Moving a TMS320C54x DSP/BIOS Application to the TMS320C55x DSP/BIOS Application

Stephen Lau, Vijaya Sarathy

ABSTRACT

The TMS320C54x™ processor is upwardly compatible to the TMS320C55x™. Although compatible, there are several differences that a DSP/BIOS™ application developer should be aware of. This document is intended to describe the different aspects that need to be considered when transitioning a C54x DSP/BIOS application to the C55x DSP/BIOS application.

Contents

1 Overview 3
1.1 Code Composer Studio Versions 3
1.2 Architectural Differences 3
2 Application Environment 4
2.1 C Environment 4
2.1.1 C54x C Environment 5
2.1.2 C55x C Environment 5
2.2 Generic Arguments 6
2.2.1 Example 6
2.3 Memory 7
2.3.1 Memory Access Sizes 7
2.3.2 Stack 8
2.3.3 Memory Model 11
2.4 Status Registers 12
3 DSP/BIOS Implementation 13
3.1 Initialization/Startup Sequence Differences 13
3.2 Real-Time Data Exchange (RTDX) 14
4 Conclusion 14
Appendix A Example: Migrating a 54x Code Composer Studio v2.0 DSP/BIOS Project 15
A.1 Introduction 15
A.2 Project Migration 19
A.3 DSP/BIOS Migration 19
A.3.1 Hardware Interrupt Manager (HWI) 21
A.3.2 MEM 21
A.3.3 Scheduling 21
A.3.4 RTDX 21
A.3.5 Generic Arguments 21

Trademarks are the property of their respective owners.

A.4 Code Issues . . . 25
A.5 Conclusion . . . 25

Appendix B Migrating a 54x Code Composer Studio v1.2 DSP/BIOS Project . . . 26
B.1 Stage 1: Migration from Code Composer Studio v1.2 to Code Composer Studio v2.0 . . . 26
B.2 Stage 2: Migration from C54x DSP/BIOS to C55x DSP/BIOS . . . 26

List of Figures

Figure 1. DSP/BIOS Mininum Addressable Data Unit . . . 8
Figure 2. Memory Section from Generated Linker Command File . . . 8
Figure 3. HWI Manager Dialog . . . 10
Figure 4. Task Stack Size Dialog . . . 10
Figure 5. Small Memory Model Data Memory Placement . . . 11
Figure A–1. copySwi and Host Pipe Notify Functions . . . 16
Figure A–2. Figure 1 copySwi software interrupt object . . . 16
Figure A–3. Input Host Pipe Object . . . 17
Figure A–4. Output Host Pipe Object . . . 18
Figure A–5. Configuration Tool Target Selection . . . 20
Figure A–6. RTDX Properties . . . 21
Figure A–7. inputReady Stub Function . . . 22
Figure A–8. outputReady Stub Function . . . 22
Figure A–9. 55x HST Input Object . . . 23
Figure A–10. 55x HST Output Object . . . 24

List of Tables

Table 1. CCStudio Versions Supported . . . 3
Table 2. C54x/C55x Comparison . . . 4
Table 3. Function Argument Categories . . . 5
Table 4. DSP/BIOS Modules Utilzing Generic Arguments . . . 6
Table 5. DSP/BIOS Provided Argument Type Converter . . . 6
Table 6. Specialized DSP/BIOS API for Generic Arguments . . . 7
Table 7. Stack Modes and Reset Vector . . . 9
Table 8. Assembly Macros for Register Context Manipulation . . . 12
Table 9. DSP/BIOS Initialization and Startup Sequence . . . 13
Table A–1. Pre-build Checklist . . . 25
Table B–1. Migration Items . . . 26

1 Overview

The TMS320C54x DSPBIOS application is upwardly compatible with the TMS320C55x DSP/BIOS application. The C55x DSP implements a superset of the C54x instruction set. DSP/BIOS for the C55x device implements the same application program interfaces (API) as DSP/BIOS for the C54x device. Though functionally equivalent, DSP/BIOS for the C55x has been coded as efficiently as possible for that architecture.

Thus, though compatible at the API level, there are several differences that a BIOS application developer should be aware of when moving from the C54x device to the C55x device. These differences are a result of:

- Hardware architecture
- Codegen tools (compiler and assembler)
- DSP/BIOS implementation

This document describes the different aspects that need to be considered when migrating a C54x DSP/BIOS application to the C55x DSP/BIOS application. Appendix A includes examples to illustrate this procedure.

1.1 Code Composer Studio Versions

Table 1 lists the different versions of Code Composer Studio™ (CCStudio) for the C5000 DSP platform, the DSP/BIOS version, and processor support.

Table 1. CCStudio Versions Supported

Code Composer Studio Version	DSP/BIOS Processor Support
Code Composer Studio C5000 v1.2	C54x
Code Composer Studio C5000 v2	C54x, C 55x

This document discusses migrating C54x DSP/BIOS assembly code to C55x DSP/BIOS assembly code; please refer to the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280). Appendix B explains migrating a C54x application developed in Code Composer Studio v1.2 to the C55x application developed in Code Composer Studio v2. For migration between Code Composer Studio C5000 v1.1 to V1.2, please refer to SPRA675, *Upgrading Applications to DSP/BIOS II*.

Besides DSP/BIOS support for the C55x DSP, Code Composer Studio v2 contains new features such as: Code Maestro, configuration management interoperability, and C++ support. DSP/BIOS now integrates the Chip Support Library for easier configuration and usage of device specific peripherals.

1.2 Architectural Differences

Fundamentally, the C55x device architecture is an extension of the C54x device architecture. It has been designed to improve efficiency and to maintain backward compatibility as well. Table 2 compares the two architectures.

