TMS320C31 DSP Programming

D. W. Hawkins
dwh@ovro.caltech.edu
Document Revision: 1.14
July 7th, 2005

Contents
1 Introduction 3
2 DSP Features 3
3 Programmers Model 5
4 DSP Memory Map and Data Types 8
5 Compiler Features 11
6 C and Assembler Interfacing 14
7 Interrupt Handling 18
8 $\mu$COS-II RTOS Port 25
  8.1 Port Description ....................................... 28
    8.1.1 OS_CPU.H ....................................... 28
    8.1.2 OS_CPU.C.C ..................................... 28
    8.1.3 OS_CPU.A.ASM ................................... 29
    8.1.4 OS_CPU.CMD ..................................... 30
    8.1.5 BSP.H,.C ....................................... 30
  8.2 Port Testing ........................................... 30
    8.2.1 Task-to-ISR context switching ......................... 30
    8.2.2 Task-to-task context switching ........................ 30
    8.2.3 Interrupt nesting .................................. 31
    8.2.4 Small-memory model (data-page) ......................... 32
9 Memory Transfer Benchmarking 33
10 Data Processing Benchmarking 37
   10.1 Integer-to-floating point conversion .................... 37
   10.2 Data normalization .................................... 37
   10.3 Data accumulation .................................... 39
   10.4 FFT .................................................. 40
   10.5 Complex-valued exponential calculation .................. 41
   10.6 Real and complex-valued vector multiplication ............ 45
   10.7 TI-to-IEEE float-point format conversion ................. 47
11 Peripheral Programming

11.1 Memory-mapped peripherals .............................................. 48
11.2 Bit-fields ........................................................................... 49
11.3 Peripheral Control Library .................................................. 51
11.4 Code Composer memory manipulation ................................... 51
11.5 I/O Flags register (XF[1..0] control) ................................... 51
11.6 Timers .................................................................................. 52
11.7 Serial Port ............................................................................ 53
11.8 DMA Programming .............................................................. 54
1 Introduction

The Texas Instruments (TI) TMS320LC31 is a Digital Signal Processor (DSP), that was state-of-the-art circa 1996. These DSPs are used on the digitizer and correlator boards in the COBRA Correlator System. The C31 DSP is described in detail in the TMS320C3X Floating Point DSP User’s Guide (I have a hard-copy of Rev. D of this guide, SPRU031D dated Oct 1994, while Rev. F, March 2004, is available in PDF format) [5,13]. The COBRA CVS area contains an HTML index page with links to copies of about 30 TI PDF-format publications relevent to the DSP. This document describes the key programming features of the DSP.

Code for the C31 DSP is generated using tools from Texas Instruments. The tool-suite consists of a fairly standard set of components; assembler, C-compiler, linker, archiver, etc., and a graphical IDE called Code Composer [12]. Code Composer can be used in conjunction with a TI-specific emulator interface that exists on the DSP (the C31 is a pre-JTAG device), to probe the DSP registers and debug applications. The DSP emulator hardware interface is referred to as the XDS510. The COBRA project has a TI XDS510 ISA board [10], had a PCMCIA board (its dead) from White Mountain DSP (now part of Analog Devices and no longer supporting the TI devices), and most recently a Signum Systems parallel port device. The emulator is invaluable for booting boards with no code, debugging boards with hardware issues, and testing with the DSP development kit, the DSK [6] (which has no non-volatile memory for program storage).

2 DSP Features

The basic features of the C31 are (for the full feature list, see the data sheet [11] or user’s guide [13]):

- 33MHz to 40MHz operation at 3.3V.
- One synchronous serial port (SSP).
- Two timers (with outputs that can be configured as general purpose I/O pins).
- An external interface bus.
- A single-channel DMA coprocessor
- 2 blocks of 1Kx32 words of on-chip RAM
- Multiple on-chip buses allowing for parallel CPU and DMA operations.
- 2 General purpose I/O pins (XF0, XF1)
- Four external Interrupts

Figure 1 shows the architecture of the TMS320C31 DSP.

On the COBRA boards, the DSP synchronous serial port (SSP) is configured as an asynchronous RS-232 serial port. The COBRA Revision C boards (the ones produced for use in the OVRO/SZA/CARMA correlators) use the SSP to implement an SPI interface to an asynchronous RS-232 UART. Earlier COBRA board revisions used software to emulate an RS-232 UART. The RS-232 interface is used for debugging of the boards, eg. on the benchtop. The two DSP timer outputs are routed to the COBRA system controller for general purpose use (one timer is used by the DSP RTOS, the other is unused). The DMA coprocessor is used to perform data transfers. The two general purpose I/O pins are connected to two front panel LEDs to allow visual feedback during code operation or debug. These two pins are controlled via writes to the I/O Flags (IOF) register. The external interrupts are routed to the system controller to allow for general use and to ensure correct interrupt pulse timing (an interrupt input should only be asserted for two clock periods of the DSP H1 clock [5, p6-23]).
Figure 1: The TMS320C31/TMS320LC31 DSP [11].
Table 1: TMS320LC31 Registers [13].

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0-R7</td>
<td>Extended-precision registers 0-7</td>
</tr>
<tr>
<td>AR0-AR7</td>
<td>Auxiliary registers 0-7</td>
</tr>
<tr>
<td>DP</td>
<td>Data-page pointer</td>
</tr>
<tr>
<td>IR0</td>
<td>Index register 0</td>
</tr>
<tr>
<td>IR1</td>
<td>Index register 1</td>
</tr>
<tr>
<td>BK</td>
<td>Block size register</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>ST</td>
<td>Status register</td>
</tr>
<tr>
<td>IE</td>
<td>CPU/DMA interrupt enable</td>
</tr>
<tr>
<td>IF</td>
<td>CPU interrupt flags</td>
</tr>
<tr>
<td>IOF</td>
<td>I/O flags</td>
</tr>
<tr>
<td>RS</td>
<td>Repeat start address</td>
</tr>
<tr>
<td>RE</td>
<td>Repeat end address</td>
</tr>
<tr>
<td>RC</td>
<td>Repeat counter</td>
</tr>
</tbody>
</table>

3 Programmers Model

The C31 DSP has a RISC-like load-and-store programming architecture with hardware support for DSP operations. Relevant features are:

- A CPU register file containing 28 registers:
  - 8 extended-precision 40-bit registers, R0-R7, for floating-point and integer operations.
  - 8 auxiliary registers, AR0-AR7, for integer operations and addressing modes
  - 12 other specific-purpose and general-purpose registers.

- A hardware floating-point/integer multiplier (32-bit floating-point input, 40-bit result, 24-bit integer input, 32-bit result).

- An arithmetic-logic unit (ALU).

- Barrel shifter.

- Two auxiliary register arithmetic units (ARARUs) (used for address calculations).

Table 1 gives the list of C31 registers. Chapter 2 and 3 of the C3x User’s Guide provides detailed descriptions of each register [13]. All of the primary registers can be operated upon by the multiplier ALU and can be used as general-purpose registers. The registers also have some special functions. For example, the eight extended-precision registers are used for maintaining extended-precision floating-point results. The eight auxiliary registers support a variety of indirect addressing modes and can be used as general-purpose 32-bit integer and logical registers. The remaining registers provide such system functions as addressing, stack management, processor status, interrupts, and
block repeat (p2-9 [13]). There are two additional registers not accessible to the programmer; the program counter (PC) and instruction register (IR) (p2-12 [13]).

**SP Stack pointer**
The C31 stack pointer is a 32-bit register that contains the address of the top of the stack. The SP always points to the last element pushed onto the stack. A push performs a preincrement; a pop performs a postdecrement of the stack pointer. This type of stack is called a "full-ascending" (i.e., grows in the direction of higher memory, and points to the last item pushed). For additional details see Chapter 6, p6-29, of the User's Guide [13].

**ST Status register**
The status register contains information on the state of the CPU. The result of a load, store, arithmetic, or logical operation sets or clears the status register condition flags. The status register can be loaded and stored, allowing the register to be saved and restored. Figure 2 shows the status register bits, while Table 2 gives the bit definitions.

**DP Data-page pointer**
The data-page pointer is a 32-bit register. The 8-LSBs of this register are used by the direct addressing mode as a pointer to the page being addressed, i.e., as address bits [23:16], with the 16-bit immediate address bits corresponding to address bits [15:0]. The data-page pointer is loaded using the LDP instruction. The bits [23:16] of the source operand are stored into the bits [7:0] of the data-page register (p13-136 [13]). Given an 8-bit data-page pointer, and a 16-bit offset, there are a total of 256-pages, each of length 64K. In the small-memory model, the data page is fixed (by the processor initialization code), and .bss and the .const sections must lie within a 64K page.

The C31 supports the following general instruction addressing modes:

- **Register.**
  The operand is a CPU register.

- **Short immediate.**
  The operand is a 16-bit (short) or 24-bit (long) immediate value.

- **Direct.**
  The operand is the contents of a 24-bit address formed by concatenating the 8 bits of data-page pointer and a 16-bit operand.

- **Indirect.**
  An auxiliary register indicates the address of the operand.

These and other addressing modes are summarized on p2-12 [13], and detailed in Chapter 6 [13].
Figure 2: The TMS320C31/TMS320LC31 DSP status register (p3-5 [11]).

Table 2: The TMS320C31/TMS320LC31 DSP status register bit definitions (p3-6 [11]).

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Reset Value</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>Carry flag</td>
<td>Carry condition flag</td>
</tr>
<tr>
<td>V</td>
<td>0</td>
<td>Overflow flag</td>
<td>Overflow condition flag</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>Zero flag</td>
<td>Zero condition flag</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
<td>Negative flag</td>
<td>Negative condition flag</td>
</tr>
<tr>
<td>UF</td>
<td>0</td>
<td>Floating-point</td>
<td>Floating-point underflow flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>underflow flag</td>
<td></td>
</tr>
<tr>
<td>LV</td>
<td>0</td>
<td>Latched overflow</td>
<td>Latched overflow condition flag</td>
</tr>
<tr>
<td>LUF</td>
<td>0</td>
<td>Latched floating-point</td>
<td>Latched floating-point underflow flag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>underflow flag</td>
<td></td>
</tr>
<tr>
<td>OVM</td>
<td>0</td>
<td>Overflow mode flag</td>
<td>Overflow mode flag</td>
</tr>
<tr>
<td>RM</td>
<td>0</td>
<td>Repeat mode flag</td>
<td>Repeat mode flag</td>
</tr>
<tr>
<td>CF</td>
<td>0</td>
<td>Cache freeze</td>
<td>Controls freezing of the cache.</td>
</tr>
<tr>
<td>CE</td>
<td>0</td>
<td>Cache enable</td>
<td>CE enables or disables the instruction cache.</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
<td>Cache clear</td>
<td>Invalidate all entries in the cache.</td>
</tr>
<tr>
<td>GIE</td>
<td>0</td>
<td>Global interrupt</td>
<td>Enable/disable CPU interrupts.</td>
</tr>
</tbody>
</table>
mapped peripherals, and a bootloader. The DSP also contains a 64 × 4 [13]). The DSP contains two on-chip 1K-word RAM blocks (RAM0 and RAM1), a set of memory mapped peripherals, and a bootloader. The DSP also contains a 64 × 32-bit instruction cache that stores often-repeated sections of code. This cache increases performance and allows for parallel operations (e.g., DMA from external memory to an internal RAM block, while the CPU is executing instructions from the cache and manipulating data in the other internal RAM block).

4 DSP Memory Map and Data Types

The C31 DSP uses a 24-bit address bus, and a 32-bit data bus. The DSP can access a total of 16M (million) 32-bit words. Most C31 data types are 32-bits wide, i.e., a char, short, int, float, and double are all 32-bits (p3-4 [9]). The long double floating-point data type uses 40-bits. Storage of a long double requires two memory locations. Floating-point operations in the registers R0 through R7 use the 40-bit representation (an extra 8-LSBs are added to the fractional part of the 32-bit floating-point number). As soon as a float or double is stored to memory, a 32-bit representation (the 32-MSBs of the 40-bit register value) is saved. The floating-point format is a TI-specific format, and differs from the IEEE 754 format.

Figure 3 shows the C31 predefined memory map for microcomputer/bootloader mode (see Ch. 4 [13]). The DSP contains two on-chip 1K-word RAM blocks (RAM0 and RAM1), a set of memory mapped peripherals, and a bootloader. The DSP also contains a 64 × 32-bit instruction cache that stores often-repeated sections of code. This cache increases performance and allows for parallel operations (e.g., DMA from external memory to an internal RAM block, while the CPU is executing instructions from the cache and manipulating data in the other internal RAM block).
Table 3: PCI memory map and DSP external memory map.

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000h</td>
<td>1M</td>
<td>SDRAM</td>
</tr>
<tr>
<td>400000h</td>
<td>512K</td>
<td>Flash RAM (8-bit)</td>
</tr>
<tr>
<td>600000h</td>
<td>256K</td>
<td>FPGAs</td>
</tr>
<tr>
<td>700000h</td>
<td>128K</td>
<td>SRAM</td>
</tr>
<tr>
<td>780000h</td>
<td>1K</td>
<td>Board Control Registers</td>
</tr>
<tr>
<td>781000h</td>
<td>1K</td>
<td>1-Wire Buffer RAM</td>
</tr>
<tr>
<td>782000h</td>
<td>1K</td>
<td>Scratch RAM</td>
</tr>
<tr>
<td>783000h</td>
<td>1K</td>
<td>Digitizer Module</td>
</tr>
<tr>
<td>784000h</td>
<td>1K</td>
<td>DSP Bus Latches</td>
</tr>
<tr>
<td>785000h</td>
<td>1K</td>
<td>Monitor ADC</td>
</tr>
<tr>
<td>N/A</td>
<td>1K</td>
<td>PLX-9054 Control Registers</td>
</tr>
<tr>
<td>N/A</td>
<td>16K</td>
<td>PCI Master-to-PCI Memory</td>
</tr>
<tr>
<td>N/A</td>
<td>16K</td>
<td>PCI Master-to-PCI I/O or Configuration Space</td>
</tr>
</tbody>
</table>

The COBRA boards define a 2M x 32-bit region of custom memory and memory-mapped peripherals. This memory region can be accessed by the DSP as an external memory access, or by the host as a PCI bus access. Note that the host cannot access the internal memory of the DSP (the C31 does not support it). Table 3 and Figure 4 show the 2M x 32-bit memory map of the correlator and digitizer boards (on a correlator board, the digitizer module registers are not available, and this region is considered reserved). The PCI and DSP memory maps have the same layout; the PCI base address is determined by the PCI BIOS on boot, whereas the DSP base address is always 300000h. The DSP base address was chosen to be 300000h, as this places the Flash RAM at the DSP boot address of 400000h. Additional information on the system controller implementation, and the control register bit meanings can be found in the digitizer and correlator board engineering specifications, and in the system controller HDL documentation.

