	73.33	10.42%

The optimization techniques applied to the reference code to reach the speed and size improvements are discussed in the following subsections.

3.1 Delay Slots on Change of Flow

The delayed flow control instructions are designed to increase throughput by eliminating execution cycles that are wasted when program flow changes. An instruction that affects normal program flow (such as a branch or jump instruction) requires 2 or 3 additional instruction cycles to flush the execution pipeline. The program controller stops fetching instructions at the current location and begins to fill the pipeline from the target address. The execution pipeline stalls while this switch occurs. The additional cycles required to flush the pipeline are reflected in the total cycle count for each change-of-flow instruction. A special group of instructions referred to as “delayed” instructions provide a mechanism for executing useful tasks during these normally wasted cycles.

It is simple to employ these instructions. Replace every BRA, JMP, RTI, and RTS instruction with BRAD, JMPD, RTID, and RTSD, respectively. These instructions provide a number of delay slots (see Table 6), which must be filled with instructions.

Table 6. Available Delay Slots

Delayed Instructions	Number of Delay Slots
BRAD	2
JMPD	2

Table 6. Available Delay Slots (Continued)

Delayed Instructions	Number of Delay Slots
RTID	3
RTSD	3
FRTID	2

The number of instruction words filling the delay slots must equal the number of delay slots. If some delay slots cannot be filled with valid instructions, then each unused delay slot must be filled with a NOP instruction. If a pipeline dependency occurs due to instructions executed in the delay slots, the appropriate number of interlock cycles are inserted by the core, reducing correspondingly the number of delay-slot cycles that are available for instructions.

Code Example 4. Pipeline Dependency in Delay Slot

```
        jmpd        rx_next_task          ; 2-3 cycles 2-3 words
        mpy         y0,x0,b               ; 1 cycle    1 word
        move.w      b,x:(r1)+n            ; 1 cycle    1 word
```

In Code Example 4, the pipeline dependencies force the MOVE.W instruction in the delay slot to execute in 2 cycles instead of 1. (For more on pipeline dependencies, see Section 5, “Pipeline Effects on DSP56800E.”) The total number of instruction cycles in the delay slots is 3, while JMPD provides only 2 delay slots.

An example from rx_eqerr.asm appears in Code Example 5.

Code Example 5. Using Delay Slots

```
    move        y1,x:>LASTDP          ; 2 cycles   2 words
    add         y1,b                  ; 1 cycle    1 word
    move.w      b,x:(r1)+n            ; 1 cycle    1 word
End_RXEQERR
    jmp         rx_next_task           ; 4-5 cycles 2-3 words
; DSP56800 original code:   8-9 cycles / 6-7 words

    move        y1,x:>LASTDP          ; 2 cycles   2 words
End_RXEQERR
; use delay slots
    jmpd        rx_next_task          ; 2-3 cycles 2-3 words
    add         y1,b                  ; 1 cycle    1 word
    move.w      b,x:(r1)+n            ; 1 cycle    1 word
; DSP56800E optimized code: 6-7 cycles / 6-7 words
```

The first MOVE instruction does not insert any pipeline interlocks, and the ADD instruction in the delay slot is executed normally (in 1 cycle and with 1 word). Also, the second MOVE.W instruction takes 1 cycle and 1 word.

This optimization can be easily implemented, and it was widely used in the code selected for this application note. It can be used in every type of code, DSP or control, with respect to the restrictions specified in the *Core Reference Manual*.

The main benefit of this optimization is obtained in control code, where flow control instructions are heavily used.

3.2 New Registers

Compared to the DSP56800, the DSP56800E has the following new registers:

- Two new accumulators, C and D

- Two new address registers, R4 and R5
- A second offset register, N3
- New loop address and counter registers, LA2 and LC2
- FISR and FIRA registers
- Shadow registers for R0, R1, N, and M01

LA2 and LC2 are discussed in Section 3.8. The second offset register, N3, can be used for the second memory read in a dual parallel memory read, but it was not used in this project. FISR, FIRA, and the shadow registers support faster interrupt processing, but this subject is beyond the scope of this note.

Generally, using the new DSP56800E registers reduces the register pressure in some parts of the program, thus eliminating additional memory loads and stores (spill code).

The new accumulators can also be used to eliminate some moves from an accumulator to another register that are required on the DSP56800. These moves are required on that platform because the results of certain instructions are always obtained in an accumulator, and the value in that accumulator must be saved first.

Using the new registers in these ways is not usually automatic. Data flow and control flow must be inspected to see which registers are used and how, and so forth.

These methods of optimization apply to both control and DSP code. In the sample code, new registers were used on almost all of the functions that were optimized.

Code Example 6 shows how to replace spill in memory with spill in the new accumulators. Both this example and Code Example 7 are taken from the function RXBPF from the file rx_bpf.asm.

Code Example 6. Avoiding Saving Variables in Memory

```
        do          #12,END_RX_BPF
        ...
        move        a,x:TEMP1             ; 2 cycles, 2 words
        move        b,x:TEMP2             ; 2 cycles, 2 words
        ...
        move        x:TEMP1,y0            ; 2 cycles, 2 words
        ...use y0
        move        x:TEMP2,y0            ; 2 cycles, 2 words
        ...use y0
END_RX_BPF
; DSP56800 original code:   12*8 cycles / 8 words

        do          #12,END_RX_BPF
        ...
        move.w      a,c1                  ; 1 cycle, 1 word
        move.w      b,d1                  ; 1 cycle, 1 word
        ... calculations
        ...use c1
        ...use d1
END_RX_BPF
; DSP56800E optimized code: 12*2 cycles / 2 words
```

Code Example 7 presents one of the ways that new address registers can be used. R3 is not loaded with an immediate value inside the loop; instead, the immediate value is preloaded in R4 before the loop starts, and R4 is used inside the loop.

Code Example 7. Using Address Registers to Store Addresses

```
    do          #12,END_RX_BPF
    move        x:>BPF_PTR,r3         ; 2 cycles, 2 words
    ... use and modify r3
END_RX_BPF
; DSP56800 original code: 12*2 cycles / 2 words

    move.w      x:>BPF_PTR,r4         ; 2 cycles, 2 words
    do          #12,END_RX_BPF
    tfra        r4,r3                 ; 1 cycle, 1 word
    ... use and modify r3
END_RX_BPF
; DSP56800E optimized code: 2+12*1 cycles / 2+1 words
```

The optimization methods presented save 74 cycles per symbol out of an initial average of 4278.5 cycles per symbol, resulting in an improvement of 1.7 percent.

3.3 Immediate Operands

This optimization method consists of using ADD, SUB, and CMP between a register and an immediate value directly instead of first loading the immediate into a temporary register.

On the DSP56800, there are two ways to use ADD, SUB, or CMP between an immediate value and a register. The first one is to load the immediate into a register and then perform the operation between two registers. The second one is to use the immediate directly. Both variants have the same speed, but the second takes 1 less word to be encoded. The variants are presented in Code Example 8.

Code Example 8. Two Ways of Performing CMP with Immediate on DSP56800

```
    move        #$125,x0              ; 4 cycles, 2 words
    cmp         y0,x0                 ; 2 cycles, 1 word
; DSP56800 original code: 6 cycles / 3 words
    cmp         #$125,x0               ; 6 cycles, 2 words
; DSP56800 code optimized for size: 6 cycles / 2 words
```

On the DSP56800E, both variants are possible, but the second variant takes not only 1 fewer word but also 1 fewer cycle. See Code Example 9.

Code Example 9. Two Ways of Performing CMP with Immediate on DSP56800E

```
    move.w      #$125,x0              ; 2 cycles, 2 words
    cmp.w       y0,x0                 ; 1 cycle, 1 word
; DSP56800E original code: 3 cycles / 3 words
    cmp.w       #$125,x0              ; 2 cycles, 2 words
; DSP56800E code optimized for size and speed: 2 cycles / 2 words
```

This method of optimization can often be used automatically. However, programmers must be careful to observe whether the immediate value that is loaded into the register is used somewhere else. If it is, this method cannot be used.

This method was performed on more than half the functions that were optimized because in the main code example, most all comparisons with immediates were performed through intermediate registers.