Table 2. C54x/C55x Comparison

	C54x	C55x
MACs	1	2
Accumulators	2	4
Read buses	2	3
Write buses	1	2
Program fetch bus	1	1
Address buses	4	6
Program word size	16 bits	8/16/24/32/40/48
Data word size	16 bits	16 bits
Auxiliary Register ALUs	2 (16-bit each)	3 (16-bit each)
ALU	1 (40-bit)	1 (40-bit) 1 (16-bit)
Auxiliary Registers	8	8
Data Registers	0	4
Memory Space	Separate Program/Data	Unified space

It should also be noted that the C55x DSP has a fully-protected pipeline and the capability to do full 32-bit memory write in a single cycle. The multiple memory and stack modes will be discussed elsewhere in this document. For further information on the C55x DSP, please refer to the *DSP CPU Reference Manual* (SPRU371).

2 Application Environment

2.1 C Environment

With Code Composer Studio V2 , the C compiler has new features such as C++ support and more efficient optimization. With Code Composer Studio V2, the C54x device and Code Composer Studio V2 55x C environments are constructed differently. The C compiler register conventions dictate how the compiler uses registers and how values are preserved across function calls. The parent function is the calling function (caller). The child function is the called function.

DSP/BIOS fully complies with the C compiler conventions. More specificaly, the prescribed “function caller/callee” norms are strictly obeyed. This includes the proper usage of processor registers for passing arguments, handling stack alignment nuances, etc.

Only caller-preserved registers are used by DSP/BIOS. Asynchronous context switch preserves all caller-function preserved registers. Multi-tasking (TSK) context switches preserve only the called-function preserved registers, as per the C specification.

A list of utility macros is shown in Table 8 and are provided in \bios\include\c55.h55 to work with register context. They are useful when an application needs to alter the C environment as they provide a method to conveniently save and restore context.

The function structure and calling conventions of 55x differs from that of the 54x. Please refer to the *TMS320C55x Optimizing C Compiler User's Guide* (SPRU281) and the *TMS320C54x Optimizing C Compiler User's Guide* (SPRU103) for details.

2.1.1 C54x C Environment

The C54x C Compiler does not strongly typify function arguments in terms of which registers are used to pass them fromthe calling function to the called function. The C54x C environment uses a generic parameter/register binding where the first parameter is always passed in accumulator A. The remaining arguments are passed onto the stack with the top of the stack holding the left-most remaining argument.

In case of functions with variable number of arguments (ellipsis), the last explicitly declared argument onwards is placed on the stack. This means that for ellipsis functions, even if there is only one explicitly declared argument, it must go on the stack and not the accumulator A.

If the function returns a structure, the caller allocates space for the structure and then passes the address of the return space to the called function in accumulator A. All returns are passed via the accumulator A.

2.1.2 C55x C Environment

The C55x C Compiler strictly binds function arguments to processor registers. This allows the compiler to optimally utilize the data address, generation unit resources for improved efficiency.

Function arguments are classified into three broad categories: pointers, 16-bit quantities, and 32-bit quantities. A close look at Table 3 and the register/argument association reveals the dual role of the AR0, AR1 registers.

The (X)AR0 and (X)AR1 registers are specialized to carry data pointers, but are also used to pass 16-bit (non-pointer type) quantities as well, based on availability.

Note: Because of the binding order, care must be taken when interfacing assembly functions to C code.

Table 3. Function Argument Categories

Category	Register Binding Order
Specialized Data Pointers (int*, long*, etc.)	(X)**AR0**, (X)**AR1**, (X)AR2, (X)AR3, (X)AR4
16-bit quantities (char, short, int)	T0, T1, **AR0**, **AR1**, AR2, AR3, AR4
32-bit quantities (long, float, double, function pointer)	AC0, AC1, AC2, AC3

For efficiency, the allocation strategy on the C55x DSP attempts to maximize the utilization of hardware registers. If the first argument is a data pointer, it will be assigned the first unallocated auxiliary register, (X)AR register. If all (X)AR registers (XAR0–XAR4) are used, then the pointer is passed onto the stack. If the first argument is a 16-bit, non-pointer type quantity, the compiler will check the temporary registers T0 and T1. If T0 or T1 are free, then the 16-bit value is assigned the first one that is free. If neither T0 nor T1 is free, then the compiler finds the first free auxiliary register (AR0 to AR4) register in order to pass the integer argument. If there are no free auxiliary registers, then the integer will be passed onto the stack.

The C54x device does not make a distinction between an integer and a data pointer; if the first argument were an integer, it would be passed in accumulator A. Because the allocation strategy of the C55x device utilizes more registers than the C54x device, stack usage for argument frames is optimized.

2.2 Generic Arguments

Many DSP/BIOS objects call functions with a generic argument prototype. Table 4 lists the various DSP/BIOS modules that utilize generic arguments. An issue may arise when the DSP/BIOS object calls a function or DSP/BIOS API that does not match the generic argument prototype.

On the C54x DSP, a generic argument (of type Arg or Void *) is acceptable because a 16-bit passed value of type Uns could not be distinguished from a value of type Arg, and the location of the passed parameters remains constant.

On the C55x DSP, generic arguments (of type Arg or Void *) are treated as pointer quantities and are thus placed into the (X)ARx registers, as per the compiler conventions. These pointer quantities can be either 16-bit or 23 bit. This increases hardware resource utilization, but complicates the writing of called functions.

Table 4. DSP/BIOS Modules Utilzing Generic Arguments

Module	Uses Generic Arguments?
HST	Yes
PIP	Yes
PRD	Yes
SWI	Yes
TSK	Yes

As with DSP/BIOS objects, C55x DSP application code that uses 16-bit quantities passed as function arguments of type (Void*) or Arg will need to be modified to account for the more efficient register usage. Two potential methods, stub function or function modification, are shown below.