On the COBRA boards, the DSP is configured in microcomputer/bootloader mode, and INT1 is asserted on reset. This causes the DSP bootloader to initiate program load from address 0x400000 (the Flash RAM) (p11-3 [13]). The Flash RAM on the COBRA boards is 8-bits wide (with each location taking up a 32-bit word). The COBRA system controller can also overlay the SDRAM to location 0x400000. This allows the host to download a DSP program to SDRAM, select boot from SDRAM, and then reset the DSP. The host typically uses this feature for loading the DSP, and the DSP Flash is typically programmed with a serial port debug program (for benchtop use). The hex30 program can be used to convert DSP object files into bootloader compatible files. Object files are converted to 8-bit width for Flash programming, and 32-bit width for SDRAM download. Note that the DSP bootloader will fail if a board is booted from a blank Flash. A blank Flash has all words set to 0xFF, and the bootloader incorrectly interprets this as a valid memory width setting since it only tests individual bits (see the bootloader code in Appendix B [13]). If the bootloader fails, then the DSP external bus arbiter stops working, and since the DSP Flash exists on the DSP external bus, a host request for the bus fails, and the host cannot program the Flash. The solution to this is to first boot the DSP from SDRAM, then use the host to program the DSP Flash.
Figure 4: COBRA PCI memory map and DSP external memory map (Note: a dword is a 32-bit word). (a) shows the 8MB (2M-Dword) memory map, while (b) shows a more detailed view of the 512KB (128K-Dword) control register memory map. The PCI and DSP memory maps have the same layout; the PCI base address is determined by the PCI BIOS on boot, whereas the DSP base address is always 300000h (this places the Flash RAM at the DSP boot address of 400000h). Note: the PLX control registers, and PCI Master regions are only visible at the locations shown in (b) for the DSP memory map.
5 Compiler Features

The COBRA DSPs are generally programmed in C, and any functions that require optimization are written as C-callable assembler. When the DSP boots, an assembly coded startup routine initializes the C-runtime environment. The TI DSP tools provide the runtime setup code, and an implementation of the standard C library called the runtime support library (p8-60 [8]). The source for the startup and library code is archived in rts.src. A precompiled version of the library is typically linked with C-code, eg. rts30.lib is linked against COBRA code. No customization of the startup code, boot.asm, contained within the archive rts.src was required.

The DSP assembler is described in the TMS320C3x/C4x Assembly Language Tools User’s Guide [8], while the DSP C-compiler is described in the TMS320C3x/C4x Optimizing C Compiler User’s Guide [9]. The TMS320C3x General-Purpose Applications User’s Guide [7] has examples of processor initialization, and processor usage.

The following processor initialization options, and compiler features, are relevant to the COBRA software:

• Define the heap (the runtime support library uses it).

  The DSP tools use an uninitialized section called .sysmem for the heap. The size of the heap can be set size using the -heap command line option (the Code Composer IDE has an entry box under Project, Options, Linker). The default size is 1K (p8-61 [8]).

  The linker file is used to setup the address of the heap. The linker creates the .sysmem section only if there is a .sysmem section in an input file. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it a value equal to the size of the heap (p8-11 [8]).

• Define the stack.

  The DSP tools use an uninitialized section called .stack for the stack. The size of the stack can be set size using the -stack command line option (the Code Composer IDE has an entry box under Project, Options, Linker). The default size is 1K (p8-61 [8]).

  The linker file is used to setup the address of the stack. When the linker creates the .stack section, it also creates a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the stack (p8-15 [8]).

• The DSP tools can use two models for setting up initialized sections; RAM and ROM models (p4-36 [9]).

  In the RAM model, it is up to the loader to initialize the memory region where initialized variables are loaded, while in the ROM model, the assembler startup code (in boot.asm) copies initialized variables from ROM to their modifiable location in RAM (p8-61 [8]).

  The ROM model is used by the COBRA boards (the Code Composer IDE has a pull-down selection under Project, Options, Linker).

• The DSP tools support two memory models for data access; the small-memory model, and the big-memory model (p2-35, 4-5 [9], Appendix C [7]).

  In the small-memory model, the data-page pointer is initialized to the page of the .bss section (by boot.asm), and is not modified by the compiler after that point. All external variables, global variables, static variables, and compiler-generated constants must link into a single 64K data page (p2-35 [9]), i.e., the sum of the .bss and .const sections must not exceed, and must not cross, a 64K page boundary (the linker script can be used to ensure 16-bit alignment) (p2-70, p4-3 [9]).

1 The boot code can be extracted from the archive using: ar30 x rts.src boot.asm
In the big-memory model, there are no addressing restrictions, as the compiler reloads the data-page pointer before each access to a global or static variable (p5-4 [7]). Because of this additional instruction before every data access, the big-memory model should generally be avoided.

The size of the `.bss` section can be minimized, allowing the use of the small-memory model, by placing large data structures, or arrays, in their own sections, and accessing them through pointers. The `#pragma DATA_SECTION` can be used in C code to create a new data section, eg., see p5-14 [7], p3-13 [9]. The linker file is then used to place the new section into the output executable.

The C-compiler sets the macro `_BIGMODEL` to 1 if the big-memory model is used (p2-30 [9]). The assembler generates the corresponding symbol `.BIGMODEL` (p4-24 [9]).

- The DSP tools support two argument calling conventions for C-code; the register-argument model, and the stack-based model (the default) (p2-34 [9]).

The standard runtime model is to use the stack-based calling convention, where all arguments are passed on the stack. The register-argument model passes some arguments directly in registers. Details of the conventions for both models can be found in Section 4.4, p4-15 [9].

The C-compiler sets the macro `_REGPARM` to 1 if the register-argument model is used (p2-30 [9]). The assembler generates the corresponding symbol `.REGPARM` (p4-22, p4-24 [9]).

- Other C-compiler features and nuances:
  - The `cregister` keyword allows C code to directly access DSP registers (p3-7 [9]).
  - The `#pragma INTERRUPT` can be used to identify a C-function as an interrupt service routine (p3-16 [9]).
  - The TI tools do not clear the `.bss` section on boot (the default `boot.asm` does not zero the section). (p3-18 [9]).

- Default sections created by the assembler (default COFF object file sections, p2-2 [8], p4-3 [9]):

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.text</code></td>
<td>Usually contains executable code</td>
</tr>
<tr>
<td><code>.data</code></td>
<td>Usually contains initialized data</td>
</tr>
<tr>
<td><code>.bss</code></td>
<td>Usually contains uninitialized data</td>
</tr>
</tbody>
</table>

- Sections created by the compiler (p2-69, p4-2 [9]):

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.bss</code></td>
<td>Uninitialized</td>
<td>Global and static variables</td>
</tr>
<tr>
<td><code>.cinit</code></td>
<td>Initialized</td>
<td>Initialization values for explicitly initialized global and static variables</td>
</tr>
<tr>
<td><code>.const</code></td>
<td>Initialized</td>
<td>Global and static constant variables that are explicitly initialized and string literals</td>
</tr>
<tr>
<td><code>.stack</code></td>
<td>Uninitialized</td>
<td>Stack</td>
</tr>
<tr>
<td><code>.text</code></td>
<td>Initialized</td>
<td>Executable code and floating-point constants</td>
</tr>
<tr>
<td><code>.sysmem</code></td>
<td>Uninitialized</td>
<td>Memory for malloc functions</td>
</tr>
</tbody>
</table>
• Symbols created by the linker (p8-53 [8]):

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.text</code></td>
<td>Start of the <code>.text</code> output section</td>
</tr>
<tr>
<td><code>etext</code></td>
<td>The first address after the end of the <code>.text</code> output section</td>
</tr>
<tr>
<td><code>.data</code></td>
<td>Start of the <code>.data</code> output section</td>
</tr>
<tr>
<td><code>edata</code></td>
<td>The first address after the end of the <code>.data</code> output section</td>
</tr>
<tr>
<td><code>.bss</code></td>
<td>Start of the <code>.bss</code> output section</td>
</tr>
<tr>
<td><code>end</code></td>
<td>The first address after the end of the <code>.bss</code> output section</td>
</tr>
</tbody>
</table>

The DSP tools can be setup to generate a `.map` file, which contains all the symbols created by the linker and program code.
6  C and Assembler Interfacing

The general philosophy for embedded programming is to code in C first, then create assembly-coded optimized functions for performance-critical sections of code if testing (profiling) shows this is necessary. C-callable assembly-coded routines are also required when implementing functions that access processor-specific registers that are not supported by the C language. Interrupt handling also typically requires a small amount of assembler to implement processor context saving and restoring.

The DSP tools have a couple of additions to the C language; the `cregister` keyword allows C code to directly access DSP registers (p3-7 [9]), and the `#pragma INTERRUPT` can be used to identify a C-function as an interrupt service routine (p3-16 [9]). These two features are useful, but not required. The same functionality can be implemented using C-callable assembler routines. In the μCOS-II RTOS port used on the COBRA boards, the `cregister` keyword is used to manipulate the I/O flags register (`IOF`) to toggle the LEDs attached to the I/O flags pins, and is used to manipulate the interrupt enable (`IE`) register.

The DSP tools use two memory calling conventions; the small-memory model and the big-memory model. The two models differ in how data variables are accessed. The small-memory model is the default model, and is adequate for use in the COBRA system. The DSP C runtime environment is detailed in Chapter 4 of the TMS320C3x/C4x Optimizing C Compiler User’s Guide [9]. The C compiler register conventions are on p4-11, where Table 4-1 in that guide lists the registers that the code generation tools assume to be preserved across calls. C-callable assembler routines need to adhere to the register conventions. Function structure and calling conventions are detailed on p4-15. Interfacing C with assembly code is covered on p4-22.

The simplest method for generating a C-callable assembly function is to write a C function declaration (prototype), and place it in a header file. Then write the C function body, and place it in a C-coded file (use a file name of the form `*_c.c`). Build the C code with the compiler tools, and enable the tools to generate an assembler listing. Copy the assembler listing file to a new file (use a file name of the form `*_a.asm`). Edit the assembler version, e.g. by replacing the C-compiler generated assembler with hand-optimized assembler (e.g. replace sections of code with parallel DSP operations). Then write a test program. Link first with the C-coded version of the program and test, then with the assembler version of the program and test again. One useful test for benchmarking is to call the function inside an infinite loop while toggling an external I/O pin each time through the loop. This generates a square-wave on the I/O pin that can be measured using an oscilloscope. The performance of the C-coded function versus the assembler function can then be compared for various options; cache on/off, linked to internal/external RAM, etc.

Consider the following header that defines a function that adds two integer values, and returns an integer sum;

```c
/* sum.h */
int sum(int a, int b);
```

with the corresponding C-coded implementation

```c
/* sum_c.c */
int sum(int a, int b)
{
    return a+b;
}
```
Compilation of this code with optimization level 2, and full inlining enabled gives the (edited) assembler code listing (from sum.c.asm)

```assembly
fp .set ar3
.text
.global _sum
_sum:
push fp
ldiu sp,fp
ldiu *-fp(3),r0
ldiu *-fp(2),r1
addi3 r0,r1,r0
ldiu *-fp(1),r1
ldiu *fp,fp
subi 2,sp
bu r1
```

Explanation:

- The frame-pointer (explained shortly) is pushed onto the stack.
- The frame-pointer is set equal to the stack pointer.
- Register r0 is loaded with an integer value, using the frame-pointer for relative addressing on the stack, the value is that of b (explained shortly).
- Register r1 is loaded with the value of a.
- addi3 adds a and b and stores the result to r0 (which is the register used for the return variable).
- Register r1 is loaded with the return address.
- The frame-pointer is loaded with the old frame pointer that was stored onto the stack on function entry.
- The stack pointer is adjusted (moving it back past the frame-pointer and return address).
- A branch is performed to the return address saved in r1.

Figure 5 shows the DSP stack frame during a function call. In the small-memory model, function arguments are pushed onto the stack in reverse order (p4-16 [9]). In our example above, this means that the caller of \texttt{sum}(a, b) would push b, and then a onto the stack, and then execute a CALL assembler instruction. The CALL instruction automatically places the return address, i.e., the next program counter (PC), onto the stack. On entry into the function, register AR3 is initialized for use as the frame-pointer, and the stack-pointer is moved to accommodate any local variables (none in the example above). Indirect addressing of the frame-pointer is then used to access arguments (below the frame-pointer), and local variables (above the frame-pointer). In the example above, the frame-pointer points at the old frame-pointer, below that is the return address, and then the last argument placed on the stack, a, followed by the first argument placed on the stack, b. Hence, \texttt{ldiu -*-fp(3),r0 loads b into r0, and ldiu -*-fp(2),r1 loads a into r1. The register r0 is used to return integers, floating-point numbers, and pointers (p4-13, p4-20 [9]). A function returns either by way of the RETS instruction, or by branching to the return address as shown in the example above. The choice between the two options depends on whether a delayed branch can be used, with the delay slots being used for restoring registers and the frame-pointer (p4-20 [9]).}
Before
CALL

Push arguments,
call function

Allocate
local frame

Saved
registers

Local
frame

Old frame
pointer

Return
address

Argument 1

Argument n

Caller's
local frame

Return
address

Argument 1

Argument n

Caller's
local frame

Caller's
local frame

FP

SP

FP

SP

Low

High

(a)  

(b)  

(c)

Figure 5: DSP stack use during a function call. (a) shows the stack prior to the function call, (b)
shows the stack once arguments have been pushed onto the stack, and the function call made, and
(c) shows the stack after the function prolog has completed.

Given the compiler generated assembler code, and the knowledge of the calling conventions, we
can now write our own assembler version of _sum:

; sum_a.asm
fp .set ar3
.text
.global _sum
_sum:

; setup frame-pointer
push fp
ldiu sp,fp

; load a into r0
ldiu *-fp(2),r0

; load b into r1
ldiu *-fp(3),r1

; r0 = r0 + r1
addi3 r0,r1,r0

; restore the old frame-pointer
pop fp
This hand-coded version swaps the registers used to store \( a \) and \( b \), changes the exit sequence to pop the frame-pointer stored on the stack into the frame-pointer register, and uses the \texttt{RETS} instruction to return (this automatically pops the return address off the stack).

A test program consisting of a while loop that pulsed an LED high for every call to \texttt{sum}. Two command files were used; code in on-chip DSP RAM, code in off-chip SRAM. The following pulse high-times were recorded:

<table>
<thead>
<tr>
<th>Source</th>
<th>Optimization</th>
<th>Inlining</th>
<th>Cache</th>
<th>RAM</th>
<th>Time (( \mu s ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum_c_c</td>
<td>any</td>
<td>any</td>
<td>on/off</td>
<td>on-chip</td>
<td>3.8</td>
</tr>
<tr>
<td>sum_a_asm</td>
<td>any</td>
<td>any</td>
<td>on/off</td>
<td>on-chip</td>
<td>3.8</td>
</tr>
<tr>
<td>sum_c_c</td>
<td>any</td>
<td>full</td>
<td>on/off</td>
<td>off-chip</td>
<td>7.9</td>
</tr>
<tr>
<td>sum_c_c</td>
<td>disable</td>
<td>disable</td>
<td>on/off</td>
<td>off-chip</td>
<td>8.6</td>
</tr>
<tr>
<td>sum_a_asm</td>
<td>disable</td>
<td>full</td>
<td>on/off</td>
<td>off-chip</td>
<td>7.6</td>
</tr>
<tr>
<td>sum_a_asm</td>
<td>disable</td>
<td>disable</td>
<td>on/off</td>
<td>off-chip</td>
<td>8.3</td>
</tr>
</tbody>
</table>

Having code in off-chip memory causes a significant reduction in performance. Enabling the cache is supposed to help when accessing slow external memory, but for this test the cache memory did not really help (variations on the 100ns timescale were seen). Turning inlining on and off showed a small change (less assembly code gets generated). For this simple example, recoding the function into assembler did not help. However, the purpose of this example was to show how to create a C-callable assembly coded function, not code optimization.
7 Interrupt Handling

The C31 can respond to the following nine interrupt sources:

- Four external interrupts: INT0 through INT3
- Serial port transmit interrupt: XINT0
- Serial port receive interrupt: RINT0
- Timer 0 interrupt: TINT0
- Timer 1 interrupt: TINT1
- DMA interrupt: DINT

Figure 3(c) shows the interrupt vector locations for microcomputer/bootloader mode. The interrupt vectors and software TRAPs map to the bottom 63-locations of the DSP internal RAM bank 1.