3.4 AGU Arithmetic

Compared to the DSP56800, the arithmetic capabilities of the AGU improved considerably on the DSP56800E. Pointer arithmetic can be done directly in the AGU, whereas on the DSP56800 the arithmetic operations must be performed in the data ALU and the result must be transferred in a pointer. These new DSP56800E facilities provide capabilities to improve both speed and code density.

The improvement achieved by the AGU arithmetic is illustrated in Code Example 10, which is taken from the function RXDEMOD (from the file rx_demod.asm).

Code Example 10. Performing Address Calculations on DSP56800

```
        do          #12,end_rx_demod        ; Loop 12 times
        ...
        move.w      #SIN_TBL,y0             ; Get address of the table
        add.w       a1,y0                   ; Add offset to the start address
        move.w      y0,r1                   ; Load into the address register
        ...
end_rx_demod
; DSP56800 original code: 12*4 cycles / 4 words
```

The presented sequence is 4 words in size and runs in 4 cycles. Code Example 11 shows how the code in Code Example 10 can be rewritten using AGU arithmetic.

Code Example 11. Performing Address Calculations on DSP56800E Using AGU

```
        move.l      #SIN_TBL,r5             ; SIN_TBL address kept in r5
        ...
        do          #12,end_rx_demod        ; Loop 12 times
        ...
        move        a1,r1                   ; Load offset in r1
        adda        r5,r1                   ; Add offset to the start address
        ...
end_rx_demod
; DSP56800E optimized code: 12*2+3 cycles / 4 words
```

The size of the code remains 4 words (considering the entire function), but the code runs in 2 cycles inside the loop (which means that the improvement must be multiplied by the number of loops) plus 3 additional cycles outside the loop. Of course, the code could be also written as in Code Example 12.

Code Example 12. Size Optimization on DSP56800E Using AGU

```
        do          #12,end_rx_demod        ; Loop 12 times
        ...
        adda        #SIN_TBL,a1,r1          ; Add offset a1 to the start address
                                            ;    and put result in r1
        ...
end_rx_demod
; DSP56800E optimized code: 12*4 cycles / 2 words
```

This optimization method is for size. The size decreases by 2 words, but the code would still run in 4 cycles.

The arithmetic operations performed for initializing pointers can be easily identified, and the modifications can be made rather easily if there is no pressure on pointer registers.

3.5 Operations and Memory Access on 32 Bits

Another feature introduced in the DSP56800E core is 32-bit operations. The arithmetical and logical operations (such as ADD, SUB, CLR, and ASL) are extended to support 32-bit operands. The extended instructions have the suffix ".L" (ADD.L, SUB.L, CLR.L, and ASL.L). Memory access on 32 bits is also possible with the new core.

Code Example 13 (taken from the function tx_a_ton from the file tx_feed.asm) illustrates how an array of data can be copied on the DSP56800. An array with 16-bit elements is updated with values read from a table. Each value is read in the register X0, and then it is written in the array. In the original version there were no restrictions regarding the alignment of the array.

Code Example 13. Copying a Buffer on DSP56800

```
        SECTION     TX_MEM
        ...
        tx_out      ds      12              ; The output buffer
        ...
        ENDSEC

        SECTION     V22B_TX
        ...
        move        #tx_out,r1              ; Load address of output buffer
        do          #12,up_txout            ; Repeat 12 times
        move        x:(r0)+,x0              ; Update 16-bit values array with
        move        x0,x:(r1)+              ;    values obtained from a table.
up_txout
        ...
        ENDSEC
; DSP56800 original code: 12*2 cycles / 2 words
```

This code was rewritten using 32-bit memory access. Each value is read in a 32-bit accumulator and is stored in the array using 32-bit access (see Code Example 14).

Code Example 14. Copying a Buffer on DSP56800E

```
        SECTION     TX_MEM
        ...
        tx_out      dsm     12              ; The output buffer
        ...
        ENDSEC

        SECTION     V22B_TX
        ...
        moveu.w     #tx_out,r1              ; Load address of output buffer
        do          #6,up_txout             ; Repeat 6 times
        move.l      x:(r0)+,c               ; Update 16-bit values array with
        move.l      c10,x:(r1)+             ;    table. The table is read on 32-bit.
up_txout
        ...
        ENDSEC
; DSP56800E optimized code: 6*2 cycles / 2 words
```

The new code has the same size, but it executes two times faster because the loop is executed only six times instead of twelve as in the initial version. This optimization method required the alignment of the vector pointed to by R0 at a 2-word boundary, which was achieved using the assembler directive DSM (see Code Example 14).

Code Example 15 (taken from the function tx_scr from the file tx_scr.asm) presents how multi-bit 32-bit shifting can be performed in a single instruction instead of by using a REP followed by a 36-bit shifting.

Code Example 15. Multi-Bit Shifting

```
        rep         n                       ; takes 2 cycles, 1 word
        asl         a                       ; takes 1 cycle,  1 word
; DSP56800 original code: 2+n*1 cycles / 2 words

        asll.l      d,a                     ; takes 2 cycles, 1 word
; d contains the same value as n in original code
; DSP56800E optimized code: 2 cycles / 1 word
```

This particular optimization cannot be applied automatically. Programmers must be careful to determine whether the original program really required 32-bit instead of 36-bit shifting and whether saturation issues could appear.

The two instances in Code Example 14 and Code Example 15 are the only places in the main project where 32-bit operations and memory access were used, because the processing unit is a nibble (4 bits).

3.6 Operations and Memory Access on 8 Bits

Compared to its predecessor, the DSP56800E architecture introduced another new data type: 8-bit data. There are instructions that have an 8-bit operand in memory. The 8-bit data can be accessed using two types of pointers: word pointers and byte pointers. The *Core Reference Manual* contains more information about these features.

Using this new data type reduces the amount of data memory. Byte arrays can be optimally used on DSP56800E architecture.

An example is the function contained in tx_enc.asm. To encode 4 bits, this function uses an array containing offsets from another table. These offsets are 2 bits wide. However, code complexity and size would increase if these offsets had been further compacted.

The original table is defined as an array of 16 words (see Code Example 16).

Code Example 16. Array with 16-Bit Values on DSP56800

```
        dc          2,0,3,1
        dc          3,2,1,0
        dc          0,1,2,3
        dc          1,3,0,2
    ; DSP56800 original code: data memory 16 words
```

Using 8-bit accesses, the table can be redefined as an array of 16 bytes. Each unit of information that was previously stored in a word is now represented as a byte (see Code Example 17).

Code Example 17. Array with 8-Bit Values on DSP56800E

```
        dcb         0,2,1,3
        dcb         2,3,0,1
        dcb         1,0,3,2
        dcb         3,1,2,0
    ; DSP56800E optimized code: data memory 8 words
```

The array elements are accessed using a MOVE.BP instruction, as in Code Example 18, which also presents the instruction used in the original DSP56800 code.

Code Example 18. Accessing an 8-Bit Memory Value on DSP56800E

```
        move        x:(r1)+,y1              ; accessing a word from memory
    ; DSP56800 original code: 1 cycle / 1 word

        move.bp     x:(r1)+,y1              ; accessing a byte from memory
                                            ;   done with a byte pointer which
                                            ;   supports this addressing mode
    ; DSP56800E optimized code: 1 cycle / 1 word
```

There are differences between a word pointer and a byte pointer. The latter has more flexibility when used with a larger variety of addressing modes.

This optimization method involves the modification of the data structures. It is therefore difficult to use, but the benefits are obvious and very important in memory-constrained systems.

3.7 New Addressing Modes and New Register Combinations in Data ALU Operations

As mentioned in Section 1, “Introduction,” the DSP56800E introduces some microcontroller features. One such feature is the extension of the addressing modes in the arithmetic instructions such as ADD and SUB. These improvements provide more flexibility than do the similar instructions on the DSP56800.

For example, compare the instruction ADD on the DSP56800 to the instructions ADD, ADD.L, and ADD.W on the DSP56800E. On the DSP56800, ADD operands can use the following addressing modes: register, immediate, direct, and displacement relative to SP. Of course, there are restrictions regarding the allowed register combinations. On the DSP56800E, these restrictions disappear, and new addressing modes exist for ADD.W: indirect and indexed.

Code Example 19, taken from the function RXDEMOD (from the file rx_demod.asm), shows how the ADD instruction was used on the DSP56800.