To ease the burden of dealing with type conversion, C55x DSP/BIOS provides type converters as specified in Table 5. The prototypes for these type converters are specified in the DSP/BIOS include file std.h, as found in bios/include.

Table 5. DSP/BIOS Provided Argument Type Converter

API	Description
ArgToInt(A)	Converts *Arg A* into an *Integer*
ArgToPtr(A)	Converts *Arg A* into a *Pointer* data type

2.2.1 Example

Consider the DSP/BIOS PIP object notifier function prototype *Fxn(Arg, Arg*). If we choose to use the DSP/BIOS function SWI_andn as the notifier function, the API does not match the generic prototype: *SWI_andn(SWI_Handle swiPtr, Uns key)*.

On the C54x device, this is acceptable, as the parameters are exactly where the called function expects them. On the C55x device, SWI_andn() is expecting the second argument to be of type Uns, and per compiler convention this second parameter will be in T0 (not XAR1). However, the compiler has interpreted the second argument to be of type Arg and placed the variable into the XAR register.

On the C55x device, there are two solutions:

1. Stub Function – Create a small function prototype that has a generic argument prototype. This stub function can then either reference objects directly or cast the passed Arg to an appropriate type when calling a function. For example:

   ```
   pipNotifierStub(Arg value1, Arg value2){
         SWI_andn(value1, (ArgToInt)(value2))
   }
   ```

   The stub function utilizes the ArgToInt type conversion macro as detailed in Table 5.

2. Modify Function – Create a new function that can access the arguments in the expected register locations. Table 6 lists the specific DSP/BIOS APIs designed for use with generic arguments. Although not essential for the C54x device, the equivalent API is available to ease application portability.

   Note: On the C54x and C6xx DSPs, the SWI_andnHook and SWI_orHook functions do not incur any code size or performance penalties. On the c55x DSP, the SWI_andnHook and SWI_orHook functions incur an overhead of one word.

Table 6. Specialized DSP/BIOS API for Generic Arguments

API	Note
SWI_andnHook	SWI_andnHook is a specialized version of SWI_andn. Typically used for configured DSP/BIOS objects which pass information as type Arg.
SWI_orHook	SWI_orHook is a specialized version of SWI_or. Typically used for configured DSP/BIOS objects which pass information as type Arg.

2.3 Memory

The C55x DSP can address up to 16M bytes of memory. When the CPU uses program space to read program code from memory, it uses 24-bit addresses to reference bytes. When your program accesses data space, it uses 23-bit addresses to reference 16-bit words. In both cases, the address buses carry 24-bit values, but during a data-space access, the least significant bit on the address bus is forced to 0. Data space is divided into 128 main data pages (0 through 127). Each main data page has 64K addresses.

2.3.1 Memory Access Sizes

The C54x DSP accesses both program and data as 16-bit words. On the C55x DSP, program is addressed as 8-bit bytes, while data is addressed as 16-bit words. DSP/BIOS standardizes on using minimum addressable units (MAU).

The MAU is the smallest unit of data storage that can be read or written by the CPU in data memory. The number of bits in an MAU varies with different DSP devices; for example, the MAU for the C54x and the C55x DSPs is a 16-bit word, and the MAU for the C6000 platform is an 8-bit byte.

DSP/BIOS handles all required words/bytes unit conversions in order to agree with the compiler, assembler, and linker requirements. DSP/BIOS application developers do not need to make any special accommodations to take advantage of this.