Interruptions are enabled or disabled globally via the Global interrupt enable bit in the status register (see Figure 2, and Table 2). Enabling and disabling of the nine interrupts is controlled by the interrupt enable register, IE, and the status of an interrupt is indicated by the interrupt flags register, IF (see p3-9 and p3-11, p7-26 [13]). The bits in the interrupt flags register can be written to software-trigger an interrupt. The interrupt flags are not reset by the DSP boot-loader. The interrupt flags register should be cleared before enabling interrupts to avoid spurious interrupts (p11-13 [13]). The interrupts are prioritized with INT0 having the highest priority, and DINT the lowest (p7-31 [13]).

In the CPU interrupt processing cycle (eg. see Figure 7-6 on p7-34 [13]), the corresponding interrupt flag in the IF register is cleared, and interrupts are globally disabled (GIE = 0). The CPU completes all fetched instructions. The current PC is pushed to the top of the stack. The interrupt vector is then fetched and loaded into the PC, and the CPU starts executing the first instruction in the interrupt service routine (ISR). In microcomputer/bootloader mode, the interrupt vector contains a branch instruction to the start of the interrupt service routine (p7-26 [13]).

An interrupt service routine must preserve the C runtime environment by saving and restoring any registers it manipulates. In the case of running an RTOS on the DSP, all registers must be saved (including the full 40-bits extended precision registers R0 to R7) since an interrupt can make a higher-priority task ready, and a context switch to that task will occur on context restore. The ordering of registers during a context save and restore is important. The status register must be saved first, and restored last. An example of context save-and-restore routines can be found in Section 2.4, p2-11 [7].

Once an interrupt service routine has started, since the GIE bit is clear, a higher-priority interrupt will not be serviced. Interrupt nesting is implemented by saving the processor context, clearing the IE bits of lower-priority interrupts, and then setting the GIE bit in the status register to reenable interrupts (eg. see p2-10 [7]). Prior to context restore, the GIE bit is again cleared, context is restored, and a RETI instruction is performed. The RETI instruction sets the GIE bit, reenabling interrupts. This method implements context save and restore as non-interruptible blocks of code.

Non-interruptible blocks of code, or critical sections of code are needed when accessing variables shared between a task and an interrupt service routine, or when a sequence of operations must be completed without interruption. Critical section code is generally bracketed by two functions; enter critical section (disables interrupts), and exit critical section (enables interrupts). The DSP tools documentation provides several examples of critical section entry and exit functions via direct manipulation of the ST and/or IE register, however, testing of these methods showed that they do not work (a test for GIE clear within a critical section would fail a small fraction of the time). The issue with these methods has to do with the DSP pipeline. The only reliable way determined to disable interrupts was to call a TRAP that simply returned via RETS. A trap call clears the GIE bit.
like any other interrupt, and a RETS call does not set the GIE bit on return. Interrupts are enabled by a call to another TRAP routine that contains a RETI instruction. On return from the trap, the GIE bit is set, enabling interrupts.

The DSP C compiler can be used to write interrupt code directly. An interrupt service routine will be recognized if it has the name c_intXY(), where XY is 01, 02, ..., 99 (p4-30 [9]). The #pragma INTERRUPT can also be used to identify a function as an interrupt service routine (p3-16 [9]), eg. #pragma INTERRUPT(ISRINTO) followed by the function void ISRINTO(void) {} would identify that function as an ISR, and change its return statement to a RETI instruction. Once you have interrupt handler code, you still need to setup the interrupt vector branches using assembler code. For example, to install the C-coded function void ISRINTO(void) as the handler for external interrupt zero, the following assembler code would need to be linked with the application:

```
.ref _ISRINT0
.sect "INT0_vector"
INTO br _ISRINT0
```

and the linker command file would then need to place the branch at the correct vector address, eg. using a SECTIONS statement like

```
.INT0_vector:> 0x809FC1
```

Support for writing interrupt handlers is not really needed by the C compiler. In fact, its much easier to control the context save and restore sequence by writing an interrupt handler as; an assembler routine that saves the context, calls a handler function, and then restores the context. The handler function can be a standard C function. For example, the vector for INTO can branch to an assembler routine INTOISR that saves context, calls INTOHandler (a C-coded function), and then restores context. The following is the INTOISR ISR code;

```
; int0isr.asm
; -----------------------------------------------------------------
.text
; C-coded handler
.global _INTOHandler

; INTO Interrupt Service Routine
; -----------------------------------------------------------------
INTOISR:
; Context save
; Status register
PUSH ST
; Extended precision registers
PUSH R0
PUSHF R0
PUSH R1
PUSHF R1
PUSH R2
PUSHF R2
PUSH R3
PUSHF R3
PUSH R4
```
PUSH R5
PUSHF R5
PUSH R6
PUSHF R6
PUSH R7
PUSHF R7
; Auxiliary registers
PUSH AR0
PUSH AR1
PUSH AR2
PUSH AR3
PUSH AR4
PUSH AR5
PUSH AR6
PUSH AR7
; Remaining registers
PUSH IR0
PUSH IR1
PUSH BK
PUSH IE
PUSH RS
PUSH RE
PUSH RC

CALL _INT0Handler

; Context restore
POP RC
POP RE
POP RS
POP IE
POP BK
POP IR1
POP IR0
POP AR7
POP AR6
POP AR5
POP AR4
POP AR3
POP AR2
POP AR1
POP AR0
POPF R7
POP R7
POPF R6
POP R6
POPF R5
POP R5
POPF R4
POP R4
POPF R3
This handler code does not support interrupt nesting (\textit{ST(GIE)} remains cleared). A linker command file is used to place the \texttt{.INT0\_vector} section at 0x809FC1, and when an external interrupt occurs, the vector branches to \texttt{INT0ISR}, which saves the context, calls the C-coded handler, and restores the context. The context save and restore code is similar to that on p2-13 [7]. The registers are saved with the status register first, and the remaining registers are saved in the order of their machine register value (p3-2 [13]). The registers; \textit{IF}, \textit{IOF}, and \textit{DP} are not saved to the stack. The flags register is not saved, since you should normally not write to it (so you would not want to restore it), and the I/O flags register is not saved, since that register is for external I/O pin control and does not represent processor state relevant to task switching. In the small-memory model, the data-page pointer is initialized and then assumed unchanged. Any code that uses the \textit{DP} needs to be considered a critical-section, since if it is interrupted before it restores the \textit{DP}, an ISR can be entered with an invalid \textit{DP} (eg. the \texttt{ffft\_rl()} C-callable assembly routine from TI uses \textit{DP}, p6-42 [7]). Alternatively, in an RTOS, where each task has a separate context, the initial context should have the stack-saved version of the \textit{DP} initialized to \texttt{.bss}, and then the context restore will load it with the correct value (the address of the \texttt{.bss} can be accessed from C code using linker defined symbols, see p4-28 [9]).

To test the interrupt context switch time, an application was written that contained an infinite loop that sets an external I/O pin high, and writes to the interrupt flags register to trigger an \texttt{INT0}, i.e., writes 1 to \textit{IF}. A C-coded \texttt{INT0\_Handler()} function then clears the external I/O pin. The high-time of the I/O pin waveform indicated the context save time, while the low time indicated the context restore time. For code linked to on-chip memory the high-time was 5.5\,\mu s, and the low-time was 4.9\,\mu s. The asymmetry is likely due to the fact that the high-time measures the time the DSP takes to recognize the interrupt, finish processing instructions, jump to the vector, save context, and then call the C-coded handler. When the code was linked into off-chip SRAM, the high and low-times increased to 17.5\,\mu s and 13.6\,\mu s (the cache was on). The time required for a context switch (save and restore) is then about 31\,\mu s for a stack located in external SRAM (which is where the COBRA task stacks will be located). The DSPs operate with a 33MHz clock, so a context switch takes \sim 1000\,clock cycles. The COBRA boards will operate in CARMA with a phase switch signal of 1024pps. The overhead associated with a context switch every phase switch interrupt is 31\,\mu s/977\,\mu s \approx 3\%, so a context switch time of 31\,\mu s is acceptable.

If interrupt handlers are written for all interrupt sources, following the example for the \texttt{INT0} handler, then the context save and restore code is repeated multiple times. The context save and restore routines can be placed into separate procedures callable by each handler. The ISR entry code needs to perform a partial context save which includes one register, so that when the ISR performs a call to the context save routine, the context save routine can pop the return address into that register, and then finish the context save (an example is given below for nested interrupts).
For non-nested interrupts, the context restore sequence can be optimized by each handler using a delayed branch followed by a partial restore of RC, RE, RS, as this keeps the pipeline full. The context restore code for a non-nested handler looks like

```
CALL _INT0Handler

; Context restore
BD ContextRestore ; Delayed branch
POP RC
POP RE
POP RS
; Delayed branch occurs here
```

```
; -----------------------------------------------------------------
; Context restore
; -----------------------------------------------------------------
ContextRestore:

POPF R0
POPF R0
POPF ST
RETI
```

Interrupt handlers can implement interrupt nesting by:

- Saving context.
- Masking IE to clear the bits of all lower priority interrupts.
- Enable interrupts by setting ST(GIE).
- Call the handler.
- Disable interrupts.
- Restore context and return.

Since each ISR needs to use a different interrupt enable mask, the ISR entry sequence is different for each handler. The common sequence of storing registers can be put into a common block of code by using a call (or trap). Each handler entry would push the status register, and the extended registers R0 and R1. The handler would then load R0 with a mask value and call the context save routine. The context save routine would pop the return address from the stack into R1, finish the context save, and push the contents of R1 on last (i.e., put the return address back on the stack). The IE register would then be masked, and a RETI would take you back to the ISR to call the C-coded handler. For example, a nested ISR for INT0 would look like

```
; -----------------------------------------------------------------
; Interrupt nesting interrupt enable masks
; -----------------------------------------------------------------
INTOMASK .set 0
```
INT1MASK .set 1
INT2MASK .set 3
INT3MASK .set 7
ContextRestoreTrapNumber .set 27

; -----------------------------------------------------------------
; INTO Interrupt Service Routine
; -----------------------------------------------------------------
INT0ISR:
; Partial context save
; Status register
PUSH ST
; Extended precision registers R0 and R1
PUSH R0
PUSH R0
PUSH R1
PUSHF R1
; Pass the enable mask in R0
LDI INT0MASK, R0
CALL ContextSave

CALL _INT0Handler

; Context restore (with interrupts disabled)
TRAP ContextRestoreTrapNumber

; -----------------------------------------------------------------
; Context save
; -----------------------------------------------------------------
ContextSave:
; Save the return address in R1
POP R1

; Remaining context save
PUSH R2
PUSHF R2
PUSH R3
PUSHF R3

; Mask the interrupt enable register
AND R0, IE

; Put the return address back on the stack
; and return enabling interrupts
PUSH R1
RET

; Context restore
; ---------------------------------------------
ContextRestore:
    ; Remove the trap call return address from the stack
    SUBI 1, SP

    ; Context restore
    POP RC
    POP RE
    POP RS
    ...
    POPF R0
    POP R0
    POP ST
    RET

; ISR and trap vectors
; ---------------------------------------------

.sect "INT0_vector"
INT0 br INTOISR

.sect "TRAP27_vector"
TRAP27 br ContextRestore

Since INT0 has the highest priority, it does not need to reenable interrupts. To keep the code identical for all handlers, the mask for INT0 is simply 0, i.e., no interrupts are enabled. However, for INT1, the mask is 1, which enables INT0, so if an INT0 occurs while processing an INT1, the INT0 ISR will be triggered and will complete processing before returning back to the INT1 handler. The process is similar for the other lower-priority interrupts. The context restore trap number was arbitrarily chosen as TRAP 27. The linker command file needs to locate the trap correctly in the interrupt vectors table.

A test program was written that included ISR handlers for INT0 through INT3. The main routine contained an infinite loop that set an external I/O pin high for each ISR handler, and wrote 0xF to the interrupt flags register to trigger all four interrupt handlers. C-coded handlers then cleared a specific external I/O pin. With four interrupts generated, the I/O pin pulses followed the expected priority, i.e., all I/O pins were asserted high, and then they deasserted in order INT0 through INT3. Inversion of the interrupt priorities was tested by replacing the AND in the context save routine with an ANDN. For code linked into external RAM, with the cache on, when a single IF bit was set, the high-time of any handler I/O pin was 19.0µs, and the low-time was 14.3µs, i.e., the logic for nesting and reuse of common context save-and-restore increases the context switch time to 33.3µs, about 2.2µs more than the earlier example.
Figure 6: μCOS-II DSP task stack layout. (a) shows the stack as it would be just after a CALL to a function of the form `void task(void *pdata);` (b) shows the stack after the processor context is saved, and (c) shows the stack after an interrupt has returned, and the function prolog has completed.

8 μCOS-II RTOS Port

The MicroC/OS-II or μCOS-II real-time operating system (RTOS) was developed by Jean Labrosse for use in embedded systems such as microcontrollers and DSPs. The RTOS and methods for writing device drivers for it are covered in his two books; MicroC/OS-II: The Real-Time Kernel [1, 3], and Embedded Systems Building Blocks: Complete and Ready-to-Use Modules in C [2]. The first edition of the RTOS book covers version 2.00 of the RTOS, while the second edition covers version 2.52. The books contain the RTOS source code, and the RTOS can be used free-of-charge in university projects. The current commercial release of the RTOS is version 2.7x. The web site www.micrium.com contains additional resources, and ports for various processors.

Porting μCOS-II version 2.52 is covered in Chapter 13 of MicroC/OS-II: The Real-Time Kernel, 2nd Ed [3]. A μCOS-II port requires the definition of the data types on the processor, assembly language routines for critical section protection, interrupt handling, and context switching, and the definition of C coded hook functions. Table 13.1 on p289 [3] summarizes the porting requirements. The main effort involved in porting μCOS-II is to determine the processor programming model, and determine the calling conventions of the compiler. Earlier sections of this document developed this knowledge, so the port of μCOS-II is now straightforward.

A task in μCOS-II is defined as a function call of the form;

```c
void task(void *pdata);
```

where `pdata` is a pointer that can be used to pass information to a task. Tasks start out life as if they were just interrupted at the entry of a call to their task function. The port-specific C-function `OSTaskStkInit()` is responsible for creating an appropriate initial stack. Figure 6 shows stack layouts for various stages in the task entry sequence. In the DSP compiler, using the default
stack-passing argument model, if you were to call the function `task(void *pdata)`, `pdata` would be pushed onto the stack, and then a `CALL` instruction would be executed, which would place the return address on the stack. Figure 6(a) shows the stack for this case (remember, a `CALL` to the task function never actually occurs to create this). The return address placed on the stack would be used if the task function was ever returned from, so it is useful to place the address of an exit handler on the stack in that location. The exit handler can log the fact that a task exited (when it probably should not have).