Code Example 19. ADD Usage on DSP56800

```
        do          #12,end_rx_demod        ;Loop 12 times
        ...
        move        x:>CDP,a                ;Load CDP
        move        x:>DPHASE,y0            ;Load DPHASE
        add         y0,a                    ;Update DPHASE value
        move        a1,x:>DPHASE            ;Save DPHASE
        ...
end_rx_demod
; DSP56800 original code: 12*7 cycles / 7 words
```

The new pointer R4 could be used for addressing the variable DPHASE, which would allow the code sequence to run faster. Also, the new accumulator C can be used to store the constant CDP. The rewritten code is presented in Code Example 20.

Code Example 20. Optimized Code Using New Addressing Modes

```
        move.l      #DPHASE,r4              ;pointer of DPHASE kept in r4
        move        x:>CDP,d                ;constant kept in d
        ...
        do          #12,end_rx_demod        ;Loop 12 times
        ...
        tfr         d,a                     ;Load CDP from D
        add.w       x:(r4),a                ;Update DPHASE value using indirect
                                            ;   addressing
        move.w      a1,x:(r4)               ;Save DPHASE using indirect addressing
        ...
end_rx_demod
; DSP56800E optimized code: 12*4 cycles / 7 words
```

The first version of the code sequence executed in 7 cycles; the modified version executed in only 4 cycles. When this gain is multiplied by the number of loops (because the sequence is in a loop), the total improvement is considerable. The code size was not modified with regard to the entire function (RXDEMOD). Although the modified sequence is 4 words smaller, there are 2 move instructions added outside this sequence to initialize the accumulator C and the pointer R4.

This method of optimization cannot be considered automatic because it depends on the availability of the pointer register to keep the memory address of variables that are frequently accessed. In addition, it requires a careful analysis of the entire function.

Another improvement is the elimination of the many restrictions regarding register combinations in data ALU operations. Consider Code Example 21, which was also extracted from RXDEMOD.

Code Example 21. Restrictions Using MACR on DSP56800

```
        do          #12,end_rx_demod        ;Loop 12 times
        ...
        move        y0,y1                   ;transfer y0 to y1 to allow the
                                            ; following macr on DSP56800
        macr        b1,y1,a
        ...
end_rx_demod
; DSP56800 original code: 12*2 cycles / 2 words
```

The transfer from Y0 to Y1 is necessary on the DSP56800 because of restrictions regarding operands of MACR (and similar instructions). On the DSP56800E, this restriction does not exist, and the code can be written as in Code Example 22.

Code Example 22. Restrictions Removed Using MACR on DSP56800E

```
        do          #12,end_rx_demod        ;Loop 12 times
        ...
        macr        b1,y0,a                 ;the register combination is
                                            ; allowed on DSP56800E
        ...
end_rx_demod
; DSP56800E optimized code: 12*1 cycles / 1 word
```

In terms of size, the gain is 1 word. In terms of speed, the gain is 1 cycle multiplied by the number of loops (because the sequence is extracted from a loop).

Generally, wherever there are transfers between registers and these registers are used in the following data ALU instructions, this method of optimization can be used automatically. However, be sure to check whether the value stored in a register is used later in the program.

3.8 Nested Loops

The DSP56800E improves hardware support for DO looping. Unlike the DSP56800, which supports one single hardware loop, the new core supports two nested hardware loops with no overhead.

To perform two nested loops on the DSP56800, the user must insert additional code that saves the LC and LA registers before the inner loop and then restores them after the **inner** loop. See Code Example 23.

Code Example 23. Two Nested Loops on DSP56800

```
        do          #times_outer,END_OUTER_LOOP
        ...
        lea         (sp)+                   ; 1 cycle / 1 word
        move        la,x:(sp)+              ; 1 cycle / 1 word
        move        lc,x:(sp)               ; 1 cycle / 1 word
        do          #times_inner,END_INNER_LOOP
        ...
END_INNER_LOOP
        pop         lc                      ; 1 cycle / 1 word
        pop         la                      ; 1 cycle / 1 word
        ...
END_OUTER_LOOP
```

The DSP56800E core directly supports two nested hardware loops by automatically saving LA and LC into the new LA2 and LC2 registers before the inner loop and restoring them afterward. See Code Example 24.

Code Example 24. Two Nested Loops on DSP56800E

```
        do          #times_outer,END_OUTER_LOOP
        ...
        do          #times_inner,END_INNER_LOOP
        ...
END_INNER_LOOP
        ...
END_OUTER_LOOP
```

Code Example 23 contains five additional instructions compared to Code Example 24. All of these instructions take 1 cycle to execute on the DSP56800. By eliminating these instructions that are unnecessary on the DSP56800E, the code has a gain of 5 cycles that is multiplied by the number of times the outer loop is executed.

This optimization method can be considered automatic. To quickly identify instances where this method can be used, search for occurrences of DO and then examine whether any occurrence is preceded by instructions that save LA and LC.

This optimization is especially suitable for DSP code, where nested loops are more frequently used.

The main project contains only one place where two imbricated DO loops are used, in the function RXBPF from the file rx_bpf.asm. The function performs band pass filtering and contains an outer loop that executes 12 times.

This optimization method saves 60 (5 × 12) cycles per symbol out of an initial average of 4278.5 cycles per symbol. This single method produces an improvement of 1.4 percent.

4 Writing DSP56800E Code from Scratch

The methods described in Section 3, “Optimizing the Ported Code,” preserve the original design of the functions. However, this design was influenced by DSP56800 limitations. This section compares the results of optimizing the ported code to those of writing entirely new DSP56800E code “from scratch.”

The functions RXDEMOD, a DSP function, and RXEQERR, a control function, illustrate the comparison. Both of them were written from scratch, tested, and benchmarked. Then they were compared to the optimized ported versions.

Writing DSP56800E code from scratch obtained better use of the following features (compared to the process of optimizing the ported code):

- Increased register set. When a function is being designed, the additional accumulators, address registers, and index registers provide more flexibility in arranging variables in registers and in deciding which variables to store in memory.
- More flexible instruction set. The new register combinations and addressing modes that the DSP56800E allows for many instructions provide more freedom to place variables in registers and to design the data flow.
- AGU arithmetic. When DSP56800E code is written from scratch, AGU arithmetic is naturally used whenever pointer manipulation is required. This capability eliminates some transfers from data registers to address registers and also enables the application to use the extended memory space.
- New data types. Instead of being used to modify existing code and data structures, 32-bit and 8-bit instructions and memory access can be used more simply from the start.

The main reason why writing from scratch is better is that it enables programmers to reconsider the code in its entirety. When optimizing ported code, one usually inspects small groups of instructions that can be replaced with other groups of instructions and usually avoids considering larger portions of code.

The optimized ported versions and the written from scratch versions of the two functions are presented in Appendix A. Section 4.1, “RXDEMOD,” and Section 4.2, “RXEQERR,” present specific issues about these functions. The code examples presented in these subsections were obtained after the functions were rewritten and parts of equivalent code were identified.

4.1 RXDEMOD

Comparative results for the initial code, optimized ported code, and written from scratch code are presented in Table 7.

Table 7. Results Obtained for RXDEMOD

RXDEMOD	Speed				Size	
	Minimum (Cycles)	Maximum (Cycles)	Average (Cycles)	Gain Over Initial (%)	Value (Words)	Gain Over Initial (%)
Initial	745	745	745	N/A	68	N/A
Optimized	621	621	621	16.64	65	4.41
Written from scratch	522	522	522	29.93	71	–4.41

The most important new features of the DSP56800E used in writing RXDEMOD from scratch were the increased number of registers and the increased number of register combinations for different instructions (which are allowed by the more flexible instruction set).

A first specific difference between the optimized ported version and the written from scratch version is that the latter chooses other variables (DPHASE instead of CDP) to stay in registers and preloads two more constants in registers to be available in the inner loop (mod_tbl_offset and #$0040-1). This arrangement makes better use of the registers.

Another difference is that the written from scratch version uses the new DSP56800E AGU instruction ZXTA.B, which takes 1 cycle, instead of BFCLR, which takes 2 cycles. See Code Example 25.