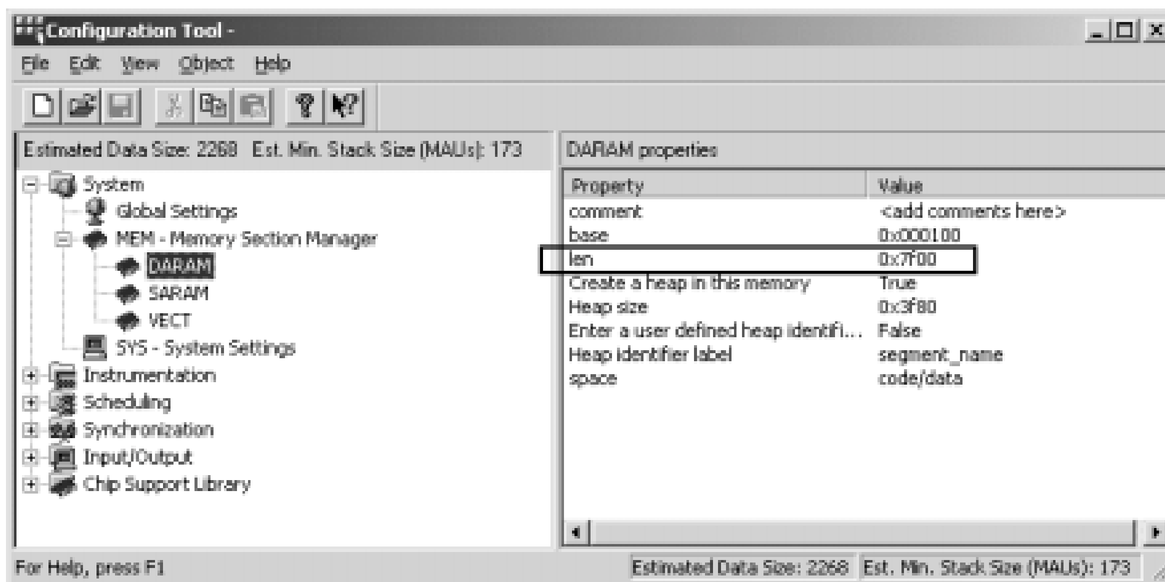


Figure 1. DSP/BIOS Mininum Addressable Data Unit

On the C55x device, all addresses and sizes supplied in the linker command file should be byte addresses. This is true for both code as well as data sections. In contrast to this, the C55x Assembler treats data as 16-bit units (words), whereas program code is treated in terms of 8-bit units (bytes). DSP/BIOS helps deal with this delicate difference by *always* allowing the user to supply *size-related* information in terms of 16-bit quantities (MAUs) and appropriately translates it based on whether this information is fed to the Linker (via generated Linker command file) or fed to the Assembler (via generated Assembly source files). An example of this can be seen in Figure 1 and Figure 2. In Figure 1, the user has input the DARAM memory segment length as 0x7f00 MAUs (i.e., 16-bit quantities). When the DSP/BIOS Configuration database is saved, the resulting Linker command file will contain the DARAM size information in terms of 8-bit bytes. This is seen in Figure 2.

```
MEMORY {
   DARAM:       origin = 0x200,         len = 0xfe00
   SARAM:       origin = 0x10000,       len = 0x40000
   VECT:        origin = 0xffff00,      len = 0x100
}
```

Figure 2. Memory Section from Generated Linker Command File

2.3.2 Stack

The C55x DSP supports two 16-bit software stacks known as the stack and the system stack. SPH holds the 7-bit main data page of memory, and SP points to the specific word on that page.

For an access to the stack, the CPU concatenates SPH with SP to form the XSP. The XSP contains the 23-bit address of the value last pushed onto the data stack. The CPU decrements SP before pushing a value onto the stack and increments SP after popping a value off the stack.

When accessing the system stack, the CPU concatenates SPH with SSP to form XSSP. XSSP contains the address of the value last pushed onto the system stack. The CPU decrements SSP before pushing a value onto the system stack and increments SSP after popping a value off the system stack. SPH is not modified during system stack operations.

The C54x DSP has a single stack mode and one stack pointer, SP. As can be seen in Table 7, the C55x stack has three modes of operation. Please refer to the *TMS320C55x DSP CPU Reference Guide* (SPRU371), for additional information.

Table 7. Stack Modes and Reset Vector

Mode	Description	Vector	DSP/BIOS Stack Mode
Dual 16-bit stack with fast return	Data stack and the system stack are independent. SP and SSP are not synchronized. The registers RETA and CFCT are used to implement a fast return.	XX00–XXXX	USE_RETA
Dual 16-bit stack with slow return	Data stack and the system stack are independent. SP and SSP are not synchronized. The registers RETA and CFCT are NOT used for program counter save/restore.	XX01–XXXX	NO_RETA
32-bit stack with slow return (default)	The data stack and the system stack act as a single 32-bit stack. Both SP and SSP are increment/decremented by the same amount to keep them synchronized. The registers RETA and CFCT are NOT used.	XX10–XXXX	54X_STK

Regarding the C55x stack, the RESET vector also holds the Processor Stack configuration setting. The first byte in the reset vector is interpreted as listed in Table 7 by the processor.

Note: Use of either of the dual 16-bit stack modes in CCStudio v2 requires the use of the reset command from the debug menu.

For the C55x version of DSP/BIOS, it is possible to select the stack mode from the HWI manager properties dialog. Figure 3 illustrates the dialog box. The three stack modes are transparently handled by the DSP/BIOS Kernel scheduler.

Note: Dual 16-bit mode stack operations are distinct from the dual 16-bit mode Arithmatic-Logic Unit (ALU) operations.

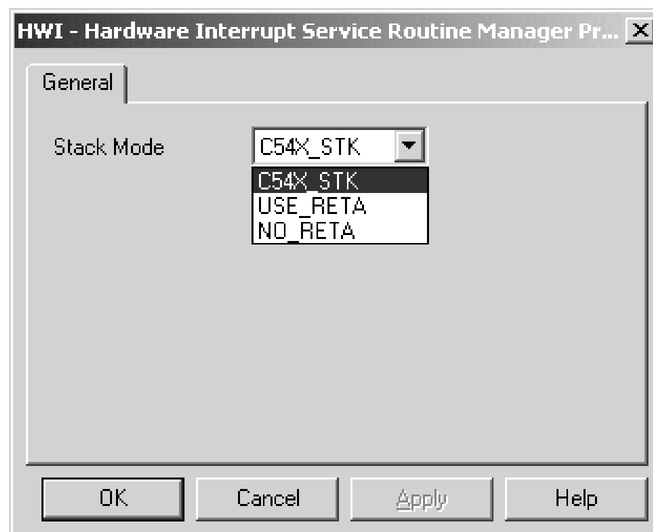


Figure 3. HWI Manager Dialog

In the DSP/BIOS memory manager, it is possible to independently specify the size of the stack and the system stack in MAU. The memory manager also allows specification of the memory segment for placement of the data stack and the system stack. The size and location of the stacks, as illustrated in Figure 4, can also be specified for individual tasks in the TASK properties.

The stack pointers are aligned to an even address boundary to comply with C requirements. This ensures that 32 bit quantities can be accessed in true Most Significant Word:Least Significant Word (MSW:LSW) form. For more details on the C55x stack architecture and its implications please refer to the *TMS320C55x CPU Reference Guide* (SPRU371).

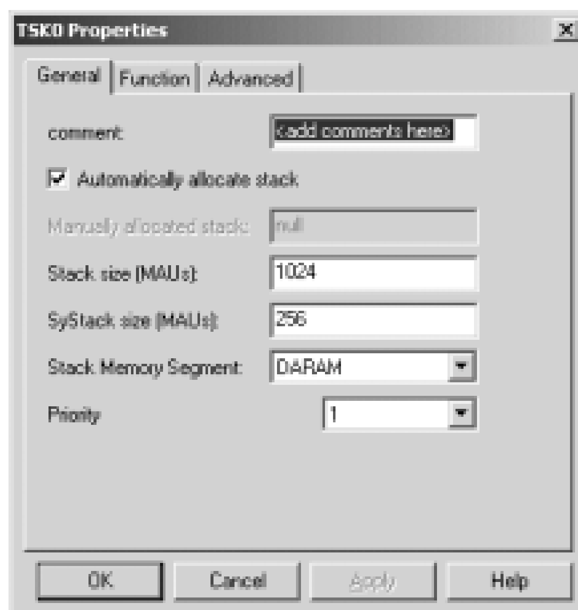


Figure 4. Task Stack Size Dialog

2.3.3 Memory Model

The C55x significantly increased the amount of addressable data memory. This section looks at the details for each memory model on the C55x.