Figure 6(b) shows the stack once the processor context has been saved to it. When a task is ready, and is the highest priority, the OS will execute a sequence that performs a context restore, as if the processor was returning from an interrupt. Since each task was really not interrupted, the contents of most of the registers saved onto the task stack can be arbitrary. It is common to place a set of well-known hexadecimal numbers into the stack frame (e.g. `0x01010101`, `0x02020202`, etc), so that a debugger can be used to confirm the correct stack locations were restored to the correct registers.

The DSP compiler supports two function calling schemes: stack-argument and register-argument calling schemes. If the DSP compiler was using the register-argument model, then it would have passed the argument `pdata` in register `AR2`. By placing `pdata` in the `AR2` location on the stack, both compiler calling conventions are easily supported by the port.

The DSP compiler supports two memory models; small-memory and big-memory. The difference between the two is that in the small memory model, all data is assumed to be within the immediate addressing range of the `.bss`, and so the `DP` register is initialized to the 8-bit page (bits 16 to 23) of the `.bss` inside `boot.asm` and after that point is not modified. The large-memory model makes no assumption about the data-page pointer and loads it prior to each data access. Given that the small-memory model never changes `DP`, the port could implement context save and restore routines that ignore `DP` for the small memory model, and save it for the large memory model, i.e., saving of `DP` can occur inside checks of `.BIGMODEL` in the C code, and `.BIGMODEL` in the assembly code.

The small-memory model assumption that the data-page pointer remains unchanged can be violated; code that uses the data-page pointer can save the data-page pointer (e.g. by pushing it onto the stack), and then restore it, before code that accesses data in the `.bss` executes. The FFT code, `ffft_rl()` (forward FFT, real-valued samples), supplied by TI is an example of code that modifies the data-page during its execution, but restores it before returning. Modification of the data-page pointer can cause a problem in a multi-tasking system. For example, a task performing an FFT could be interrupted, and when that interrupt service runs it will have the wrong data page! If the interrupt service routine never accessed data in the `.bss` (an unlikely scenario), then this would not cause problems. But if the interrupt made a higher-priority task ready, the context of that task would be restored, and the data-page would be wrong for that task too! The problem with the higher-priority task data-page pointer could be resolved by storing the data-page pointer as part of the processor context regardless of the memory model, and coding `OSTaskStkInit()` to initialize the data-pointer stored on the initial task stack to the data-page of the `.bss` (linker symbols can be used to determine the address of the `.bss` from C code). However, this solution does not solve the issue with the data-page pointer for an interrupt service routine that interrupts a task with a modified data-page. One solution to the problem is to treat the data-page pointer as a global (shared) variable, and to use critical sections around blocks of code that use a modified data-pointer. However, this solution would not be acceptable in the case of say, a large FFT that should be interruptible. The solution to the problem is that each task needs to have a copy of the data-page pointer as part of its context (since a task may modify it), and every interrupt service routine needs to set the data-page pointer inside the ISR code after a context save has been completed (since the data-page of the task needs to be saved).

Figure 7 shows the μCOS-II DSP task initial stack context. The registers that can take on arbitrary values are loaded with their register numbers (0 to 27) (p13-37 [13]) packed into a byte, and repeated four times (five times for the 40-bit extended-precision registers). The data page, `DP`,
is initialized to the page of the .bss, and pdata is passed on the stack (stack-calling convention) and in AR2 (register-calling convention). The status register value has the global interrupt enable (GIE) bit set, and the cache enabled. The setting of the GIE bit in the saved status register does not actually matter, since a return from interrupt (RETI) is executed during a context restore, and that instruction always sets GIE.

The registers missing from the context in Figure 7 are: IOF, IF, and IE. The I/O flags register is really just an I/O port, and its context is not task specific, so should not be saved. The interrupt flags register is generally not written to, a flag gets set by an interrupt, and cleared by the processor automatically when an ISR is entered. The interrupt enable register is not saved, since the register should be considered a global variable. Different tasks may enable different interrupt sources (by modifying the register inside a critical section), but the state of the register is not task specific. An

![Stack Context Diagram]

Figure 7: $\mu$COS-II DSP task initial stack context.
additional problem with saving IE as part of the task context, is that there is no way to determine the initial value of the register to place in the initial task stack context.

The interrupt enable register does need to be modified by the interrupt service routines (ISRs) to implement interrupt nesting. The DSP hardware prioritizes the external interrupt sources, so that if multiple interrupts are asserted at the same time, the highest priority interrupt is started first. The DSP clears the status register global interrupt enable bit when servicing interrupts, so if the ISR does not re-enable interrupts, higher-priority hardware interrupts that occur slightly after the ISR starts will be blocked until the ISR finishes. The ISR can enable interrupt nesting by saving the IE register setting onto the stack, clearing interrupts of lower-priority, updating the IE register, and then setting the GIE bit in the status register. If a higher-priority interrupt occurs, it will also push the IE register setting onto the stack, and clear lower-priority interrupts. As the nested interrupts unwind, each will pop the intermediate IE settings off the stack, until the first interrupt restores the original IE register setting. Once the nested interrupts are complete, a higher-priority task may be scheduled, and at this point the interrupt enable flags register is unchanged relative to the setting for the previously running task. The IE register can be considered interrupt-specific context, and all ISRs save and restore it. The implementation details are described in the next section.

8.1 Port Description

Porting µCOS-II version 2.52 is covered in Chapter 13 of MicroC/OS-II: The Real-Time Kernel, 2nd Ed [3]. The following sections comment on the implementation of the TMS320C31 port.

8.1.1 OS_CPU.H

Critical section protection

The TMS320C3x status register (ST) contains a global interrupt enable (GIE) bit that is used to enable or disable interrupts on the C3x. When the DSP receives an interrupt, this bit is cleared, and when the DSP executes a RETI, the bit is set. The register can not be safely read while the GIE bit is set due to pipelining issues on the DSP (read all about it in the C3x Users Guide), so critical methods 2 and 3 are difficult (if not impossible) to implement. Method #1 is the only method defined for this port.

Critical section entry is implemented by calling a TRAP to a RETS instruction. The TRAP is a software interrupt that automatically clears the GIE bit. Critical section exit is implemented by calling a TRAP to a RETI instruction. The RETI instruction sets the GIE bit on return. The trap implementations are in OS_CPU_A.ASM, and the vectors need to be installed into the correct vector location by the linker command script, eg. use OS_CPU.CMD as the basis for customized linker scripts.

Task-level context switch

The task-level context switch macro, OS_TASK_SW(), is defined as a call to OSCtxSw() (see the OS_CPU_A.ASM section).

8.1.2 OS_CPU.C.C

The only C function the port needed to define was OSTaskStkInit() to initialize the stack as shown in Figure 7. Read the source code to see how the linker defined symbol for the .bss start address is used to initialize the data-page pointer.
8.1.3 OS_CPU_A.ASM

A port requires the implementation of four assembler routines: OSStartHighRdy (start multi-tasking), OSCtxSw (task-level context switch), OSIntCtxSw (interrupt-level context switch), and OSTickISR (time-tick ISR). Figure 7 shows that 32-words need to be pushed onto the stack to save the processor registers. Since the context save and restore routines are used in multiple places, the port also defines context save and context restore routines.

Start multi-tasking

OSStartHighRdy() is called at the end of OSStart() (OS_CORE.C), and is the exit point from main()’s context into the RTOS. OSStart() does not call OSStartHighRdy() from within a critical section, so there is no protection from interrupts. OSInit() is called prior to OSStart(), and it does contain calls to OS_ENTER/EXIT_CRITICAL, and since this port uses Method #1, that will cause the global interrupt enable bit to be set. However, since the OS has not started yet, it would be an error for an interrupt to be serviced. The OS timer is expected to be setup in the first task, and interrupts enabled at that point (in this case via the interrupt enable register, IE).

Since interrupts should not occur when this function is called, the context restore function can be executed using a branch, instead of a TRAP.

Task-level context switch

OS_Sched() (OS_CORE.C) calls OS_TASK_SW() to implement a task-level context switch from inside a critical section. The macro OS_TASK_SW() is a call to OSCtxSw() in this port. The job of OSCtxSw() is to save the current task context, switch over to the higher-priority task, and then restore context. This whole sequence must take place with interrupts disabled.

The only difference between the context switch from task level relative to the interrupt-level context switch performed by OSIntCtxSw(), is that ISRs push a copy of IE onto the stack, and the context restore sequence for an interrupt needs to restore IE, and then the context of the task.

Interrupt-level context switch

Each ISR saves the processor context, adjusts the OSIntNesting counter, pushes the interrupt enable register onto the stack, adjusts the interrupt enable register for interrupt nesting, and then re-enables interrupts. An ISR calls handler code (written in C), and then calls OSIntExit().

OSIntExit() (OS_CORE.C) checks to see if interrupt nesting is over, and then if a higher-priority task is ready. If interrupts are still nested, or the same task has the highest priority, then OSIntExit() returns, and the ISR runs to completion (i.e., performs the context restore of the task or interrupt it interrupted). If however, interrupt nesting is over, and a higher-priority task has been made ready, then a switch to the new task is required; that is the job of OSIntCtxSw(). OSIntExit() calls OSIntCtxSw() inside a critical section, so interrupts are disabled when this function is called.

An interrupt-level context switch is slightly different than a task-level context switch, as the interrupt enable register saved to the stack needs to be restored. In the case of a nested interrupt, the context restore call comes as a consequence of passing through the entire ISR (since OSIntExit() does nothing in that case), and calling a TRAP that ends up at ContextRestoreTrap. In the case of the last interrupt, and a higher-priority task being ready, OSIntCtxSw() is called to perform the switch to the new task. Prior to exchanging stacks OSIntCtxSw() needs to restore IE from the stack. To locate the saved value of IE, the stack pointer needs to be re-loaded from the task control block of the task that is about to be switched.
Interrupt service routines (ISRs)

The DSP has nine interrupt sources. The OS timer is generated using timer 1. When a timer tick occurs, the ISR saves the processor context, increments the OSIntNesting variable, masks off interrupts of lower-priority, and then enables interrupts while calling the interrupt handler OSTimeTick. Once the handler completes, OSIntExit() is called and the processor context is restored (with interrupts disabled). This sequence is similar for all the other interrupt sources, the only differences being the interrupt mask used, and the handler called. ISRs are defined for all the DSP interrupt sources, and the board-support package (BSP) contains empty handlers. See the source code for the interrupt priority mask definitions and the implementation details.

8.1.4 OS_CPU.CMD

The TMS320C31 memory map, and interrupt vector locations are controlled by the linker script. Each interrupt or trap vector is placed into its own section, and those sections are placed by the linker script into the correct vector location. The linker command script OS_CPU.CMD can be used as the basis for project specific linker scripts.

8.1.5 BSP.H,.C

The port assembly language file OS_CPU_A.ASM defines the OS time tick interrupt service routine (ISR), and ISRs for the eight other DSP interrupt sources. The assembly language file calls C-coded handler functions, eg. INTO_Handler() that are defined by the board-support package (BSP) file BSP.H,.C. Functions for starting the OS timer, and blinking LEDs are also included in the BSP.

8.2 Port Testing

This section presents results from tests of the µCOS-II port.

8.2.1 Task-to-ISR context switching

Figure 8(a) shows the sequence of a task-to-ISR test. A test application was written containing a single task, and an interrupt handler for INTO. The interrupt handler posted a semaphore. The task would write to an I/O pin, then write to the interrupt flags register (IF) to trigger an interrupt on INTO, and then pend on the semaphore. When the pend completed, the I/O pin was toggled, and the process was repeated.

The high or low time of the square-wave on the I/O pin was measured to be 57.5µs. A square-wave that just toggled the I/O pin in an infinite loop has a high or low time of 4.0µs, so the task-to-ISR-to-task context switch time was 53.5µs. Interruption by the ISR causes a context save, and the ISR posting a semaphore and returning causes a context restore, hence a context switch time for the DSP port takes about 53.5µs. This corresponds to more than 18,000 context switches per second.

The COBRA correlator system processes a 1024Hz interrupt. The period of a 1024Hz interrupt is 977µs, so the 53.5µs context switch time will result in a CPU loading of 53.5µs/977µs × 100 = 5.5%, due simply to context switching.

8.2.2 Task-to-task context switching

Figure 8(b) shows the sequence of a task-to-task test. A test application was written containing two tasks, task A and task B. Task A posts semaphore A and then pend on semaphore B. Task B does the opposite, it pend on semaphore A, and posts semaphore B.

When Task A pend on semaphore B, it gives up the processor, and causes a task-level context switch to task B (task B is now ready, since task A posted the semaphore it was waiting for). The
 ISR posts semaphore

Task A sets I/O pin high
sets IF bit
pend on semaphore A

(a)

ISR posts semaphore

Task B sets I/O pin high
sets IF bit
pend on semaphore

(b)

Figure 8: μCOS-II DSP port testing. (a) task-to-ISR context switching, and (b) task-to-task context switching.

Time between the rising-edge of the I/O pin toggled by task A, to the rising-edge of the I/O pin toggled by task B, is the time taken for a task-to-task context switch. The time between edges was measured to be 74.5μs, so adjusting for the 4μs I/O pin pulsing overhead, the task-to-task context switch time was 70.5μs.

8.2.3 Interrupt nesting

Figure 9 shows the sequence of an interrupt nesting test. A test application was written containing a single task that set four I/O pins high, wrote to the interrupt flags register to trigger interrupts INT0 to INT1, and then went to sleep using OSTimeDly(). Interrupt handlers were defined that cleared individual I/O pins and then returned. The time between the I/O pins being set and INT0 clearing the I/O pin 0 consists of the time taken for the processor to recognize that four interrupts have been asserted, select INT0 as the highest priority, and then process the INT0 ISR. Hence the high-time of the I/O pin 0 about that of a context save. The measured high-time of the pulse was 21.7μs. The subsequent I/O pin pulses are longer by the context restore of the interrupt that has just completed, and then the context save of the ISR being entered. The time between falling edges of the ISRs was measured to be about 41μs.
The main result of the test was that interrupt nesting operates as intended.

8.2.4 Small-memory model (data-page)

The correct context saving and restoring of the data-page pointer by the port was tested by writing an application with two tasks, and setting up a repetitive timer using the DSP timer 0 (timer 1 is used for the OS tick). The highest priority task pended on a semaphore that was posted by the timer ISR. When the pend completed, the task would pulse an I/O pin, and loop back to the pend. The second task pulsed an I/O pin around the calculation of a 256-point FFT using the Forward FFT for real-valued samples, \texttt{ffft\_rl()}, supplied by Texas Instruments.

The application tests the data-page pointer due to the following:

- During the FFT, the data-page pointer is modified in the second task.
- When the timer ISR occurs during the FFT, the timer ISR saves the task context and then resets the data page.
- Since the first task has higher-priority than the FFT task, once the timer ISR handler posts the semaphore, the first task will run.

Viewing the two task I/O pins on an oscilloscope showed the FFT I/O pin pulse normally lasting 300\(\mu\)s, however, when the timer task pulse occurs, the FFT task I/O pin pulse increased to 465\(\mu\)s due to the context save to process the ISR, the context restore to the timer task, the context save of the timer task, and the context restore back to the FFT task.