Code Example 25. Using DSP56800E AGU Instruction

```
    bfclr       #$FF00,a                ; 2 cycles, 2 words
    ...
    move.w      a1,r1                   ; 1 cycle, 1 word
; DSP56800E optimized code: 3 cycles / 3 words

    move.w      a1,r1                   ; 1 cycle, 1 word
    ...
    zxta.b      r1                      ; 1 cycle, 1 word
; DSP56800E written from scratch code: 2 cycles / 2 words
```

The register combinations used for MPY and MACR in Code Example 26 are not valid on the DSP56800 but are valid on the DSP56800E.

Code Example 26. New Register Combinations Allowed on DSP56800E

```
    mpy         b1,y0,b                 ; 1 cycle, 1 word
    macr        -a1,y1,b                ; 1 cycle, 1 word
```

To perform these instructions, the original DSP56800 code first exchanges values between Y0 and Y1 to obtain a valid register combination. This extra step adds 3 more instructions, each taking 1 cycle to execute, as presented in Code Example 27.

Code Example 27. Additional Code Needed by Less Flexible DSP56800 Instruction Set

```
    move.w      y0,n                    ; 1 cycle, 1 word
    move.w      y1,y0                   ; 1 cycle, 1 word
    move.w      n,y1                    ; 1 cycle, 1 word
    mpy         b1,y1,b                 ; 1 cycle, 1 word
    macr        -a1,y0,b                ; 1 cycle, 1 word
```

The limited number of accumulators forces the original DSP56800 code to frequently move results from accumulators to other registers to make room for the results of the next operations.

The original code contains 10 register-to-register moves that compensate for the lack of accumulators and the reduced number of register combinations for MACs. The optimized ported code eliminates five of these moves, leading to an improvement of 60 (12 × 5) cycles on the entire function. The written from scratch version eliminates all of these transfers, leading to an improvement of 60 additional cycles, or a total improvement of 120 cycles.

4.2 RXEQERR

The written from scratch RXEQERR function has a new design. By arranging values in registers without considering the initial DSP56800 design, it performs more parallel moves, and by choosing other variables to be stored in registers, the new code avoids a few memory transfers. This redesign leads to a gain of 7–12 cycles, depending on the function flow.

Note that the original DSP56800 code does not use parallel moves, but the optimized version rearranges the values in memory so that parallel moves can be performed. The gain of 7–12 cycles is relative to the optimized version.

Comparative results for the initial code, optimized ported code, and written from scratch code are presented in Table 8.

Table 8. Results Obtained for RXEQERR

RXEQERR	Speed				Size	
	Minimum (Cycles)	Maximum (Cycles)	Average (Cycles)	Gain (%)	Value (Words)	Gain (%)
Initial	52	126	99.50	N/A	108	N/A
Optimized	50	124	97.50	2.01	108	0.00
Written from scratch	44	95	77.66	21.94	81	25.00

Much of the improvement results from better coding rather than from using new DSP56800E features. For example, in the original code, there is a check at one point whether two variables (A and B) have different signs. If they do, then the value in A is negated. The optimized ported code is presented in Code Example 28.

Code Example 28. Optimized Ported Code on DSP56800E

```
    move.w      #0,y1                   ; 1 cycle, 1 word
    ...
    tst.w       a                       ; 1 cycle, 1 word
    ...
    jgt         APOS                    ; 5/4 cycles, 2 words
    move.w      #$0100,y1               ; cycles, 2 words
APOS
    ...
    move.w      #0,x0                   ; 1 cycle, 1 word
    tst         b                       ; 1 cycle, 1 word
    jgt         BPOS                    ; 5/4 cycles, 2 words
    move.w      #$0100,x0               ; 2 cycles, 2 words
BPOS
    ...
    move.w      x0,b1
    ...
    eor.w       y1,b                    ; 1 cycle, 1 word
    ...
    tst         b                       ; 1 cycle, 1 word
    jeq         TANOK                   ; 5/4 cycles, 2 words
    neg         a                       ; 1 cycle, 1 word
TANOK
; DSP56800E optimized ported code: 23-25 cycles / 13 words
```

In the code that was written from scratch, a better sequence was obtained. See Code Example 29. Note that this sequence does not use new DSP56800E features.

Code Example 29. DSP56800E Code Written from Scratch

```
    move.w      a,y1                    ; 1 cycle, 1 word
    ...
    move.w      b,c1                    ; 1 cycle, 1 word
    ...
    eor.w       c1,y1                   ; 1 cycle, 1 word
    bge         SAME_SIGN               ; 5/4 cycles, 1 word
    neg         a                       ; l cycle, 1 word
SAME_SIGN
; DSP56800E written from scratch code: 8 cycles / 5 words
```

Another difference between code written from scratch and optimized code is that the latter uses conditional transfers wherever possible instead of using conditional jumps, that take a higher number of cycles to execute. Code Example 30 presents this.

Code Example 30. Using Conditional Transfers Instead of Conditional Jumps

```
    move.w      #$0400,x0               ; 2 cycles, 2 words
    sub         b,a                     ; 1 cycle, 1 word
    jge         POS                     ; 5/4 cycles, 2words
    move.w      #$fc00,x0               ; 2 cycles, 2 words
POS
    ... use x0
; DSP56800 optimized ported code: 8/9 cycles / 7 words

    move.w      #$0400,b                ; 2 cycles, 2 words
    ...
    move.w      #$fc00,y0               ; 2 cycles, 2 words
    sub         a,d                     ; 1 cycle, 1 word
    tgt         y0,b                    ; 1 cycle, 1 word
    ... use b1
; DSP56800E written from scratch code: 6 cycles / 6 words
```

Note that, unlike the other instructions, Jcc and Bcc take approximately the same number of cycles on both DSP56800 and DSP56800E. It is recommended to replace them with Tcc wherever possible.

5 Pipeline Effects on DSP56800E

DSP56800E has a different pipeline structure, with more pipeline stages as compared to DSP56800. This explains the different pipeline effects of these two DSPs. Both DSPs have pipeline dependencies which can be met (especially AGU dependencies). DSP56800E introduced a few pipeline dependencies that did

not occur on DSP56800 (specifically, data ALU pipeline dependencies and hardware looping dependencies). Also, DSP56800E eliminated additional dependencies, such as, loading an address register with an immediate value and using it to address the next immediate instruction.

The DSP56800E core handles the pipeline dependencies in two different manners:

- In most cases a hardware interlock automatically causes stalls of the DSP56800E pipeline. The assembler can warn the programmer about these cases.
- There are a few cases when the core does not stall (for example, modification of N3 or M01 and using them to address in the next immediate instruction or hardware looping dependencies). The assembler can insert NOPs and warn the programmer about this insertion, or it can report an error.

Because of the new types of pipeline dependencies, the code ported from DSP56800 can stall in some cases. There are examples of data ALU or AGU pipeline effects on DSP56800E in the V.22 bis code. The DSP core automatically inserts stalls in these cases and the code executes correctly. However, many cycles are lost during these stalls, so the dependencies that generate them should be removed.

Special attention must be made to dependencies that involve hardware looping. Generally, the ported code could contain these new types of dependencies, which were not an issue for DSP56800. In the selected application example these dependencies are not met.

It is assumed that readers are familiar with Chapter 10, "*Instruction Pipeline*" from the *Core Reference Manual*. Several pipeline dependencies and methods to avoid them are illustrated in the selected code.

5.1 Data ALU Pipeline Dependencies

Because of the pipeline structure of DSP56800E, a few pipeline dependencies can occur for data ALU instructions, dependencies that did not occur on DSP56800. The reason they occur is that the "Execute" stage in the DSP56800 was broken into four stages in the DSP56800E pipeline; Address Generation, Operand Prefetch 2, Execute and Operand Fetch, and Execute 2.

The data ALU of DSP56800E can cause pipeline dependencies, when one of the three following conditions occurs:

- The result of a data ALU instruction executed in the "Late" state (Execute 2) is used in the instruction that immediately follows as a source register in a move instruction.
- The result of a data ALU instruction executed in the "Late" state is used in the two-stage instruction that immediately follows as a source register to a multiplication or multi-bit shifting operation. A dependency does not occur if the result is used in an accumulation, arithmetic, or logic operation in the instruction that immediately follows.
- An instruction requiring condition codes, such as Bcc, is executed immediately after a data ALU instruction is executed in the "Late" state.

When a data ALU dependency occurs, core interlocking hardware automatically stalls the core for 1 cycle to remove the dependency, affecting the execution time of a sequence of instructions, but not the correctness of the results.