2.3.3.1 Small

On the C55x device, small memory model, data, stack, and dynamic memory heap(s) are constrained to a single page of 64k words size, and can be on any one of the 128 pages available. This also applies to the heap(s).

There is no restriction on the size or placement of program code. The C compiler uses 16-bit data pointers to access data. Code pointers are always 24 bits and occupy two words when stored in memory.

The extended auxiliary registers, XARn[23:16], are initialized to point to the page that contains the .bss section. In the small model, these extended auxiliary registers should not be changed throughout the execution of the program.

Care should be exercised when placing the data sections or heaps in memory to reduce the likelihood of the XARn registers wrapping around. As illustrated in Figure 5, wraparound occurs in the small memory model because the XARn[23:16] registers are not incremented when crossing the 64K boundary. Thus, only the ARn[15:0] registers are used, which only allows for 64K of addressing. Thus, in the small memory model, one must be prevent data, stack and heaps from straddling 64k pages.

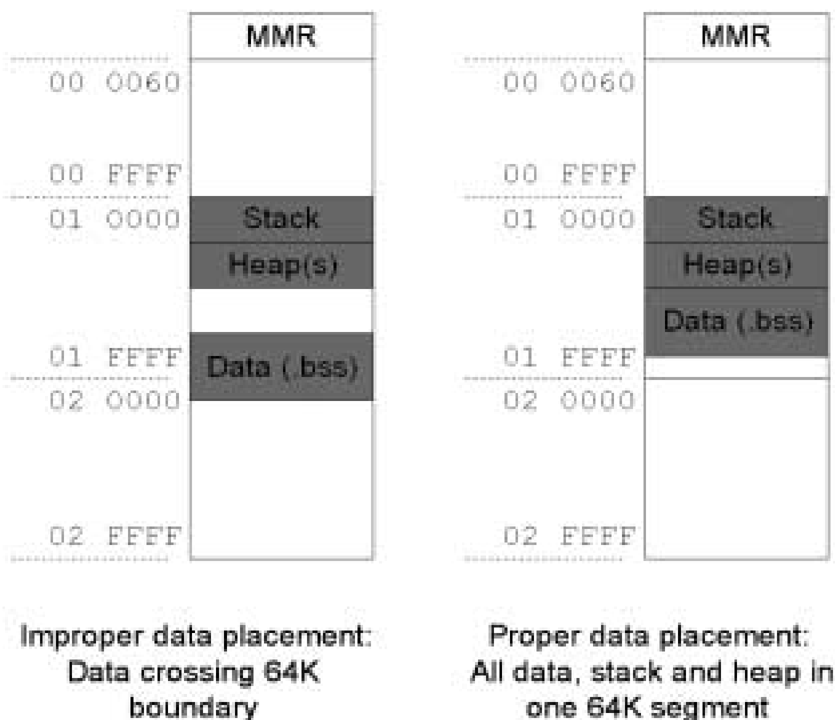


Figure 5. Small Memory Model Data Memory Placement

2.3.3.2 Large

The large memory model supports an unrestricted placement of data and memory heap(s) over all of the 128 pages. There is no restriction on the size or placement of program code. Use of the large memory model requires the use of the –ml compiler shell option.

Program builds must also use the appropriate large memory model libraries. It is not possible to use both small and large memory model libraries in a single executable.

Data pointers are 23 bits and occupy two words when stored in memory. Code pointers are 24 bits and occupy two words when stored in memory.

As with the small memory model, care should be exercised when placing the data section and heaps in memory to reduce the likelihood of having the XARn registers wrap around. Unlike the small memory model, it is possible to have different 64K pages holding different data sections or heaps at the same time.

2.4 Status Registers

The C54x device has two status registers (ST0 and ST1) and a single processor mode status register (PMST). The C55x device has four status registers (ST0_55, ST1_55, ST2_55, and ST3_55). The C55x device maintains compatibility with the C54x device by having some status registers available in two locations. One location is native to the C55x device, while the other maintains compatibility with the C54x device. Please refer to the *TMS320C55x DSP CPU Reference Guide* (SPRU371) for more information on the status registers.

DSP/BIOS does have some requirements for the status register bits, details of which can be found in the *DSP/BIOS API Guide* (SPRU404). A macro, C55_setBiosSTbits has been included to simplify setting of the status registers to the DSP/BIOS defaults. Table 8 is a list of utility macros that are provided for register context manipulation (please see the c55.h55 file which is typically found in C:\ti\c5500\bios\include)

Table 8. Assembly Macros for Register Context Manipulation

DSP/BIOS API	Description
C55_setBiosSTbits	This macro sets processor Status register bits as required by DSP/BIOS.
C55_saveBiosContext	This macro saves on the stack all registers declared as Preserved in the DSP/BIOS multi-threading context.
C55_restoreBiosContext	This macro restores from the stack all registers declared as Preserved in the DSP/BIOS multi-threading context.
C55_saveCcontext	This macro saves on the stack all registers declared as caller preserved in the C compiler conventions.
C55_restoreCcontext	This macro restores from the stack all registers declared as caller preserved in the C compiler conventions.