The main result of the test was that the data-page pointer context was being saved and restored correctly.
Table 4: Memory transfer benchmarks (512-words).

<table>
<thead>
<tr>
<th>Test</th>
<th>Code Location</th>
<th>Source Buffer</th>
<th>Destination Buffer</th>
<th>Cache</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPTS copy</td>
<td>RAM0</td>
<td>RAM1</td>
<td>RAM1</td>
<td>on/off</td>
<td>36.5</td>
</tr>
<tr>
<td>RPTB copy</td>
<td>RAM0</td>
<td>RAM1</td>
<td>RAM1</td>
<td>on/off</td>
<td>36.5</td>
</tr>
<tr>
<td>RPTS copy</td>
<td>SRAM</td>
<td>RAM1</td>
<td>RAM1</td>
<td>on</td>
<td>41.6</td>
</tr>
<tr>
<td>RPTS copy</td>
<td>SRAM</td>
<td>RAM1</td>
<td>RAM1</td>
<td>off</td>
<td>41.9</td>
</tr>
<tr>
<td>RPTB copy</td>
<td>SRAM</td>
<td>RAM1</td>
<td>RAM1</td>
<td>on</td>
<td>41.6</td>
</tr>
<tr>
<td>RPTB copy</td>
<td>SRAM</td>
<td>RAM1</td>
<td>RAM1</td>
<td>off</td>
<td>72.8</td>
</tr>
</tbody>
</table>

9 Memory Transfer Benchmarking

Figure 1 shows the architecture of the TMS320C31 DSP. The DSP has two internal blocks of RAM of 1K each, multiple internal program and data buses, and a single external bus. Memory Access for Maximum Performance is the subject of Section 8.4 in the C31 User’s Guide (p8-22 [13]). Table 8-1 in the guide shows the options for one program fetch and one data access in a single cycle, while Table 8-2 shows the options for one program fetch and two data accesses. This section contains test results of various combinations of program code and data location, i.e., on-chip or off-chip, the effect of the cache, and the improvement possible with hand-optimized assembler routines.

The DSP’s multiple internal buses make it possible to fetch an instruction in parallel with two internal data accesses, and the instruction set contains parallel instructions to exploit this. The COBRA system is I/O bound, so a critical performance measurement is the memory transfer bandwidth, where memory can be DSP on-chip RAM, external SRAM, FPGA RAM, or SDRAM. The DSP instruction set contains the parallel load and store instruction \( \text{LDI} \| \text{STI} \). If the register used in the load is the same as that used in the store, then the store will occur first, so for that specific use the instruction is a store-and-load. A block-copy function can be written using the parallel store-and-load instruction, by loading the first word into a register, then performing a repeated loop of parallel store-and-load operations, followed by the final store. The repeated loop can be implemented using the repeat-block instruction, \( \text{RPTB} \), or using the repeat-single instruction, \( \text{RPTS} \). The C31 on the COBRA boards operates with a 33MHz clock. The DSP runs with a clock at half that rate, so an instruction cycle is 60ns. A block-copy of 512-words using single-cycle parallel store-and-load instructions would occur in approximately 60ns \( \times 512 \approx 31\mu s \).

Table 4 shows the memory benchmark results from the COBRA boards. The test application toggled an LED around repeated calls to the copy function under test. C-callable assembly functions were written that used the repeat-block, \( \text{RPTB} \), and repeat single, \( \text{RPTS} \), instructions around the parallel load and store instruction \( \text{LDI} \| \text{STI} \). The optimal bandwidth was achieved with the main application stored in on-chip RAM0, and the source and destination buffers in RAM1. Moving the program into off-chip RAM increased the measurement time slightly, however, this increase should be due to the I/O pin pulsing code, not the copy, since the \( \text{RPTS} \) instruction and its target instruction get loaded by the DSP, and then repeat runs with no further instruction fetches. This is not the case for the \( \text{RPTB} \) instruction, and you can see that if the cache is disabled, the transfer time almost doubles. The I/O pin pulsing overhead was determined to be 2.8\( \mu s \) on-chip, and 5.3\( \mu s \) off-chip, i.e., an increase of 2.5\( \mu s \) between the two link locations. Since this is less than the 5\( \mu s \) increase observed in the copy tests, the on-chip to off-chip overhead is not totally due to the I/O pulsing code.
The C-callable assembly functions were preceded with tests based on a C-coded while-loop, basically:

```c
void dsp_memcpy(int *dst, int *src, int len)
{
    while (len--)
        *dst++ = *src++;
}
```

and the similarly coded for-loop

```c
void dsp_memcpy(int *dst, int *src, int len)
{
    int i;
    for (i = 0; i < len; i++)
        *dst++ = *src++;
}
```

The code was compiled with various optimization and inlining options, and in the best performing code, the assembly code generated for either function used a load instruction followed by a store instruction inside a repeat-block. The transfer time for 512-words between buffers in RAM1, with code in RAM0, was 67.4\(\mu\)s. The same test with code linked in SRAM gave 72.4\(\mu\)s with the cache on and 135\(\mu\)s with the cache off (so use the cache!). With compiler optimization and inlining disabled, the transfer time was 880\(\mu\)s with the cache on, and over 1.4ms with the cache off!

The transfer test was repeated using the `memcpy()` routine supplied by TI as part of the DSP standard C library. The transfer time with code linked to on-chip RAM was 37.8\(\mu\)s. The compiler optimization and inlining settings only affect the compilation of the test code, since `memcpy()` is linked from a precompiled library; in this case `rts30.lib` (which, according to p5-2 [9], was built with level 2 optimization, inlining, and repeat-single disabled). Since the transfer time was twice as fast as the C functions above, the TI `memcpy()` must be triggering the use of the parallel store-and-load operation. The `memcpy.c` file was extracted from the `rts.src` library to see how this was achieved. The key difference; the source parameter was defined as `const`! Declaring the source as `const` in the C functions above, and using level 2 optimization with inlining, gave transfers of 36.7\(\mu\)s. However, if optimization and inlining are turned off, the performance drops to 690\(\mu\)s (cache on).

The conclusion of this investigation was that the C compiler optimization is highly dependent on the form of the C code, and that the assembly output from an optimized section of performance-critical code should be inspected for additional optimizations missed by the compiler. In fact, using the compiler-optimized code as the basis for hand-optimized C-callable assembler code can be used to create code that is independent of the compiler optimizer settings. Another conclusion of this analysis was that any memory transfers of data in the COBRA code should call the C-callable assembly, and not simply use a while or for-loop.

Transfers of data between the DSP on-chip RAM blocks obviously yields the highest performance measurements, but data still needs to be transferred into and out of the on-chip memory. Two options exist: DSP initiated transfers, and DMA transfers. Both methods are likely needed in code, and there are restrictions as to what the DSP can do while the DMA controller is performing a transaction between external and internal memory.

The DSP DMA controller has its own internal bus, so it can perform reads and writes in parallel with the DSP (in cases where there are no resource conflicts). The DMA controller uses a single data bus, so its highest-performance is a single clock-cycle read followed by a single clock-cycle write (p12-68 [13]), i.e., a block transfer of 512-words between DSP internal RAM should take 120\(\mu\)s \(\times\) 512 = 61.4\(\mu\)s. DMA transfer completion is generally handled by an interrupt, however, for benchmarking tests, it is possible to poll the interrupt flags register (IF) without causing a DSP-DMA access conflict (p12-75 [13]), i.e., the polling should not affect the measurement.
Table 5: DSP and DMA transfer benchmarks (512-words, RAM0-code, RPTS copy, cache on).

<table>
<thead>
<tr>
<th>Source Buffer</th>
<th>Destination Buffer</th>
<th>Time (µs)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DSP</td>
<td>DMA</td>
<td></td>
</tr>
<tr>
<td>RAM1</td>
<td>RAM1</td>
<td>36.5</td>
<td>67.5</td>
<td></td>
</tr>
<tr>
<td>RAM1</td>
<td>SRAM</td>
<td>98.5</td>
<td>98.5</td>
<td></td>
</tr>
<tr>
<td>RAM1</td>
<td>FPGA RAM</td>
<td>160</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>RAM1</td>
<td>SDRAM</td>
<td>193</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>RAM1</td>
<td>67.7</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>SRAM</td>
<td>191</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>FPGA RAM</td>
<td>253</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>SDRAM</td>
<td>285</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>FPGA RAM</td>
<td>RAM1</td>
<td>160</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>FPGA RAM</td>
<td>SRAM</td>
<td>285</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>FPGA RAM</td>
<td>FPGA RAM</td>
<td>345</td>
<td>377</td>
<td></td>
</tr>
<tr>
<td>FPGA RAM</td>
<td>SDRAM</td>
<td>378</td>
<td>409</td>
<td></td>
</tr>
<tr>
<td>SDRAM</td>
<td>RAM1</td>
<td>254</td>
<td>316</td>
<td></td>
</tr>
<tr>
<td>SDRAM</td>
<td>SRAM</td>
<td>379</td>
<td>409</td>
<td></td>
</tr>
<tr>
<td>SDRAM</td>
<td>FPGA RAM</td>
<td>440</td>
<td>475</td>
<td></td>
</tr>
<tr>
<td>SDRAM</td>
<td>SDRAM</td>
<td>475</td>
<td>503</td>
<td></td>
</tr>
</tbody>
</table>

Table 5 shows the transfer results for memory transfers while program code is linked to RAM0.

**DSP results:**

The key feature of these tests is that there should be no program fetch conflicts. The transfer from source RAM1 to destination SRAM involves a DSP external bus write, while the transfers from source SRAM to destination RAM1 involves an external bus read. The DSP can perform an external bus write in as few as two DSP clock cycles (120ns), and an external bus read in as few as one clock cycle (60ns). The COBRA SRAM interface adds an additional wait-state to both writes and reads to avoid a timing violation at the system controller FPGA, so a DSP external bus write takes 180ns, and a read 120ns. A 512-word write should take about 92.2µs, and a read should take about 61.4µs. The measurements in Table 5 are both 6.3µs larger than this, which is due to the overhead of the loop logic and the external I/O pin toggle code. A read from SRAM (120ns) followed by a write to SRAM (180ns) should take about 300ns per word, so an SRAM to SRAM transfer of 512-words should take 153.6µs. The measured value of 191µs indicates that there is at least a single clock (60ns) period turn-around, i.e., $512 \times 360ns + 6.3\mu s \approx 191\mu s$. The other memory transfer results can be similarly analyzed.

**DMA results:**

The C31 User’s Guide indicates than an external memory write by the DMA controller is the same as a DSP external memory write, whereas a read by the DMA controller will be one clock cycle longer than a DSP initiated read, due to the DMA controller write to its internal temporary register (p12-69 [13]). The difference between the DSP external memory read measurements versus the DMA external memory read measurements should be about $512 \times 60ns \approx 31\mu s$. The DMA transfer benchmark for RAM1 to RAM1 of 67.5µs is close to the expected value of 61.4µs, indicating that
Table 6: DSP and DMA transfer benchmarks (512-words, SRAM-code, RPTS copy, cache on).

<table>
<thead>
<tr>
<th>Source Buffer</th>
<th>Destination Buffer</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DSP</td>
</tr>
<tr>
<td>RAM1</td>
<td>RAM1</td>
<td>41.6</td>
</tr>
<tr>
<td>SRAM</td>
<td>104</td>
<td>105</td>
</tr>
<tr>
<td>FPGA RAM</td>
<td>166</td>
<td>168</td>
</tr>
<tr>
<td>SDRAM</td>
<td>199</td>
<td>202</td>
</tr>
<tr>
<td>SRAM</td>
<td>RAM1</td>
<td>72.8</td>
</tr>
<tr>
<td></td>
<td>199</td>
<td>229</td>
</tr>
<tr>
<td>FPGA RAM</td>
<td>259</td>
<td>292</td>
</tr>
<tr>
<td>SDRAM</td>
<td>290</td>
<td>322</td>
</tr>
<tr>
<td>FPGA RAM</td>
<td>RAM1</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td>290</td>
<td>322</td>
</tr>
<tr>
<td>SDRAM</td>
<td>352</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>384</td>
<td>416</td>
</tr>
<tr>
<td>SDRAM</td>
<td>RAM1</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td>384</td>
<td>416</td>
</tr>
<tr>
<td>FPGA RAM</td>
<td>445</td>
<td>483</td>
</tr>
<tr>
<td>SDRAM</td>
<td>480</td>
<td>512</td>
</tr>
</tbody>
</table>

the DMA controller is performing two-cycle transfers. The transfers from RAM1 to SRAM, FPGA RAM, and SDRAM are the same for DSP and DMA as expected. The DMA transfer from SRAM to RAM1 takes almost twice as long as the DSP initiated transfer due to the fact that the DMA controller sequence is external memory read (120ns), temporary register write (60ns), RAM1 write (60ns) for a total expected transfer time of 512×240ns ≈ 123µs, whereas the DSP can use its multiple buses to perform a read in parallel with a write, so the external memory read time is the limiting factor (512 × 120ns ≈ 61.4µs). The DMA transfers from FPGA RAM to RAM1 and SDRAM to RAM1 both increase by 61µs for this same reason. The DMA transfers between external memory devices all increase by 31µs due to the DMA temporary register store.

Table 6 shows the transfer results for memory transfers while program code is linked to SRAM. The measured times are all about 6µs greater than the measurements in Table 5. As stated during the discussion of Table 4, this increase is due in part to the I/O pin pulsing code and the test code loop, and is not due to the copy, since for the DSP transfers, the RPTS instruction and its target instruction get loaded by the DSP, and then repeat runs with no further instruction fetches. During DMA transfers, there is the potential for a bus conflict given that the program code is now located in external SRAM, however, the only code that can cause the conflict is the IF polling loop, and that should be in the cache (which the measurements confirm it must be). The effect of the cache can be determined by disabling it. For example, RAM1 to SRAM transfers with the cache on take 105µs, whereas with the cache off the transfer takes 385µs. Similarly SRAM to RAM1 transfers with the cache on take 136µs, whereas with the cache off the transfer takes 355µs. The external memory bus conflict due to program fetch and DMA causes the transfer to take about three times longer.

Use of the DMA controller in software needs to be carefully coordinated with the DSP performing some parallel action such that resource (bus) conflicts are avoided.
10 Data Processing Benchmarking

Data processing in the correlator consists of

- Integer-to-floating point conversion.
- Normalization; offset (subtraction) and scaling (multiplication).
- Fourier transform
- Delay correction vector generation (calculation of samples from a complex exponential).
- Delay correction application (complex multiplication of two vectors).
- Accumulation of data.

10.1 Integer-to-floating point conversion

Data read from the correlator FPGAs is lag data in integer format. The lag data will be transferred to DSP on-chip RAM prior to FFT either by the DSP copying the data from the FPGAs, or the DMA controller copying the data. If the DSP performs the copy of the lags from the FPGA, then it can perform integer-to-floating point as part of the copy.

Integer-to-floating point conversion compiles to use the FLOAT assembly instruction. This instruction can convert an integer stored in a register, or access the integer from memory. The result is then stored to a register. Floating point conversion of a vector of data requires two clock cycles; FLOAT to load the integer and convert it to float, and STF to store the float. There is a parallel instruction FLOAT||STF, so for conversions of vectors of data located in the the DSP on-chip RAM, it is possible to perform the load and conversion of the first integer, then the store of that float in parallel with the load and conversion of the second integer. The parallel instruction can be used with repeat-block (or repeat-single) to implement store-and-conversion in a single clock cycle.