Data ALU pipeline dependencies occur in many code sequences in the ported V.22 bis application. Although they do not affect the correctness, they introduce extra stall cycles. Code Example 31 is taken from function RXEQUD (file rx_equpd.asm).

Code Example 31. Data ALU Pipeline Dependency in DSP56800E Ported Code

```
n1: macr        x0,y0,b    a,x:(r3)+   ; the result B available after Ex2
n2: move        b,x:(r2)+              ; data ALU pipeline dependency
n3: move        x:(r3),a               ;
; 4 cycles / 3 words
```

Between instruction *n2* and *n1* is a data ALU pipeline dependency. Because the result becomes available in B after the Execute 2 phase, the *n2* instruction must stall 1 cycle to be able to write the B content in the memory. Four cycles are needed for execution of the sequence and can be rewritten as shown in Code Example 32.

Code Example 32. Removing Data ALU Pipeline Dependency

```
n1: macr        x0,y0,b    a,x:(r3)+   ; the result b available after Ex2
n2':move        x:(r3),a               ; B is not used in this instruction,
                                       ;    the dependency was removed
n3':move        b,x:(r2)+
; 3 cycles / 3 words
```

The data ALU pipeline dependency was removed. The core does not stall, thus the sequence is executed in three cycles instead of four.

Considering that data ALU pipeline dependencies occur most frequently in ported applications, identifying the code with data ALU dependencies and avoiding this code increases execution speed. To identify the pipeline dependencies, the programmer must fully understand the structure and behavior of the pipeline, and must give special attention when writing new code sequences.

5.2 AGU Pipeline Dependencies

The types of AGU pipeline dependencies on the DSP56800E represent almost all AGU dependencies that occur on the DSP56800, however the behaviors of the two cores differ for a similar dependency. When one of the conditions presented below occurs on DSP56800E, hardware interlocks are generated and the core automatically stalls the pipeline 1 or 2 cycles. The stalls can be avoided by introducing one 2-cycle instruction or two 1-cycle instructions after the instruction that generates an AGU dependency. On DSP56800 a single instruction is needed to remove an AGU dependency.

A dependency occurs if the same register is used within the next two instructions cycles that immediately follow and if the register is:

- Used as a pointer in an addressing mode
- Used as an offset in an addressing mode
- Used as an operand in an AGU calculation
- Used in a TFRA instruction.

Consideration must be given to dependencies caused by the modification of the N3 or M01 registers by a move or bit-manipulation instruction because the core does not automatically stall the pipeline in these cases. Additionally, a bit-manipulation operation performed on the N register does not automatically stall the pipeline.

There are some special cases where there are no AGU dependencies. For instance, there is no dependency when immediate values are written to the address pointer registers, R0–R5, N, and SP. Similarly, there are no dependencies when a register is loaded with a TFRA instruction. DSP56800 has more restrictions regarding the AGU pipeline dependencies than does DSP56800E.

There can be situations when a sequence, which did not have dependencies on DSP56800, introduces one stall on DSP56800E (the reason was explained at the beginning of this subsection). Code Example 33 taken from function tx_sbit (file tx_enc.asm) presents a situation of this type.

Code Example 33. Code Without AGU Pipeline Dependencies on DSP56800

```
n1: move        y1,x:>tx_quad          ; Store tx_quad
n2: add         b,a                    ; Get the actual address of variable
n3: move        a,r1                   ;   in r1
n4: nop                                ; Necessary to avoid dependency on DSP56800
n5: move        x:(r1)+,a1             ; Get the variable
```

On DSP56800 the NOP introduced in instruction *n4*, avoids the pipeline dependency. On DSP56800E, there are 2 cycles needed to avoid the pipeline dependency, therefore the dependency remains even though a NOP was introduced and the core will stall 1 cycle. Seven cycles are needed to execute this sequence on DSP56800E. Removing the NOP does not influence the execution time. Moving instruction *n1*, which is performed in 2 cycles, instead of *n4*, reduces the number of cycles to five. The code sequence is presented in Code Example 34.

Code Example 34. AGU Pipeline Dependency Avoided in DSP56800E Optimized Code

```
n2: add         b,a                    ; Get the actual address of variable
n3: moveu.w     a,r1                   ;   in r1
n4':move.w  y1,x:>tx_quad              ; Store tx_quad
n5: move.w  x:(r1)+,a1                 ; Get the variable
```

Avoiding the AGU pipeline dependencies provides an opportunity to improve the speed of the ported application and to improve the size of code. Because the DSP56800 applications usually contain inserted NOPs to avoid the dependencies, NOPs can be removed from the DSP56800E code.

5.3 Dependencies with Hardware Looping

Other dependencies, which did not appear on DSP56800, are those regarding the hardware looping. They occur when the LC register is loaded prior to executing one of the hardware looping instructions (DO, DOSLC, or REP). Because of the architecture of the instruction pipeline, none of the hardware looping instructions can be executed immediately after a value is placed in the LC register.

In V.22 bis there were no dependencies of this type, but on occasion they could appear in ported code.

6 Converting Applications for Increased Data and Program Memory

DSP56800E provides extended data memory space (24-bit data addresses instead of 16-bit) and extended memory space (21-bit program addresses instead of 16-bit). However, for a program that was written for the DSP56800 family to use DSP56800E extended memory, it is necessary to perform certain changes in the source code. These modifications are not always “automatic” and require a careful inspection of all source code.

This section describes these modifications, which are performed on a DSP56800E application (obtained by porting DSP56800 code), to make use of the extended data and program memory.

The examples are from a small application which uses the state machine from the original modem and performs scrambling and descrambling over a number of nibbles.

Note that the application does not require such a large amount of data and program memory. So both program and data memory must be forced to use extended memory by two “ORG” directives placed before all the code and all the data declarations.

6.1 Extending Data Memory Size From 64K to 16M

There are two assembler switches that instruct the DSP56800E application to use more than 16 bits for addresses: -od21 and -od24. Following these instructions, all addresses will become 24 bits long instead of 16 bits. Source code changes must be made to support this.

Instructions that are forced by the '>' operator to use 16-bit data addresses must be forced with the new '>>' operator to use 24-bit addresses. Code Example 35 displays the use of the force operator.

Code Example 35. Using the 24-Bit Force Operator Instead of 16-Bit Force Operator

```
    move        a1,x:>buffer          ; forced to use 16-bit address
; DSP56800 original code
    move.w      a1,x:>>buffer         ; forced to use 24-bit address
; DSP56800E ported code
```

Instructions that load addresses wider than 16 bits into address registers must also be modified, as shown in Code Example 36.

Code Example 36. Loading 24-bit Immediates into Address Registers

```
    move        #buffer,r1
; DSP56800 original code
    move.l      #buffer,r1
; DSP56800E ported code
```

The memory storage size for pointers to data memory must also be extended from 16 bits to 32 bits as shown in Code Example 37. Care must be taken to ensure that storage location is 2 word aligned.

Code Example 37. Extending Memory Storage Size for Pointers to Data Memory

```
pointer         ds     1
;DSP56800 original code
pointer         dsm    2
;DSP56800E ported code
```

Instructions that store the values of address registers in memory must also be modified to store not only 16 bits, but also 24 bits. These modifications are shown in Code Example 38.

Code Example 38. Saving 24-bit Values from Address Registers

```
    move        r1,x:pointer          ; save LSP 16 bits of r1
; DSP56800 original code
    move.l      r1,x:pointer          ; save all 24 bits of r1
; DSP56800E ported code
```

All the instructions that access these memory locations must be changed as shown in Code Example 39.

Code Example 39. Modifying Instructions to Access Memory on 32 Bits

```
    inc         x:pointer             ; increment the word at x:pointer
; DSP56800 original code
    inc.l       x:pointer             ; increment the 2-word value at x:pointer
; DSP56800E ported code
```

Note that on more complex applications which are ported to use data memory space above 64K require more complex modifications. For example, 16-bit arithmetic on pointers must be replaced with 32-bit arithmetic.

A summary of the extended data memory size for this project is presented in the Table 9.

Table 9. Summary of Extended Data Memory Size

Extended Data Memory	Size (Words)		Speed (Cycles)
	Data	Program	
Initial version	828	316	195246
Data memory extended	832	345	199865
Increase (percentage)	+0.48	+9.17	+2.36

Code size increased and speed decreased, however the differences are insignificant. This behavior was expected because the instructions that use address memory of 24 bits are coded with more words compared to the same instructions that use 16-bit memory addresses. Additionally, the instructions need extra cycles to perform.