Note: DSP/BIOS does not modify the Global status bits. Global status bits are those options that can change the environment of the entire system (ex: ST3_HINT).

3 DSP/BIOS Implementation

3.1 Initialization/Startup Sequence Differences

Table 9 summarizes the DSP/BIOS initialization steps on C54x and C55x DSPs following recognition of processor RESET. The _c_int00 label denotes the start of this boot procedure in DSP/BIOS. This label denotes the target address in the processor RESET vector.

Table 9. DSP/BIOS Initialization and Startup Sequence

No.	C54x DSP/BIOS	C55x DSP/BIOS
1	Disable Maskable interrupts globally . INTM=1	Similiar to C54x
2	Clear individual interrupt mask bits Clear IMR	Clear individual interrupt mask bits Clear IER0 and IER1
3	Initialize Stack Pointer Set SP to bottom of user stack. Align SP to even address boundary	Initialize User and System Stack pointers Set XSP to bottom of user stack. Set XSSP to bottom of system stack. Align both XSP, XSSP to even address boundaries. Note: SPH is shared across XSP, XSSP
4	Initialize Processor Status Register ST: SXM=1, CPL=1, ARP=0, OVM=0, C16=0, CMPT=0, FRCT=0	Initialize Processor Status Registers ST1: CPL=1, M40=0, SATD=0, SXMD=1, C16=0, FRCT=0, C54CM=0 ST2: ARMS=1, RDM=0, CDPLC=0, AR[0–7], LC=0 ST3: SATA=0, SMUL=0, SST=0
5	Initialize Interrupt Vector Segment Register Set PMST to desired value.	Initialize Interrupt Vector Table Pointers Set IVPD, IVPH to desired value. Since RTDX initialization can trigger interrupts, its necessary that IVPD, IVPH are setup ahead of (step #9) PINIT processing.
6	nil	Initialize Extended Auxiliary Address Registers Setup XAR[0–7], XCDP, XDP to point to start of .bss section
7	If CINIT table exists, process it.	Similiar to C54x
8	Initialize DSP/BIOS Modules	Initialize DSP/BIOS Modules
9	If PINIT table exists, process it. This involves calling the C++ global object constructors. Therefore, the Compiler prescribed Caller/Called function responsibilities are strictly obeyed.	Similiar to C54x
10	Setup arguments for main(argc, argv, envp) function. Int argc: accumulator A Char* argv[]: *SP(#0) Char* envp[]: *SP(#1)	Setup arguments for main(argc, argv, envp) function. Int argc: T0 Char* argv[]: (X)AR0 Char* envp[]: (X)AR1
11	Call main(argc, argv, envp)	Call main(argc, argv, envp)

Table 9. DSP/BIOS Initialization and Startup Sequence (Continued)

No.	C54x DSP/BIOS	C55x DSP/BIOS
12	Startup DSP/BIOS Modules	similar to C54x
13	Drop into DSP/BIOS IDL_loop.	similar to C54x

DSP/BIOS now includes the Chip Support Libraries (CSL). The CSL is a collection of convenient modules that work with various on-chip peripheral resources. If CSL is configured, then the timer is configured through the CSL module. If CSL is not configured, the CLK module configures the timer.

At reset, the C55x DSP is placed in the programmed stack configuration. A listing of available stack modes can be found in Table 7.

The C55x DSP has two sets of interrupts: DSP and Host based interrupts. The DSP interrupts use IVPD register to point to the start of the vector table for them, while the Host interrupts use the IVPH register.

The C55x DSP supports 32 interrupts by making use of the mask bits in IER0 and IER1. The C54x DSP only had one mask register, the IMR.

3.2 Real-Time Data Exchange (RTDX)

RTDX allows the transfer of data between the target to the Host without halting the processor. This is extremely useful for non-intrusive debugging of real time events.

On the C54x device, DSP/BIOS determines if there is new information coming through from the RTDX channel by polling the communication buffers while inside DSP/BIOS IDL loop. On the C55x device, RTDX is interrupt driven. The HWI_INT25 (DATALOG ISR) is used for RTDX on the C55x DSP.

4 Conclusion

Migration of a DSP/BIOS C54x application to DSP/BIOS C55x is possible with a minimum of effort. The C54x and C55x DSP/BIOS APIs are virtually identical, making application portability convenient. Differences essential to DSP/BIOS application developer were discussed and an example is included in Appendix A to highlight the migration methodology.

Appendix A
Example: Migrating a C54x Code Composer Studio v2.0 DSP/BIOS Project

A.1 Introduction

Through the use of an example, this application note will step through the process of migrating a C54x DSP/BIOS Code Composer V2 project to the C55x. The chosen example has been installed with Code Composer Studio. The example chosen is the DSP/BIOS copy example found in c:\ti\examples\sim54xx\bios\copy.

This example will highlight:

- Project migration
- CDB file modifications
- API similarities between processors
- Migration from C54x simulator to C55x simulator

This example assumes that the reader has a thorough understanding of:

- C54x and C55x code generation tools and their respective C environments
- DSP/BIOS objects
- Code Composer Studio projects
- C54x simulator and C55x simulator

Hardware specific portions of the application will not, in general be easily portable. The chip support library (CSL) is designed to provide a level of abstraction between the architectures, but care should be taken when migrating hardware specific code sections. Please check the *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401).

When performing this migration, there are three categories of issues:

- Architecture specific (C54x code to C55x code migration)
- Device specific (5402, 5510, etc.)
- DSP/BIOS specific (CDB, etc.)

In this example, we will move the copy example from the C54x simulator to the C55x simulator. In this example, there are two DSP/BIOS host pipe objects, an input pipe and an output pipe. Data is copied from the input pipe to the output pipe inside the software interrupt (SWI). This SWI is called copySwi and only runs when the two pipes are ready for transfer. Both host pipes will be ready for a transfer when there is at least one filled frame on the input pipe and there is at least one empty frame on the output pipe.