Floating-point conversion of a 512-word vector should take \( 512 \times 120\text{ns} = 61.4\mu\text{s} \) for non-parallel code, and \( 512 \times 60\text{ns} = 30.7\mu\text{s} \) for parallel code. Benchmark tests in RAM1 measured 67.2\mu\text{s} and 35.6\mu\text{s} (code was executing from RAM0).

10.2 Data normalization

The lag data is normalized by subtracting an offset, and multiplying by a scaling factor. There are several options for the data normalization;

1. \( \text{normalized}[n] = (\text{float})(\text{int}\_\text{data}[n] - \text{int}\_\text{offset})\times\text{scaling} \)
   Integer subtraction, floating-point conversion, and floating-point multiplication.

2. \( \text{normalized}[n] = ((\text{float})\text{int}\_\text{data}[n] - \text{flt}\_\text{offset})\times\text{scaling} \)
   Floating-point conversion, floating-point subtraction, and floating-point multiplication.

3. \( \text{normalized}[n] = (\text{flt}\_\text{data}[n] - \text{flt}\_\text{offset})\times\text{scaling} \)
   Floating-point subtraction, and floating-point multiplication, on a vector of data previously converted to floating-point. For vectors of data, this option effectively splits the conversion and normalization into two separate loops.

The three options can be unrolled into different parallel instruction options. The instruction sequence for option (1) is:

- SUBI integer subtraction (including load)
- FLOAT floating-point conversion
- MPYF floating-point multiplication
- STF store floating-point
If you take a copy of this sequence, and line it up with the sequence skewed by one, two, and three instructions, and compare the instruction-pairs to the parallel instructions available, you will find a parallel multiply-and-store float ($\text{MPYF}||\text{STF}$). If the register target of both instructions is the same, then the store occurs before the multiply, so you cannot use it to combine the last two instructions. However, you can use it in parallel to multiply the second sample while storing the first, i.e., there is the potential to save a single clock cycle.

The instruction sequence for option (2) is;

- **FLOAT** floating-point conversion (including load)
- **SUBF** floating-point subtraction
- **MPYF** floating-point multiplication
- **STF** store floating-point

Parallel instructions possible with this sequence are; floating-point convert and store $\text{FLOAT}||\text{STF}$, floating-point multiply and subtraction $\text{MPYF}||\text{SUBF}$, floating-point multiply and store $\text{MPYF}||\text{STF}$, and floating-point subtract and store $\text{SUBF}||\text{STF}$. The data normalization can be parallelized into the following sequence:

- **FLOAT** load and floating-point convert index 0 sample
- **SUBF** offset correct first sample 0
- **FLOAT** load and floating-point convert index 1 (next) sample
- **RPTB** repeat block
- **MPYF||SUBF** scale sample, offset next sample
- **FLOAT||STF** store sample, load next sample

The parallel multiple-subtract instruction has limitations as to what arguments it supports, and the targets must be registers, i.e., the destination of the multiply must be $F0$ or $F1$ (extended-precision registers $R0$ and $R1$), and the destination of the subtract must be $F2$ or $F3$. Given that the result of the subtraction will be the argument to the multiplication in the next loop iteration, the form of the desired parallel code is

\[
\text{MPYF} F2, \text{scaling}, F0 \\
|| \text{SUBF} F0, \text{offset}, F2
\]

where data is loaded into $F0$, the subtraction result is placed in $F2$, where it is the argument to the multiplication, the result of which is stored into $F0$. The scaling and offset arguments to the parallel multiplication-subtraction instruction would ideally be in registers. This option is available in the code generation tools greater than version 5.0 (the tools used were version 5.11) by using the assembler flags `-v31 -msrev6`.

The following parallel instructions used inside a repeat-block halve the processing time required to floating-point convert, and normalize a lag vector;

\[
\text{MPYF3} F2, F1, F0 \\
|| \text{SUBF3} F0, F3, F2 \\
\text{FLOAT} \*\text{AR0++(1)},F0 \\
|| \text{STF} F0,\*\text{AR1++(1)}
\]

where $F1$ stores the scaling parameter, $F3$ the offset, $\text{AR0}$ the data load address, and $\text{AR1}$ the data store address. The parallel loop needs to be initialized with the first data converted to float, offset corrected, and stored to $F2$ in preparation for the multiply. And the second data needs to be converted to float and stored in $F0$, ready for its offset correction. Since each iteration of the loop loads the next data value, the repeat counter is loaded to perform length-1 iterations. This ensures that only samples within the span of the vector are read (this is probably not really required, but it is the correct thing to do).
The instruction sequence for option (3) is:

- **SUBF**: floating-point subtraction
- **MPYF**: floating-point multiplication
- **STF**: store floating-point

where it is assumed that the data has already been converted to floating-point format. Various parallel versions of each of these instructions exist, and it would be possible to create a parallel sequence that took an average of 1.5 clock cycles. Adding this to the 1 clock cycle required to floating-point convert gives a total lag normalization time of 2.5 clocks per sample. Option (2) achieves 2 clocks per sample, so is the optimal choice.

Floating-point conversion, offset correction, scaling, and store of a 512-word vector should take \(512 \times 240\text{ns} = 123\mu\text{s}\) for non-parallel code, and \(512 \times 120\text{ns} = 61.4\mu\text{s}\) for parallel code. Benchmark tests in RAM1 measured 129\(\mu\text{s}\) and 67.5\(\mu\text{s}\) (code was executing from RAM0).

### 10.3 Data accumulation

The correlator spectra are accumulated for between 100ms and 500ms before the data is converted to IEEE and sent to the host CPU. The accumulation is simply a floating-point summation (even for the complex valued data). An obvious C coded implementation is

```c
void accumulate(float *dst, const float *src, int len)
{
    int i;
    for (i = 0; i < len; i++) {
        *dst++ += *src++;
    }
}
```

The assembly code for the accumulation operation compiles to

```
ADDF3  *AR0,*AR1++(1),F0
STF    F0,*AR0++(1)
```

where AR0 is used for dst and AR1 is used for src. A parallel add-store instruction exists, i.e., ADDF3||STF, however, it does not support two indirectly addressed arguments to the add, so it can not be used to improve the performance of the accumulation. This restriction is likely due to the fact that the DSP has only two internal data buses, so the DSP can only load one sample from memory, while storing another.

A floating-point add (ADDF3) requires one clock cycle (subject to operand ordering effects), and a floating-point store (STF) requires one clock cycle, i.e. the code should take two cycles per sample, and so accumulation of a 512-sample vector would require \(512 \times 120\text{ns} = 61.4\mu\text{s}\). Benchmarking of a C coded function and an assembly version both gave 98.5\(\mu\text{s}\), i.e., 3 clock cycles per word (for code in RAM0 and buffers in RAM1). According to p8-26 of the User's Guide [13]), two internal memory reads can occur in a single cycle, so the ADDF3 should be a single-cycle operation, and similarly, the store can occur on the next clock cycle. Its possible that the additional clock cycle overhead is due to the auxiliary register arithmetic unit (ARAU) used for source and destination addressing.

An assembly coded accumulate function was written that contained a repeat-block with the following two adds and parallel store instruction:

```
ADDF3  *AR0,*AR1++(1),F0
ADDF3  **AR0(1),*AR1++(1),F1
STF    F0,*AR0++(1)
|| STF    F1,*AR0++(1)
```

Benchmarking of this code for 512-word transfers gave 67.5\(\mu\text{s}\), i.e., 2 clock cycles per word.
Table 7: Real-valued samples FFT benchmarking.

<table>
<thead>
<tr>
<th>Code</th>
<th>Twiddles</th>
<th>Data Buffer</th>
<th>Time (µs)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>32-samples</td>
<td>64-samples</td>
<td>128-samples</td>
<td></td>
</tr>
<tr>
<td>RAM0</td>
<td>RAM0</td>
<td>RAM0</td>
<td>51</td>
<td>93</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM0</td>
<td>RAM1</td>
<td>42</td>
<td>71</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM0</td>
<td>SRAM</td>
<td>97</td>
<td>200</td>
<td>450</td>
<td></td>
</tr>
<tr>
<td>RAM0</td>
<td>RAM1</td>
<td>RAM0</td>
<td>49</td>
<td>87</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM1</td>
<td>RAM1</td>
<td>42</td>
<td>71</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM1</td>
<td>SRAM</td>
<td>97</td>
<td>200</td>
<td>450</td>
<td></td>
</tr>
<tr>
<td>RAM0</td>
<td>SRAM</td>
<td>RAM0</td>
<td>52</td>
<td>95</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM0</td>
<td>RAM1</td>
<td>44</td>
<td>79</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM1</td>
<td>SRAM</td>
<td>100</td>
<td>209</td>
<td>475</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>RAM0</td>
<td>RAM0</td>
<td>72</td>
<td>109</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM0</td>
<td>RAM1</td>
<td>72</td>
<td>109</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM0</td>
<td>SRAM</td>
<td>142</td>
<td>257</td>
<td>528</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>RAM1</td>
<td>RAM0</td>
<td>72</td>
<td>109</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM1</td>
<td>RAM1</td>
<td>72</td>
<td>109</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAM1</td>
<td>SRAM</td>
<td>142</td>
<td>257</td>
<td>528</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>SRAM</td>
<td>RAM0</td>
<td>77</td>
<td>119</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SRAM</td>
<td>RAM1</td>
<td>77</td>
<td>119</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SRAM</td>
<td>SRAM</td>
<td>145</td>
<td>267</td>
<td>555</td>
<td></td>
</tr>
</tbody>
</table>

10.4 FFT

Table 7 shows benchmarking results for the FFT routine `ffft_r1()`; forward FFT, real-valued samples, supplied by TI (the code was version 2.0, written by Alex Tessarolo, July 1991, see p6-42 [7]). The test code was run with the cache on. The FFT code is in assembler, so inlining and optimization were disabled (they do not affect assembler code). The efficiency of a complex FFT is of operation order $O(N \log_2 N)$, and for a real-valued FFT, $N$ can be replaced with $N/2$. Real-valued FFT sizes 32, 64, and 128 correspond to operation orders of 64, 160, and 384. The best FFT times in Table 7, i.e., 42µs, 71µs, 141µs, correspond to the real-valued FFT operation orders multiplied by a scale factor of 0.3µs/operation and an overhead of 29µs (using a least-squares fit).

Table 7 shows that off-chip processing of the FFT results in a significant loss of performance; almost four times worse than the on-chip FFT. The optimal FFT performance is obtained with the FFT code (text and data sections from the assembly source) in RAM0, the twiddle factors in either RAM0 or RAM1, and the FFT buffer in RAM1.

The DSP DMA controller has an independent address and data bus internal to the DSP, so it should be possible to DMA into either RAM block while the DSP is executing an FFT. If however an FFT in parallel with a DMA operation causes a conflict, the FFT data buffer can be moved into RAM0 with a corresponding loss in FFT performance. The trade-off being that a DMA in parallel with the DSP processing an FFT could eliminate the DSP moving data on or off-chip.
10.5 Complex-valued exponential calculation

The correlator system applies a delay correction to the cross-correlation data. The correction has the form:

\[ D(f; f_0, \tau) = \exp(j2\pi(f + f_0)\tau) = \exp(j2\pi f\tau + j2\pi f_0\tau) \]  

(1)

where \( f \) is a frequency variable, \( f_0 \) is a fixed frequency, and \( \tau \) is a delay (fixed during the application of a correction, but varying with time). The second form of the equation consists of a linear phase slope plus a phase offset.

The correlator delay correction algorithm calculates the complex-valued correction vector, and then applies the correction to the cross-spectra using a complex-valued vector multiplication. The delay correction is split into these two steps, so that an assembler optimized complex-vector multiplication routine can be used if necessary.

The calculation of the delay correction exponential consists of the calculation of a cosine and sine component for each sample in cross-correlation spectra. According to the source code for \( \cos() \) and \( \sin() \) extracted from \texttt{rts.src} the DSP code is based on algorithms from Chapter 8 of Software Manual for the Elementary Functions, Cody and Waite, Prentice Hall 1980. The calculation of a single \( \cos() \) and \( \sin() \) requires multiple internal calculations.

A correlator board processes up to ten cross-correlations, with up to 128-lags (which each FFT to 65-channel cross-spectra), and new lag data is ready for FFT, delay correction, and accumulation every 16ms. Each cross-spectra has a different delay correction. The calculation of the delay correction sinusoidal functions is too expensive to meet these processing requirements. The processing load can be reduced by first calculating a table of sinusoidal values across a full period of a sinusoid (or a quadrant of a sinusoid if memory is limited). The calculation of \( \cos() \) and \( \sin() \) can then be reduced to index generation and table lookup. The size of the table is determined by the acceptable phase and delay correction error. For example indexing into a table of length 1024 entries, has a phase resolution of 0.35° (360°/1024). A correction error of 0.35° for a 500MHz spectral channel is equivalent to a delay error of 1.95ps (2000ps/1024). If the table length is increased to 2048 entries, then the delay error drops to 0.98ps.

Sine table generation

A sine table of length \texttt{TWO_PI_LENGTH} was built using code containing the following loop;

```c
    dphi = 2*PI/TWO_PI_LENGTH;
    p = sin_table;
    for (i = 0; i < TWO_PI_LENGTH; i++) {
        *p++ = sin(i*dphi);
    }
```

The table generation times for 1024, 2048, and 4096 length tables were 9.3ms, 18.6ms, and 37.2ms (the time scales linearly with table size). The code and sine table were located in off-chip SRAM, as this represents the use-case in the correlator system (there is only 2K of on-chip SRAM). The DSP C compiler options were set to use full-inlining and optimization level 2 (the same options used to compile the \( \sin() \) routine in \texttt{rts30.lib}). The sine table can be generated during system initialization, so its execution time is not critical.

The correlator host CPU sends the DSP cross-correlation processing code the sum-of-frequencies component \( f_0 \), and the baseline delay correction \( \tau \). The baseline delay correction is sent as two components; whole nanoseconds, \( \tau_{ns} \), and residual femtoseconds \( \tau_{fs} \), where \( \tau_{fs} = \tau - \tau_{ns} \). The host scales these quantities to 32-bit integer values to pass to the DSPs, and the DSPs convert them back
to single-precision floating-point numbers. The delays are passed as two separate numbers due to the fact that the digitizer DSPs update the 1ns digital delay lines using the nanosecond value. Having the host perform all the calculations in a single place ensures consistency with the calculations. The use of two 32-bit numbers also increases the dynamic range of the delay representation. The delay across a 2km baseline at the speed-of-light is 6.6µs, and since some of this delay is through slower fiber, we will give the maximum delay as 10µs. The digitizer DSPs program 1ns delay lines, and so the nanosecond component of delay requires ceil(log₂(10µs/1ns)) = 14-bits (τns will range from -10000 to 10000). The cross-correlation corrections should be applied with sub-ps accuracy, so as not to contribute to the system phase error. The cross-correlation correction will range between -1ns and 1ns, and will be sent to the DSPs scaled to femtoseconds, i.e., the representation will have 1fs accuracy, and requires ceil(log₂(1ns/1fs)) = 20-bits (τfs will range from -1000000 to 1000000).