6.2 Extending Program Memory Size From 64K to 2M

In porting applications from the DSP56800 platform to the DSP56800E platform, one coding recommendation is to not use the program memory space above 64K. However, here are some issues related to this feature if the programmer needs to use it.

In the selected example the program addresses were forced to 21 bits by using the assembler switch –op21. Also, relocation counters for program memory were initialized with addresses higher than 64K with an ORG p: directive.

The original code, which stores routine pointers in 1-word storage locations, had to be changed. This pointer array was transformed into a long pointer array. To be accessed with long word instructions, this array had to be 2-words aligned and is shown in Code Example 40.

Code Example 40. Extending Storage Size For a Pointer Array

```
RXQ     ds     25
; DSP56800 original code
RXQ     dsm    2
        ds     24*2
; DSP56800E ported code
```

Access to these pointers was achieved using word instructions in the DSP56800 original code. The pointers must be accessed using long word instructions (the array where they are stored should be 2-word aligned). This is presented in Code Example 41.

Code Example 41. Accessing 21-Bit Program Addresses

```
    move        #RX_dummy,a          ; getting the routine pointer
    move        a,x:(r0)+            ; storing the routine pointer
; DSP56800 original code
    move.l      #RX_dummy,a          ; getting the routine pointer
    move.l      a10,x:(r0)+          ; storing the routine pointer
; DSP56800E ported code
```

Another issue is the change of flow instructions. Instructions such as JMP or JSR can perform a change of flow to an address contained in a register. This feature was not available on DSP56800 platform and it was substituted by a technique that used the stack and RTS instruction. Basically, the address of the routine was placed on the stack together with SR and then an RTS instruction was executed.

The Code Example 42 is extracted from rx_ctrl.asm.

Code Example 42. DSP56800 Original Code

```
rx_next_task
    lea         (sp)+
    move        x:>RxQ_ptr,r3          ; Restore the RxQ pointer
    incw        x:RxQ_ptr              ; Increment the RxQ_ptr.
    move        x:(r3),x0              ; Get the address of next task
    move        x0,x:(sp)+             ; Push the address of task to be
    move        sr,x:(sp)              ;  performed onto the stack
    rts                                ; Perform task
; DSP56800 original code
```

This code is functional on the DSP56800E platform within the 64K program memory boundary. The problem occurs when the program memory is extended over 64K. The upper bits of a program address are stored in the SR register. The previous code will not work on the DSP56800E platform. It should be replaced with the specialized instruction available only on DSP56800E: JMP (n). This is presented in Code Example 43.

Code Example 43. DSP56800E Modified Code

```
rx_next_task
    moveu.w     x:>RxQ_ptr,r3
    inc.w       x:RxQ_ptr
    moveu.w     x:(r3),n
    jmp         (n)                    ; 3 cycles less
; DSP56800E ported code
```

At this stage, the code is still not ready to run on the extended program memory because the data width stored into the R3 register is 16 bits. The Code Example 44 presents all the corrections required by this program memory extension.

Code Example 44. DSP56800E Code That Allow Program Memory Access Beyond 64K

```
    move.l      x:>RxQ_ptr,r3          ;Pointer stored in memory has 32b
    adda        #2,r3,n                ;Long arithmetic: added 2 words
    move.l      n,x:RxQ_ptr            ;Storing back the pointer
    move.l      x:(r3),n               ;Reading the address for jump
    jmp         (n)                    ;Performing the jump
; DSP56800E ported code
```

A summary of the extended program memory size for this application is presented in Table 10.

Table 10. Summary of Extended Program Memory Size

Extended Program Memory	Size (Words)		Speed (Cycles)
	Data	Program	
Data memory extended (initial)	832	339	169145
Program memory extended	862	355	185292
Increase (percent)	+3.6%	+4.7%	+9.5%

The initial code was the code modified to support data memory extended addresses, presented in Section 6.1. This code was optimized using JMP (N) instead of the initial jump to subroutine mechanisms. This method of optimization is responsible for the difference between the speed of **169,145** cycles and **199,865** cycles of the ported code from the Section 6.1.

The explanation for the program memory increase and speed decrease is that every instruction that accepts X:xxxx when encoded to X:xxxxxx is 1 word longer and takes 1 extra cycle.

7 Conclusions

This application note investigated the process of porting an application developed for DSP56800 to the DSP56800E and the methods to optimize the ported code using the new features of DSP56800E. Also, the methods to optimize selected ported functions were analyzed and compared to redesigning and rewriting the functions.

Porting an existing DSP56800 code to DSP56800E is almost a direct process because the assembly code is compatible. There are certain requirements the code must meet to comply, but in normal applications they are not usually an issue. The only exception is the “MAC Output Limiter,” but that can also be corrected. The ported code runs on DSP56800E in almost half the number of cycles and uses nearly the same program size. Some pipeline effects can occur on the ported code, however these do not influence the correctness of the results. Only the execution time (in cycles) is slightly longer than half the number of cycles of the DSP56800 original code. Also, the program memory size of the ported application is slightly larger than the original. For the selected application, the number of cycles decreased from 61,918,898 to 31,694,501 cycles, however, because the DSP56800E processor runs at a higher clock frequency, the actual time is much shorter. This corresponds to a decrease from 6.73 MCPS to 3.43 MCPS in the processing load.

Additional speed improvement can be achieved by performing methods to optimize the ported code, by making use of the new DSP56800E features. Most of these methods can be done easily without a deep understanding of the algorithm and the overall code. The new features introduced by DSP56800E, which are most useful in this process, are additional registers, the extended set of data ALU operations, increased flexibility of the instruction set, AGU arithmetic, and hardware support for nested looping. In the example presented in this note, all of these features were used in different selected functions. The overall processing load improvement was from 3.43 MCPS to 3.13 MCPS—that is, about 10 percent. Achieving this improvement is realistic for general applications. The code of the optimized version was slightly smaller (about 2 percent). However, these methods of optimizing preserved the original code structure, as designed for DSP56800. If code is written from scratch, designed directly for DSP56800E, some of the new features can be exploited on a larger scale (for example, extended register set, more flexibility of the instruction set, new data types and AGU arithmetic). On selected examples, total improvements between 22 percent and 30 percent less cycles were obtained.

In summary, the following rules of thumb are presented:

- Unmodified DSP56800 code ported to DSP56800E generally takes half the number of clock cycles.
- Modification can further improve performance:
  - Local optimizations result in 10 percent clock cycle improvement.
  - Code rewrite may result in 20-30 percent clock cycle improvement.

Regarding the new pipeline structure, the original DSP56800 code runs directly, giving correct results. However, there are situations when code that did not violate pipeline restrictions on DSP56800 creates dependencies on DSP56800E. The core resolves these dependencies by introducing stalls (as in the case of data ALU dependencies). If the assembler signals these situations, the programmer can rearrange the code and eliminate the stalls, increasing speed even more.

In certain cases it might be necessary to extend the application making full use of DSP56800E addressing capabilities. The process of extending a ported application beyond the 16-bit boundary for program and data was analyzed. Usually this is not a straightforward process. However, if a new DSP56800E application is designed from scratch for this purpose, there are absolutely no problems in using the whole addressing space.

This application note proved that using of the new DSP56800E in existing DSP56800 applications is quite direct and brings performance improvements. These applications run in half the number of cycles

compared to DSP56800. Also new optimization methods can be introduced to further increase the performance. Moreover, the new DSP is faster than the older (120 MHz versus 35 MHz) and this means that the actual execution time is much shorter. Being a processor which can be defined as low-cost, low-power, and mid-performance computing, and which combines DSP power and parallelism with microcontroller programming simplicity, the DSP56800E is recommended for a large range of embedded applications.