This is implemented by making use of the DSP/BIOS *SWI_andn()* API to conditionally post the *copySwi* via the pipe reader notifier function on the pipe *inputPipe* and write notifier function on the pipe *outputPipe*. Each notifier call to SWI_andn() will clear a different bit in the mailbox. The software interrupt copySWI will be posted to run when the mailbox value becomes zero. Thus, the copySWI will only run when the input pipe has a valid frame and the output pipe has an empty frame.

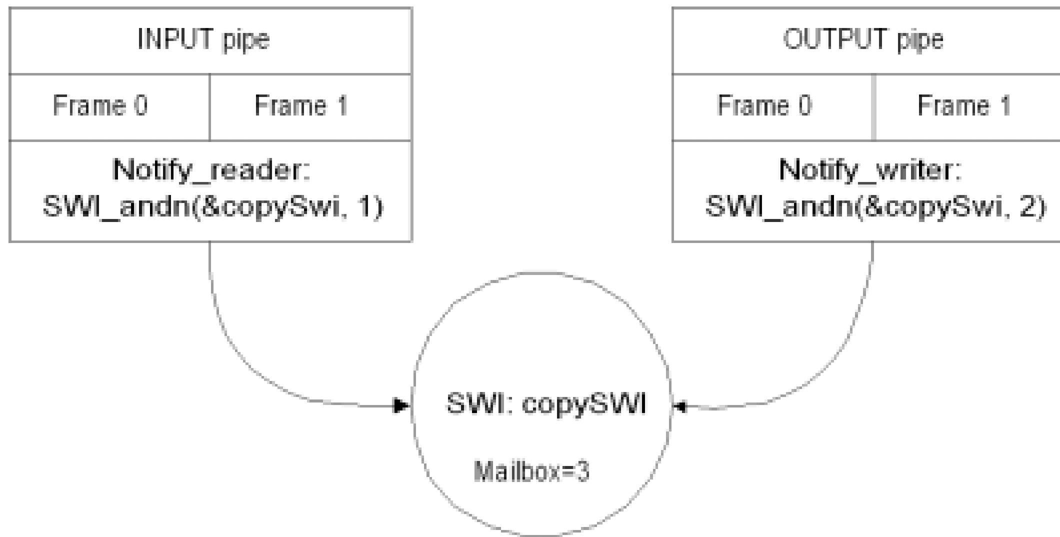


Figure A–1. copySwi and Host Pipe Notify Functions

The software interrupt is setup as illustrated in Figure A–2. As can be seen in Figure A–3 and Figure A–4, the notifier functions are setup as follows:

- *input*: reader notifier function is *SWI_andn*(*©Swi, 1*)
- *output*: writer notifier function is *SWI_andn*(*©Swi, 2*)

Note: Included with Code Composer Studio is a utility called CDBprint which produces a text-file suitable for documentation of DSP/BIOS object configuration.

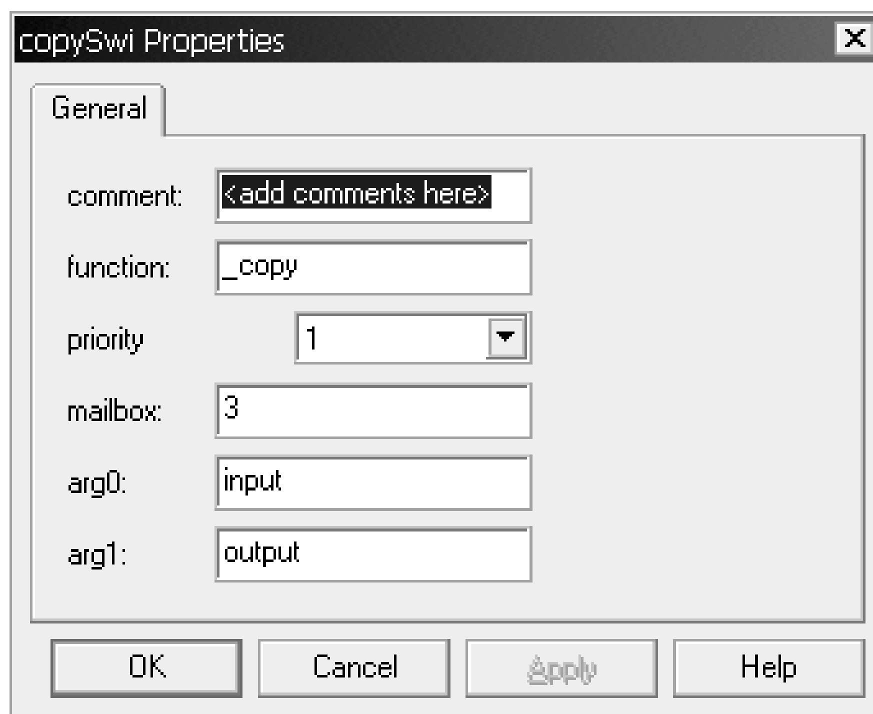
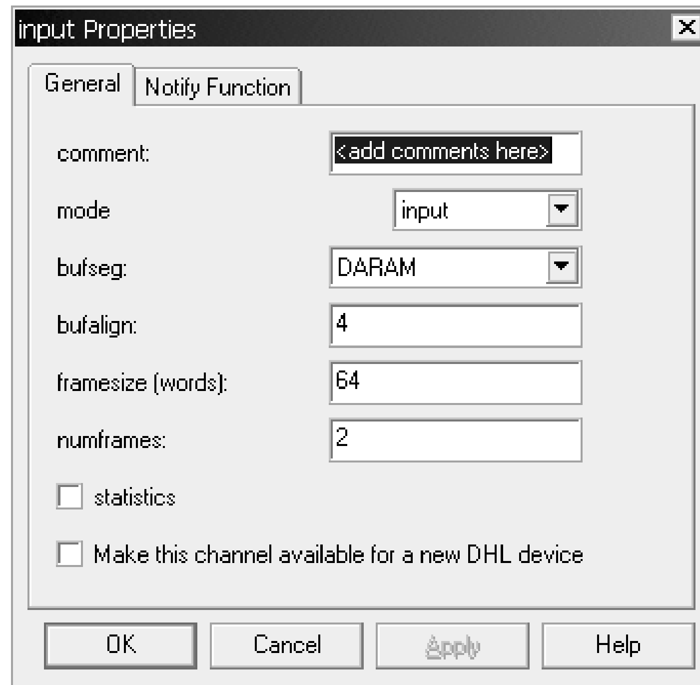