The DSP single-precision floating-point format has a fractional part of 23-bits, so both integer value delay components can be converted to floating-point without loss of precision. However, those floating-point numbers should not be added to produce a single delay component, since delay correction calculations on delay values of up to 10µs with 0.01ps precision requires ceil(log₂(10µs/0.01ps)) = 30-bits. Floating-point calculations in the DSP registers occur using 40-bit floating-point format numbers that carry an extra 8-LSBs in the fractional part, giving a 31-bit fractional part, so higher-precision calculations are possible.

The calculation of the phase terms involving fτ and f₀τ must also maintain precision. The values of fτ and f₀τ are arguments to an exponent, so the modulo-2π values are only of interest, i.e., only the fractional part of fτ or f₀τ is required. The calculation involving the modulo component should occur with no more than 0.01° error, i.e., at least ceil(log₂(360°/0.01°)) = 16-bits. The value of f₀ can be as high as 9GHz, which when multiplied by 10µs gives 90000, a 17-bit number, so only 6-bits of the 23-bit fractional part of a single-precision floating-point number are potentially preserved by the calculation, or 14-bits if the calculation is left in a DSP floating-point register. The dynamic range of the calculation can be increased by eliminating the modulo component of the delay, i.e.,

\[
\begin{align*}
\tau'_\text{ns} &= \tau\text{ns} - \text{floor}(f_0\tau\text{ns})/f_0 \\
\tau'_\text{fs} &= \tau\text{fs} - \text{floor}(f_0\tau\text{fs})/f_0
\end{align*}
\]  

(2)

which removes a whole number of f₀ periods from the delay components τns and τfs. The floor function rounds down to the nearest integer, even for negative values, so the residual phase fraction will be a positive value in the range 0.0 to 1.0. The residual phase fraction is ultimately multiplied by a lookup table length parameter and then that answer is rounded to the nearest integer to produce a sine or cosine lookup value.

The fτ component of the delay is corrected for in two parts; digital delay lines correct to the nearest nanosecond on an antenna basis, leaving a residual ±0.5ns, and cross-correlations are corrected for the residual delay, which can range from ±1.0ns (since the antenna residuals can add). The highest frequency in the cross-correlation spectra is 500MHz, so the largest residual fτfs components are ±0.5 (±180° phase corrections), which already lie within the modulo range, so the residual delay component, τfs, does not need adjustment.

The code for delay correction calculation is

```c
tau_ns0 = tau_ns - floor(tau_ns*f0)/f0;
tau_fs0 = tau_fs - floor(tau_fs*f0)/f0;
offset_ns = tau_ns0*f0; // 0.0 to 1.0
offset_fs = tau_fs0*f0; // 0.0 to 1.0
offset = offset_fs + offset_ns;
df = bandwidth/(nchannels-1);
dfrac = df*tau_fs;
p = delay_buffer;
for (i = 0; i < nchannels; i++) {
```

// Fractional phase
sin_frac = i*dfrac + offset;
cos_frac = sin_frac + 0.25;

// Modulo index generation
sin_frac -= floor(sin_frac);
cos_frac -= floor(cos_frac);
sin_index = (int)round(sin_frac*two_pi_length);
cos_index = (int)round(cos_frac*two_pi_length);

// Table lookup
*p++ = sin_table[cos_index];
*p++ = sin_table[sin_index];

The round operation is not defined in the TI tools math library, so the code was replaced with an integer cast:

sin_index = (int)(sin_frac*two_pi_length+0.5);
cos_index = (int)(cos_frac*two_pi_length+0.5);

The addition of 0.5 forces the number to the nearest integer, instead of truncating to the integer part of the floating-point number. If the addition is not used, then the phase error due to the lookup operation will increase by a factor of two.

The floor operation turns out to be very expensive; a loop to calculate the delay correction vector for a 65-channel spectrum takes 931µs (the delay correction buffer was located in RAM0, the code and sine table were located in external SRAM, the compiler was set to full inlining with optimization level 2, and the cache was on). Viewing the generated assembly code shows that the code contains a number of conditional tests, and calls to a floating-point modulus operator, modf(). Replacing the floor operation with a cast to an integer is not quite equivalent, since the resulting fractional part can be any value between -1.0 to 1.0 (exclusive). Negative values can have 1.0 added to them to force the value into the range 0.0 to 1.0. Code without floor in the loop looks like

for (i = 0; i < nchannels; ++i) {
  // Fractional phase
  sin_frac = i*dfrac + offset_fs + offset_ns;
cos_frac = sin_frac + 0.25;

  // Modulo index generation
  sin_frac -= (int)sin_frac;
  if (sin_frac < 0) {
    sin_frac += 1.0;
  }
cos_frac -= (int)cos_frac;
  if (cos_frac < 0) {
    cos_frac += 1.0;
  }
sin_index = (int)(sin_frac*two_pi_length+0.5);
cos_index = (int)(cos_frac*two_pi_length+0.5);

  // Table lookup
  *p++ = sin_table[cos_index];
  *p++ = sin_table[sin_index];
}
The loop for a 65-channel spectra takes 285\(\mu\)s. For 10 correlations delay corrected every 15.625ms, this delay correction vector generation code represents a processor load of \(\frac{285\mu\text{s}}{15.625\text{ms}} \times 10 \times 100 = 18.2\%\).

The number of operations per loop iteration can be minimized by removing the negative fraction checks, and by removing the addition of 0.5 in the index calculations. The resulting fraction terms in the loop will range between -1.0 and 1.0, so the sine lookup table represent the fractional angles -1.0 to 1.0 (i.e., angles \(-2\pi\) to \(2\pi\)). This doubles the size of a sine table required to reach a particular angular precision, but is acceptable since external memory is available. The value of the cosine is calculated as the sine fraction plus 0.25. The cosine operations can be minimized by making the sine table range from -1.0 to 1.25, and then generating the cosine index directly from the sine index. The minimum-operation delay correction vector generator code is

```c
sin_table_center = &sin_table[TWO_PI_LENGTH];
for (i = 0; i < nchannels; i++) {
    // Fractional phase
    sin_frac = i*dfrac + offset;
    // Index generation (-TWO_PI_LENGTH to 1.25*TWO_PI_LENGTH)
    sin_frac -= (int)sin_frac;
    sin_index = (int)(sin_frac*two_pi_length);
    cos_index = sin_index + half_pi_length;

    // Table lookup (relative to table center)
    *p++ = sin_table_center[cos_index];
    *p++ = sin_table_center[sin_index];
}
```

where the lookup table length was \(2.25 \times \text{TWO}_{-}\text{PI}_{-}\text{LENGTH}\), \(\text{HALF}_{-}\text{PI}_{-}\text{LENGTH} = 0.25 \times \text{TWO}_{-}\text{PI}_{-}\text{LENGTH}\), and \(\text{TWO}_{-}\text{PI}_{-}\text{LENGTH}\) is divisible by 4 so that \(\text{HALF}_{-}\text{PI}_{-}\text{LENGTH}\) is an integer. The avoid an integer-to-float conversion inside the loop, the variable \(\text{two}_{-}\text{pi}_{-}\text{length}\) is a float assigned to \(\text{TWO}_{-}\text{PI}_{-}\text{LENGTH}\), and the variable \(\text{half}_{-}\text{pi}_{-}\text{length}\) is an integer assigned to \(\text{HALF}_{-}\text{PI}_{-}\text{LENGTH}\). The execution time of this loop for a 65-channel spectra is 129\(\mu\)s. For 10 correlations delay corrected every 15.625ms, this represents a processor load of 8.3\% (quite a savings over the original version of the loop). The assembly code generated for the minimum-operation loop consists of simple instructions, and its unlikely that hand-optimization would yield much improvement. Table 7 shows that the optimal benchmark for a 128-sample FFT is 141\(\mu\)s. So the delay correction vector generation is almost as expensive as the FFT, and the correction still has to be applied to the data!
10.6 Real and complex-valued vector multiplication

Delay correction in the correlator involves the multiplication of a complex-valued correction vector with the complex-valued cross-spectra vector. A complex-valued multiplication of a single sample can be written

\[(a + jb)(c + jd) = (ac - bd) + j(ad + bc)\]  

(3)

i.e., the real and imaginary components of the output complex sample each require two multiplications and one add (or subtract), for a total of four multiplies and two adds per complex sample. The following code triggers the use of four registers to store the complex sample components, and three addressing registers to handle the source and destination addressing:

```c
float a, b, c, d;
float *p = src_buffer;
float *q = dst_buffer;
float *r = dst_buffer;
for (i = 0; i < BUFFER_SIZE/2; i++) {
    a = *q++;
    b = *q++;
    c = *p++;
    d = *p++;
    *r++ = a*c - b*d;
    *r++ = a*d + b*c;
}
```

The following real-valued multiplication is used for benchmark comparison,

```c
float *p = src_buffer;
float *q = dst_buffer;
for (i = 0; i < BUFFER_SIZE; i++) {
    *q++ *= *p++;
}
```

The real-valued multiplication compiles to a repeat-block around the two instructions **mpyf3** and **stf**. With code in external SRAM, the cache on, full inlining, and optimization level 2, the multiplication of 512-samples from a source buffer in RAM0 with a destination buffer in RAM1, takes 98.4µs, or 3 clocks per sample (92.2µs) plus 6.2µs overhead. This is the same initial result achieved with the data accumulation benchmarking, so the performance here could be reduced to 2 clock cycles by performing two multiplications and a parallel store.

The multiplication of a 256-sample complex valued vector (BUFFER_SIZE = 512) takes 197.6µs, or about 12 clocks per complex sample (184.3µs). The assembler code generated is

- **RPTB** block-1
- **LDFU** *AR0++(1),F4
- **LDFU** *AR0++(1),F3
- **LDFU** *AR1++(1),F1
- **LDFU** *AR1++(1),F2
- **MPYF3** F1,F4,F5
- **MPYF3** F2,F3,F0
- **SUBRF** F5,F0
- **STF** F0,*AR2++(1)
- **MPYF3** F1,F3,F1
- **MPYF3** F2,F4,F0

45
ADDF3 F0,F1,F0
STF F0,*AR2++(1)

block:

which consists of 12 single-cycle instructions. A slightly different form of the loop

float a;
float *p = src_buffer;
float *q = dst_buffer;
for (i = 0; i < BUFFER_SIZE/2; i++) {
    a = q[0];
    q[0] = a*p[0]-q[1]*p[1];
    q[1] = a*p[1]+q[1]*p[0];
    q += 2;
    p += 2;
}

compiles to

RPTB block-1
LDFU **AR0(2),F3
LDFU **AR1(1),F1
LDFU *AR1++(2),F2
MPYF3 **AR0,F1,F0
MPYF3 F3,F2,F4
SUBRF F4,F0
STF F0,*AR0
MPYF3 F3,F1,F0
MPYF3 **AR0,F2,F1
ADDF3 F0,F1,F0
STF F0,**AR0(1)

block:

which is a repeat-block containing 11 single-cycle instructions. The benchmark for 256-samples for this loop is 180.5µs.

The repeat block produced by either implementation contains a number of operations that can be performed in parallel, so hand-optimization of this loop could potentially halve the number of operations required. The relevant parallel instructions available are: add-store (ADDF3||STF), load (LDF||LDF), load-store (LDF||STF), multiply-add (MPYF3||ADDF3), multiply-store (MPYF3||STF), multiply-subtract (MPYF3||SUBF3), subtract-store (SUBF3||STF), and store (STF||STF). In addition to the use of the parallel operations, the load operations in the compiler generated assembly can be eliminated using indirect addressing modes in the multiply arguments. Hand-optimization of the code reduced the repeat block to

RPTB block-1
MPYF3 **AR0(0),***AR1(1),F2 ; src++, f2 = dst[0]*src[0] = a*c
MPYF3 **AR0(1),***AR1(1),F1 ; f1 = dst[1]*src[1] = b*d
MPYF3 *AR0++(1),**AR1(1),F0 ; f0 = dst[0]*src[1] = a*d, dst++
|| SUBRF F2, F1, F2 ; f2 = a*c - b*d
MPYF3 *AR0++(1),*AR1++(1),F1 ; f1 = dst[0]*src[0] = b*c, dst++, src++
ADDR3 F0,F1,F0 ; f0 = a*d + b*c
STF F2, *AR2++(1) ; *r++ = a*c - b*d
|| STF F0, *AR2++(1) ; *r++ = a*d + b*c

block:
The benchmark for 256-samples for this loop was 118.0 μs, or 7 clock cycles per complex sample (a saving of 4 clocks over the compiler generated code). The hand-optimized code contains 6 instructions, yet requires 7 clock cycles to execute, so there must be a pipeline conflict; probably due to the use of the same address registers in consecutive instructions.

A benchmark of the complex multiplication of a 65-sample vector took 54.1 μs for the compiler generated 11 clock cycle code (42.9 μs processing plus 11.2 μs overhead), and 38.0 μs for the hand-optimized 7 clock cycle code (27.3 μs processing plus 10.7 μs overhead). For delay correction of 10 cross-spectra every 15.625ms, the benchmarks represent processor loads of 54.1 μs/15.625ms × 10 × 100 = 3.5%, and 38.0 μs/15.625ms × 10 × 100 = 2.4%. So the hand-optimization saved about 1.1%.

The delay correction operation consisting of complex-valued correction vector generation and complex-valued vector multiplication for a 65-channel spectra will take 129 μs + 38 μs = 167 μs, and will increase the DSP load by 8.3% + 2.4% = 10.7%.

**10.7 TI-to-IEEE float-point format conversion**

The DSP uses a floating-point format that is TI specific. The host control CPU, and the rest of the control system expects data in IEEE 754 single-precision floating-point format. DSP floating-point numbers can be converted to IEEE format in C using bit-fields to map between the two data types, or in assembly using optimized routines that perform much the same operations. TI supplies a set of assembler routines that perform the conversion; a fast version (misses a few corner cases) and a complete version.

Benchmark tests performed with the assembly coded TI-to-IEEE fast conversion routine, showed no significant change in processing performance whether the assembly language text and data sections were in on-chip RAM, or off-chip SRAM (with the cache on). This indicates that the conversion routine is dominated by instruction processing, not data movement. Conversion of 512-samples of TI format to IEEE format took 1.85ms, or about 60 clocks per sample. The benchmark test involved a 512 samples loop around a C-callable assembler routine that converts a single sample. The loop time could be reduced by writing an assembler version that worked on an array of data.

The conversion from TI-to-IEEE format is required when the DSP transfers data to the control host. This transfer occurs every 250ms or 500ms. Transfer to the host of 10 correlations of in-phase and quadrature 65-channel complex spectra requires the conversion of 2600 floating-point samples. Conversion of these samples to IEEE takes 9.4ms. For transfers to the host every 250ms this represents a processor load of 3.8%, and for transfers every 500ms represents a processor load of 1.9%.
11 Peripheral Programming

Figure 3 shows the DSP memory map, with Figure 3(b) showing the locations of the DSP memory-mapped peripheral control registers. Table 3 and Figure 4 show the memory map, and memory-mapped peripherals on the COBRA digitizer and correlator boards.

Memory-mapped peripherals are convenient to work with, since their operation can usually be manually confirmed by writing directly to registers. For example, a block of memory can be filled with a known pattern, and then the DMA registers programmed to copy this data to another memory area. The copied area can be viewed to see that the DMA occurred. Once the correct register values are determined, software to implement the DMA can be written.