Appendix A
Functions Written from Scratch

A.1 Optimized Ported Version of RXDEMOD

```
RXDEMOD

    move.l      #BPF_OUT,r3             ; Init. pointer to demod inputs
    move.l      #RXCB2A,r2              ; Init. pointer to demod outputs
    move.l      #MOD_TBL,r0             ; Load address of carrier freq.
                                        ;  table.
    move.l      #SIN_TBL,r5             ; keep SIN_TBL in r5
    move.l      #DPHASE,r4              ; and DPHASE in r4
    move.w      x:CDP,d                 ; load CDP and keep it in d

    do          #12,end_rx_demod        ; Loop 12 times
    moveu.w     #$80ff,m01              ; r1 is set to mod 256 mode of
                                        ;  addressing
    tfr         d,a                     ; Load CDP from d
    add.w       x:(r4),a
    move.w      #$0080,y0               ; Load constant for DPHASE >> 8
    move.w      a1,x:(r4)               ; Save DPHASE
                                        ; Note : if DPHASE overflows  then
                                        ;  the modulo value is stored
                                        ;  DPHASE is kept in r4
                                        ; REM & OFFSET
    mpy         a1,y0,a                 ; rem = DPHASE%256 in a0
                                        ; and offset = DPHASE>>8 in a1
    move.w      a0,y1                   ; Save the fractional part
    bfclr       #$ff00,a                ; Truncate offset to 8 LS bits
                                        ;  which is also a modulo 256
                                        ;  calculation
    lsr.w       y1                      ; shift to get into 1.15 format
                                        ; SINPHI & COSPHI
    moveu.w     a1,r1
    adda        r5,r1                   ; Load the address register with the
                                        ;  correct location in the 256  point
                                        ;  sine table.
    moveu.w     #$40-1,n                ; Load offset register
    move.w      x:(r1)+,a               ; sine1 = SIN_TBL(offset)
    move.w      x:(r1)+n,b              ; sine2 = SIN_TBL(offset+1)
    sub         a,b                     ; sine2-sine1 in y0
    move.w      b1,y0                   ;
    macr        y1,y0,a     x:(r1)+,b   ; sinphi =  sine1+(sine2-sine1)*rem
                                        ; cos1 = SIN_TBL(offset+$40)
    move.w      x:(r1)+,c               ; cos2= SIN_TBL(offset+$40+1)
    sub         b,c                     ; cos2-cos1 in y0
    moveu.w     #11,m01                 ; Set r0 to mod 12 addressing  mode
                                        ; -SIN & COS
    move.w      c1,y0
    macr        y1,y0,b     x:(r0)+,y0  ; cosphi = cos1+(cos2-cos1)*rem
                                        ; Get cosw from memory
    moveu.w     x:mod_tbl_offset,n      ; Load offset to MOD_TBL
    move.w      b,c                     ; Saturate the output
    mpyr        a1,y0,b     x:(r0)+n,y1 ; sinphi*cosw
                                        ; Get -sinw from memory
    macr        c1,y1,b                 ;-SIN = -sinw*cosphi+cosw*sinphi
    tfr         b,x0                     Save -SIN
    move.w      y0,n
    move.w      y1,y0                   ; Get -sinw
    move.w      n,y1                    ; Get cosw
    mpy         c1,y1,b   x:(r3)+,y1    ; cosw*cosphi in b
                                        ; Get X
    macr        -a1,y0,b                ; COS = sinw*sinphi+cosw*cosphi
```

```
                                        ; DEMODULATE
    move.w      b,b                     ; Saturate the output
    mpy         b1,y1,a     x:(r3)+,y0  ; X*COS in a
                                        ; Y in y0
    macr        -y0,x0,a                ; X*COS-Y*-SIN
    mpy         y1,x0,a     a,x:(r2)+   ; X*-SIN in a
                                        ; Get Y
    macr        b1,y0,a                 ; Y*COS+X*-SIN in a
                                        ; this register combination for
                                        ; macr is allowed on 56800e
    move.w      a,x:(r2)+               ; Save demodulated output
    moveu.w    #$ffff,m01              ; r0 in linear addr. Mode
end_rx_demod
End_RXDEMOD
    jmp         rx_next_task            ; Go to next task
```

A.2 RXDEMOD Written from Scratch

```
RXDEMOD
    move.l      #BPF_OUT,r4             ; load #BPF_OUT
    move.l      #RXCB2A,r3              ; load #RXCB2A
    move.l      #MOD_TBL,r0             ; load #MOD_TBL
    move.l      #SIN_TBL,r5             ; load #SIN_TBL
    moveu.w     x:mod_tbl_offset,r2     ; load md_tbl_ofset
    move.w      x:DPHASE,y1             ; keep DPHASE in y1
    moveu.w     #$80ff,m01              ; set 256 modulo for r1
    moveu.w     #$0040-1,n3             ; preload this constant in N3

    do          #12,end_loop            ; execute 12 times
    add         x:CDP,y1                ; DPHASE += CDP
    move.w      #$0080,x0
    mpy         y1,x0,a
    move.w      a1,r1
    move.w      a0,y0
    lsr.w       y0
    zxta.b      r1
    adda        r5,r1                   ; add #SIN_TBL to offset
                                        ; r1 contains #SIN_TBL + offset
    moveu.w     n3,n                    ; use preloaded #$0040-1
    move.w      x:(r1)+,a               ; a1 <- sin1
    move.w      x:(r1)+n,x0             ; x0 <- sin2
    sub         a1,x0                   ; x0 <- sin2 - sin1
    macr        x0,y0,a     x:(r1)+,c   ; a  <- sin1 + (sin2 - sin1) * rem
                                        ; c1 <- cos1
                                        ; a contains sinphi
    move.w      x:(r1),x0               ; x0 <- cos2
    sub         c1,x0                   ; x0 <- cos2 - cos1
    macr        x0,y0,c                 ; c  <- cos1 + (cos2 - cos1) * rem
                                        ; c contains cosphi
    moveu.w     r2,n                    ; use preloaded mod_tbl_offset
    moveu.w     #11,m01                 ; Set r0 to mod 12 addressing mode
    move.w      x:(r0)+,x0              ; x0 <- cosw
    move.w      x:(r0)+n,y0             ; y0 <- (-sinw)
    move.w      c,b1                    ; saturate cosphi
                                        ; a1 contains sinphi
                                        ; b1 contains cosphi
    mpy         b1,x0,c                 ; c <- cosphi * cosw
    macr        -y0,a1,c                ; c <- c - (-sinw* sinphi)
                                        ; c contains COS
    mpyr        x0,a1,d                 ; (d) -SIN += cosw * sinphi
    macr        y0,b1,d                 ; (d) -SIN = (-sinw)*cosphi
                                        ; d contains -SIN
    move.w      x:(r4)+,x0              ; x0 <- X
    move.w      x:(r4)+,y0              ; y0 <- Y
    move.w      c,c1                    ; saturate COS
    move.w      d,d1                    ; saturate –SIN
    mpy         x0,c1,a                 ; (a) tmp1 = X * COS
    macr        -y0,d1,a                ; (a) tmp1 -= Y*(-SIN)
    mpy         x0,d1,b                 ; (b) tmp2 = X * (-SIN)
    macr        y0,c1,b     a,(r3)+     ; (b) tmp2 += Y * COS
                                        ; RXCB2A(l++) <- temp1
    moveu.w     #$80ff,m01              ; 256 modulo for r1
    move.w      b,(r3)+                 ; RXCB2A(l++) <- temp2
end_loop
    move.w      y1,x:DPHASE             ; save DPHASE
    moveu.w     #$ffff,m01              ; set linear addressing
End_RXDEMOD
    jmp         rx_next_task
```