The following sections provide notes on DSP peripheral programming.

11.1 Memory-mapped peripherals

There are two common techniques for accessing peripheral registers from C code; structure overlay and register defines. Both techniques are briefly reviewed, along with the effect on the resulting assembly code. The techniques will be demonstrated for an example device; a fake serial port. A fake serial port has four registers; control, status, transmit, and receive. Let’s assume we have two fake serial ports located at address SERIAL_ADDR0, and SERIAL_ADDR1. The registers can be accessed from C code using the following methods:

C structure overlay:

typedef struct serial_registers {
    volatile unsigned int control;
    volatile unsigned int status;
    volatile unsigned int transmit;
    volatile unsigned int receive;
} serial_registers_t;

serial_registers_t *serial[2] = {
    (serial_registers_t *)SERIAL_ADDR0,
    (serial_registers_t *)SERIAL_ADDR1
};

This code creates a structure to define the registers layout, and then initializes an array of two structure pointers to point to the serial port addresses. The serial port registers can then be accessed by either using the pointers directly, eg. serial[0]->control = SERIAL_ENABLE, or by passing the address of a serial port’s registers to a function, eg. serial_control(serial_registers_t *port, int val) could be called using serial_control(serial[0], SERIAL_ENABLE) and the function body can then use the structure to access the registers.

The structure overlay technique compiles to assembly code that uses a DSP address register to access the structure elements. The assembly code loads, say AR0 with the structure address and then uses indirect addressing to access the registers, eg. *AR0 for control, **AR0(1) for status, **AR0(2) for transmit, and **AR0(3) for receive.

An alternative to initializing a pointer to a structure is to define a device specific section, and use the linker file to overlay the structure onto the registers (eg. see p5-11 [7]). However, even with this technique if you maintain an array of devices, then that array will typically need to contain pointers to the devices.
Register defines:

/* Serial port 0 */
#define SERIAL_CONTROL0  (*(volatile unsigned int *)SERIAL_ADDR0)
#define SERIAL_STATUS0   (*(volatile unsigned int *)(SERIAL_ADDR0+1))
#define SERIAL_TRANSMIT0  (*(volatile unsigned int *)(SERIAL_ADDR0+2))
#define SERIAL_RECEIVE0   (*(volatile unsigned int *)(SERIAL_ADDR0+3))

/* Serial port 1 */
#define SERIAL_CONTROL1  (*(volatile unsigned int *)SERIAL_ADDR1)
#define SERIAL_STATUS1   (*(volatile unsigned int *)(SERIAL_ADDR1+1))
#define SERIAL_TRANSMIT1  (*(volatile unsigned int *)(SERIAL_ADDR1+2))
#define SERIAL_RECEIVE1   (*(volatile unsigned int *)(SERIAL_ADDR1+3))

The register defines are placed in a header file that is then used by applications (this technique is very common when programming the AVR or ARM microcontrollers). Application code can write to, or read from, registers just like they were normal variables, eg. SERIAL_CONTROL0 = SERIAL_ENABLE. However, when it comes to manipulating multiple serial ports from a common set of routines, since the register defines have different names, then the interface tends to contain functions like serial_control(int channel, int val), and then the body contains an if-statement to select channel-specific code that is identical for each channel, except for the change in define names. The assembly code also has additional instructions to load each register address explicitly.

Using a structure overlay is the recommended technique for accessing peripherals, as it requires less code than the register defines method, and produces more efficient assembly output.

11.2 Bit-fields
A bit-field is a C construct that enables the definition of a type on a bit-by-bit basis. A typical application of bit-fields is to define bits in control and status registers. For example, lets assume that our fake serial port control register has two bits; transmitter enable, receiver enable, and the status register has two bits; transmitter busy, and receiver ready (all asserted high). Bit-fields describing the control and status registers, and a structure using those bit fields is

```c
typedef struct serial_control {
    int transmitter_enable : 1;
    int receiver_enable    : 1;
} serial_control_t;

typedef struct serial_status {
    int transmitter_busy : 1;
    int receiver_ready   : 1;
} serial_status_t;

typedef struct serial_registers {
    volatile serial_control_t control;
    volatile serial_status_t status;
    volatile unsigned int    transmit;
    volatile unsigned int    receive;
} serial_registers_t;
```
Code can use the bit-fields as follows:

```c
void serial_enable(serial_registers_t *serial)
{
    serial->control.transmitter_enable = 1;
    serial->control.receiver_enable = 1;
}
```

However code cannot assign to multiple bits simulatanously,

```c
void serial_enable(serial_registers_t *serial)
{
    serial->control = SERIAL_TRANSMITTER_ENABLE | SERIAL_RECEIVER_ENABLE;
}
```

as the compiler will complain of a type mismatch.

A register can be defined to have both an integer representation and a bit-value representation by using a union of the two types, eg.,

```c
typedef union {
    volatile unsigned int intval;
    struct {
        int transmitter_enable : 1;
        int receiver_enable : 1;
    } bitval;
} serial_control_t;
```

and then the union field has to be specified, eg. `serial->control.bitval.transmitter_enable = 1` or `serial->control.intval = SERIAL_TRANSMITTER_ENABLE`. The TI Peripheral Control Library uses this approach, but adds a layer of defines to hide the fact that the structures are being accessed via two levels. The assembly code for an integer value assignment loads an address register with the structure address and stores to the appropriate register using indirect addressing. The assembly code for bit-field assignment also uses an address register, and since the assignments are to individual bits, the assembly code contains register bit masking code.

There is no efficiency issue with the use of bit-fields versus structures. However, bit-fields are inherently non-portable, and some authors refer to their use as brain-dead ([A ‘C’ Test: The 0x10 Best Questions for Would-be Embedded Programmers](http://www.embedded.com/2000/0005/0005feat2.htm) by Nigel Jones, Embedded Systems Programming, May 2000). An example of this non-portability is the compilation of bit-fields by GCC 3.3.3 on an x86 CPU. Bit-field operations can compile to byte-wide writes eg. `movb %al, (%edx)`, where byte register `al` contains the result of a bit-field manipulation, and `edx` the address of the register to write the answer back to. When this byte operation occurs on a 32-bit wide register that does not accept byte-wide writes, as is common with PCI memory mapped registers, the 3 high-bytes of the word will be overwritten with garbage. So even though bit fields are defined in terms of 32-bit integers, the assembly output is not! Bit-fields are brain-dead, do not use them.

Control registers should be manipulated using structure overlays, and the manipulation of bits within those registers should be performed using `#defines` and bit masks.
11.3 Peripheral Control Library

The DSP built-in peripherals are mapped to structure overlays and addresses in the header files found in the TMS320C3x Peripheral Control Library [4]. A similar assembly language file is defined for the DSK as c3xmmrs.asm (C3x memory mapped registers). The Peripheral Control Library header files are compressed into a single .src file. To extract them use the following syntax:

```
ar30 -x <source filename>.src <header filename>.h
```

For example, to extract the serial port header:

```
ar30 -x prts30.src serprt30.h
```

And, to extract all of the header files

```
ar30 -x prts30.src
```

The Peripheral Control Library defines structures in terms of bit-fields. Although this technique works fine on the DSP, it is non-portable, and should generally be avoided. The headers defined by the library can however be used to generate new portable code based on structures and `#define` bit masks.

11.4 Code Composer memory manipulation

The DSP contains a 14-pin hardware emulator interface referred to as an XDS-510 emulator interface (this is equivalent to the JTAG interface on more modern processors or FPGAs). The Code Composer tool can communicate to the DSP over the XDS-510 interface using the PC-AT XDS-510 board and emulator cable. The Signum Systems XDS-510 parallel port controller can be used to connect their Chameleon debugger to the DSP. Both systems offer similar memory manipulation and code download and debug options.

The hardware emulator debug interface allows you to read and write from the DSP memory map, without code running on the DSP. This turns out to be very useful on newly manufactured boards, or boards with failures, since the emulator can check whether address or data bus bits are stuck (which did happen). However, be warned that the emulator interface can stop the DSP pipeline such that external bus control signals do not deassert with the specified DSP timing, causing problems the COBRA system controller FPGA state machines (this can make single stepping through code fail).

11.5 I/O Flags register (XF[1..0] control)

On the COBRA boards, the pins XF[1..0] are routed to buffers which drive two LEDs. These pins are used to indicate either error conditions or program states.

On the DSK, XF0 is located on pin 5 of the AIC header (JP1) and on pin 19 of the J2 header. XF1 is located on pin 18 of the J2 header (see pages B-9 to B10 in [6]). These general purpose I/O pins can be programmed as outputs directly from Code Composer. From Code Composer, select the View Registers window. Table 8 shows the register values required to toggle the two I/O pins high or low. An LED with a series resistor attached between XF0 or XF1 and ground (pin 21 of J1) allows one to measure that the XF pins are being programmed (a voltmeter works too).
Table 8: I/O Flags register programming.

<table>
<thead>
<tr>
<th>32-bit word</th>
<th>XF1 state</th>
<th>XF0 state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000022</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>0x00000026</td>
<td>LOW</td>
<td>HIGH</td>
</tr>
<tr>
<td>0x00000062</td>
<td>HIGH</td>
<td>LOW</td>
</tr>
<tr>
<td>0x00000066</td>
<td>HIGH</td>
<td>HIGH</td>
</tr>
</tbody>
</table>

Table 9: Timer registers.

<table>
<thead>
<tr>
<th>Register</th>
<th>Timer 0</th>
<th>Timer 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer Global Control</td>
<td>0x808020</td>
<td>0x808030</td>
</tr>
<tr>
<td>Timer Counter</td>
<td>0x808024</td>
<td>0x808034</td>
</tr>
<tr>
<td>Timer Period</td>
<td>0x808028</td>
<td>0x808038</td>
</tr>
</tbody>
</table>

11.6 Timers

The two timers are located in the DSP memory map at the addresses shown in Figure 3(b) and in Table 9 (the timers each take up 16 words of memory, but only three locations are used). A timer registers structure, and an array of pointers to the two DSP timers is (p12-4 [13]):

```c
typedef struct dsp_timer_registers {
    volatile unsigned int control;
    int reserved1[3];
    volatile unsigned int counter;
    int reserved2[3];
    volatile unsigned int period;
} dsp_timer_registers_t;
```

#define DSP_TIMER0_ADDR 0x808020
#define DSP_TIMER1_ADDR 0x808030

dsp_timer_registers_t *dsp_timer[2] = {
    (dsp_timer_registers_t *)DSP_TIMER0_ADDR,
    (dsp_timer_registers_t *)DSP_TIMER1_ADDR
};

The DSP User’s Guide provides details on the timers in Section 12.1, pp12-2 to 12-14 [13].
Table 10: Serial port registers.

<table>
<thead>
<tr>
<th>Register</th>
<th>Serial port 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Port Global Control</td>
<td>0x808040</td>
</tr>
<tr>
<td>FSX/DX/CLKX Serial Port Control</td>
<td>0x808042</td>
</tr>
<tr>
<td>FSR/DR/CLKR Serial Port Control</td>
<td>0x808043</td>
</tr>
<tr>
<td>Serial R/X Timer Control</td>
<td>0x808044</td>
</tr>
<tr>
<td>Serial R/X Timer Counter</td>
<td>0x808045</td>
</tr>
<tr>
<td>Serial R/X Timer Period</td>
<td>0x808046</td>
</tr>
<tr>
<td>Data transmit</td>
<td>0x808048</td>
</tr>
<tr>
<td>Data receive</td>
<td>0x80804C</td>
</tr>
</tbody>
</table>

11.7 Serial Port

The C31 DSP has a single serial port located in the DSP memory map at the address shown in Figure 3(b) and in Table 10. A serial port registers structure, and a pointer to the DSP serial port is (p12-16 [13]):

```c
typedef struct dsp_serial_registers {
    volatile unsigned int global_control;
    int reserved1;
    volatile unsigned int transmit_control;
    volatile unsigned int receive_control;
    volatile unsigned int timer_control;
    volatile unsigned int timer_counter;
    volatile unsigned int timer_period;
    int reserved2;
    volatile unsigned int transmit;
    int reserved3[3];
    volatile unsigned int receive;
} dsp_serial_registers_t;
```

#define DSP_SERIAL0_ADDR 0x808040

dsp_serial_registers_t *dsp_serial = (dsp_serial_registers_t *)DSP_SERIAL0_ADDR;


The DSP serial port is a synchronous serial port, so can not be used as an RS-232 interface without some extra work. The COBRA boards use the serial port in an SPI-compatible mode and communicate with an MAX3221E SPI-to-RS232 UART. The UART SPI interface can operate at up to 4.3MHz, so the DSP serial port is configured to run at a clock rate of 33MHz/8 = 4.125MHz. A 16-bit serial transaction (an SPI transaction is both a write and a read) requires 16/4.125MHz = 3.9µs, which is much faster than the interrupt latency, so communications over the DSP serial port are simply polled (since when the port is written to, the transmitter will always receive a word). The MAX3111E interrupt pin is routed to the system controller which in turn routes a conditioned version of the interrupt to one of the DSP interrupt pins (nominally INT3). The serial port drivers use the MAX3111E interrupts to trigger the transmission and receipt of characters over RS-232.
11.8 DMA Programming

The DSP has a single-channel DMA controller that can perform a single block transfer. The DMA memory-mapped registers are shown in Figure 3(b) and in Table 11. A DMA registers structure, and a pointer to the DMA registers is (p12-52 [13]);

```c
typedef struct dsp_dma_registers {
    volatile unsigned int control;
    int reserved1[3];
    volatile unsigned int source;
    int reserved2;
    volatile unsigned int destination;
    int reserved3;
    volatile unsigned int count;
} dsp_dma_registers_t;
```

#define DSP_DMA_ADDR 0x808000
dsp_dma_registers_t *dsp_dma = (dsp_dma_registers_t *)DSP_DMA_ADDR;


A DMA controller can be used to relieve the DSP core of data movement operations. An example of data movement relief on the COBRA boards would be the movement of lag data from the FPGAs to DSP internal memory, the movement of accumulated data from SRAM to internal memory, and from internal memory back to SRAM, the movement of results from SRAM to SDRAM. These DMA operations could be arranged to occur in parallel with the DSP performing an FFT on data stored in an internal RAM block (in such a way that the DMA to internal memory would not cause a conflict). The use-case of lag transfer requires that a DMA transfer be able to perform two data movements; one for the positive lags, and one for the negative lags. This type of DMA operation is generally referred to as a scatter-gather operation, and the list of source and destination addresses is referred to as a scatter-gather list.

The use-cases of using the DMA controller to transfer lag data, transfer data from SRAM to on-chip RAM, and data from SRAM to SDRAM would occur from within different tasks within the COBRA code. The DMA controller interface should implement some form of DMA request queue, where each request contains a scatter-gather list of transactions to perform, and a pointer to a function that is executed when the scatter-gather list is complete. This would allow a task to pass a function that say posted a semaphore when the transaction was complete, so the task can pend on that semaphore and be blocked. Different tasks can provide different callbacks. Labrosse’s *Embedded Systems Building Blocks* book has a timer API that implements the concept of call-backs, so that could be used as the basis of a DMA API (see Chapter 7, p229 [2]).
References