A.3 Optimized Ported Version of RXEQERR

```
RXEQERR

    move.w      #$0,n
    moveu.w     #EQX,r3
    moveu.w     #DECX,r0
    moveu.w     #DP,r1
    moveu.w     #DX,r2                  ; Get the hard decision value of I
                                        ; Get the hard decision value of Q
                                        ; Get the soft decision value of I
    move.w      x:(r0)+,y0 x:(r3)+,x0
    tfr         x0,b       x:(r0)+n,y1
    sub         y0,b                    ; Calculate DX=EQX-DECX
    mpy         x0,y1,a    x:(r3)+n,x0 ; Calculate EQX*DECY
    macr        -x0,y0,a                ; Calculate DP=EQX*DECY-EQY*DECX
    tfr         x0,a       a,x:(r1)+n   ; Store the calculated DP
    sub         y1,a       b,x:(r2)+    ; Calculate DY=EQY-DECY
                                        ; Store DX
    move.w      b,y0
    mpy         y0,y0,b    a,x:(r2)+    ; Calculate DX*DX
                                        ; Store DY=EQY-DECY
    move.w      a,y0
    mac         y0,y0,b                 ; Calculate DX*DX+DY*DY
    asl         b                       ; Compute 2*(DX*DX+DY*DY)
                                        ; Compute 4*(DX*DX+DY*DY)
    asl         b          x:(r2)+n,x0  ; bring x:>NOISE in x0
    move.w      #$7000,y0
    mac         x0,y0,b                 ; NOISE=4*(DX*DX+DY*DY)+
                                        ; $7000*NOISE
    move.w      b,x:(r2)+n              ; Store the accumulated NOISE

EQUD22
    move.w      #0,y1                   ; If the phase error is positive,
                                        ;  y1 will be set to zero. Else
                                        ;  it will be set to $8000
    move.w      x:(r1)+n,a              ; Get the phase error
    tst         a                       ; If the phase error is zero then
                                        ;  skip normalization process
    jeq         CAR_NOR                 ; If the phase error is negative
    jgt         APOS                    ; If the phase error is negative
    move.w      #$0100,y1               ;  set y1 to $8000
APOS
    deca        r3                      ; resetting the address registers
    deca        r0
    move.w      x:(r0)+,y0 x:(r3)+,x0   ; Get EQX,DECX
    mpy         x0,y0,b    x:(r0)+n,y0  x:(r3)+,x0
                                        ; Compute EQX*DECX
                                        ; Get EQY,DECY
    macr        x0,y0,b                 ; Calculate realp=EQX*DECX+EQY*DECY
    move.w      #0,x0                   ; If realp value is +ve set x0 to
                                        ;  zero
    tst         b                       ; Test whether realp value is +ve
                                        ;  or not
    jgt         BPOS
    move.w      #$0100,x0               ; If found -ve set x0 to $8000
BPOS
    abs         b                       ; Compute |realp|
    move.w      b,y0                    ; Move |realp| into a register for
                                        ;  the division operation
    abs         a                       ; Compute |DP|
    cmp         a,b                     ; Check whether |realp|>|DP| or not
    move.w      x0,b1                   ; Get the sign information of realp
                                        ;  in b1
    jgt         DIVID                   ; If the divisor > dividend let the
                                        ;  division takes place
    eor         y1,b
    jmpd        TSTSGN                  ; use delay slots
    move.w      #$0400,a
DIVID
    eor         y1,b                    ; Computation to determine the sign
                                        ; of the phase error
    bfclr       #$0001,sr               ; Set the carry bit clear
    rep         #11
    div         y0,a
    move.w      a0,a
TSTSGN
    tst         b                       ; Determine the sign of the phase
    jeq         TANOK                   ; error and set accordingly
    neg         a
TANOK
    move.w      a,x:(r1)+n              ; Store the normalized phase error
```

```
CAR_NOR
                                        ; PHASE ERROR UNWRAP ROUTINE
    move.w      x:WRPFLG,b              ; Get the WRPFLG
                                        ; Get the phase error
                                        ; Test the WRPFLG
    tst         b           X:(R1)+N,a
                                        ; If WRPFLG >= 0 goto _start
    jge         start                   ; If WRPFLG >= 0 goto _start
    clr         a                       ; else set WRAP = 0 and
    move.w      a,x:WRAP                ; LASTDP = 0 and go to
    jmpd        rx_next_task            ; next task
    move.w      a,x:>LASTDP
start
    move.w      x:LASTDP,b              ; Get LASTDP
    move.w      a,y1                    ; Store DP in y1
    move.w      #$0400,x0               ; Set temp = $0400
    sub         b,a                     ; Compute DP-LASTDP
                                        ; If DP-LASTDP < 0
    jge         POS                     ; If DP-LASTDP < 0
    move.w      #$fc00,x0               ; set temp = $fc00
POS
    abs         a                       ; Compute |DP-LASTDP|
    move.w      #$0400,y0
    move.w      x:WRAP,b                ; Get the value of WRAP
    cmp.w       y0,a                    ; Compare |DP-LASTDP| and
                                        ; $0400
    jlt         NOWRAP                  ; If |DP-LASTDP|<$0400 then
                                        ; jump to _NOWRAP
    sub         x0,b                    ; else WRAP = WRAP-temp
    move.w      b,x:WRAP
NOWRAP
    move.w  y1, x:LASTDP                ; Set LASTDP = DP
_Ecar22_nor
End_RXEQERR
    jmpd        rx_next_task
    add         y1,b                    ; DP=DP+WRAP and go to next task
    move.w      b,x:(r1)+n
```

A.4 RXEQERR Written from Scratch

```
RXEQERR

    move.l      #EQX,r0                 ; r0 points to EQX
    move.l      #DECX+1,r3              ; r3 points to DECY
    move.l      #DX,r2                  ; r2 points to DX
    move.w      #0,n

    move.w      x:(r0)+,y0 x:(r3)-,x0   ; y0 <- EQX, x0 <- DECY
    mpy         x0,y0,a    x:(r0)+n,y1 x:(r3)+,c
                                        ; a  <- DECY * EQX
                                        ; y1 <- EQY
                                        ; c  <- DECX
    sub         c1,y0                   ; y0 <- EQX - DECX [DX]
    macr        -c1,y1,a                ; a  <- a - DECX * EQY [DP]
    sub         x0,y1                   ; y1 <- EQY - DECY [DY]
    mpy         y0,y0,b    y0,x:(r2)+   ; b  <- DX * DX
                                        ; store DX
    mac         y1,y1,b                 ; b  <- b + DY * DY
    asl         b          y1,x:(r2)+   ; b  <- b * 2
                                        ; store DY
    asl         b          x:(r2)+n,y0  ; b  <- b * 2
                                        ; y0 <- NOISE
    move.w      #$7000,y1               ; y1 <- $7000
    mac         y0,y1,b                 ; b  <- b + $7000 * NOISE [NOISE]
    tst         a
    move.w      b,x:(r2)+n              ; store NOISE
    beq         UNWRAP
    move.w      a,y1                    ; copy a in y1 for comparing signs
    move.w      x:(r0)-,y0              ; y0 <- EQY
    mpy         y0,x0,b    x:(r0)+,y0   ; b  <- EQY * DECY
                                        ; y0 <- EQX
    macr        y0,c1,b                 ; b <- b + EQX * DECX [realp]
    move.w      b,c1                    ; ; copy b in c1 for comparing signs
    abs         b                       ; |realp|
    abs         a                       ; |DP|
    cmp         a,b                     ; |DP| ? |realp|
    bgt         DO_DIV
    jmpd        SET_DP
```

```
    move.w      #$0400,a            ; tmp <- #$0400
DO_DIV
    move.w      b,x0
    bfclr       #$0001,sr
    rep         #11
    div         x0,a
    move.w      a0,a
SET_DP
    eor         c1,y1               ; compare signs of realp and DP
    bge         SAME_SIGN           ; if they have the same sign
    neg         a
SAME_SIGN
                                    ; a contains DP
UNWRAP
    tst.w       x:WRPFLG
    bge         next_2
    jmpd        final
    clr         c                   ; WRAP   <- 0
    clr         d                   ; LASTDP <- 0
next_2
    move.w      x:WRAP,c            ; c <- WRAP
    move.w      x:LASTDP,d          ; d <- LASTDP
    move.w      #$0400,b            ; b <- #$0400 [temp]
    move.w      b,x0                ; store #$0400 for further use
    move.w      #$fc00,y0           ; y0 <- #$fc00
    sub         a,d                 ; d <- LASTDP - DP
    tgt         y0,b                ; if LASTDP - DP >0 then b <- #$fc00  [temp]
    abs         d                   ; d <- | LASTDP - DP|
    clr         y0
    cmp.w       d,x0                ; |DP - LASTDP| ? #$0400
    tgt         y0,b                ; if |DP - LASTDP| < 0 then temp <- 0
    sub         b,c                 ; c <-  WRAP - temp
    move.w      a,d                 ; LASTDP <- DP
    add         c,a                 ; DP <- DP + WRAP
final
    move.w      c,x:WRAP            ; store WRAP
    move.w      d,x:LASTDP          ; store LASTDP
End_RXEQERR
    jmpd        rx_next_task
    move.w      a,x:>DP             ; store DP
```