Figure A–2. Figure 1 copySwi software interrupt object



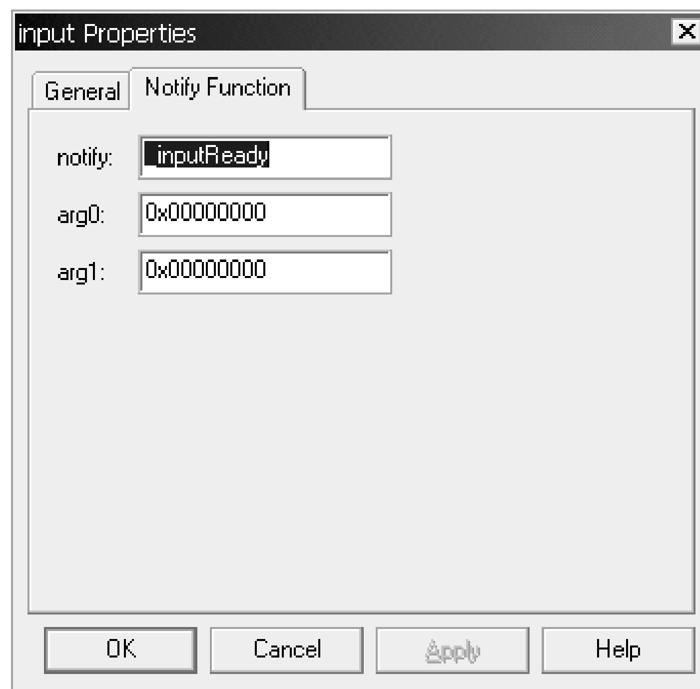


Figure A–3. Input Host Pipe Object

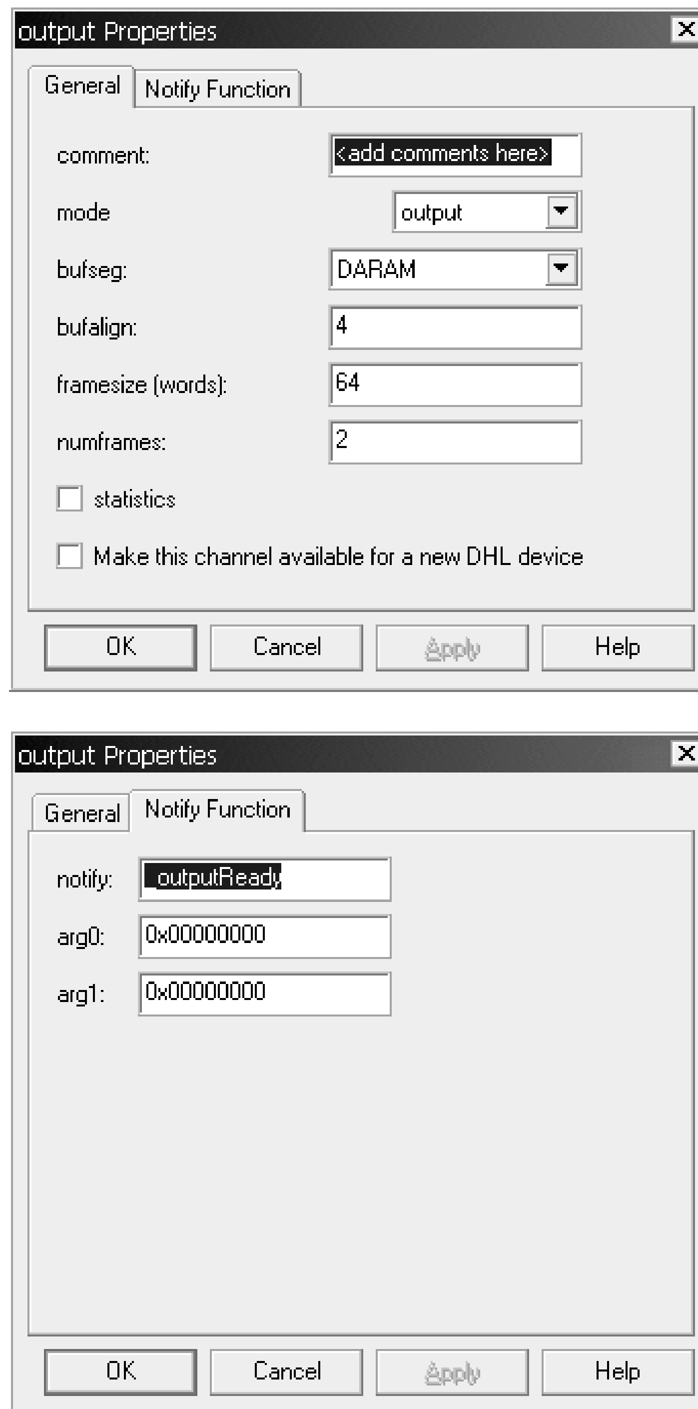


Figure A–4. Output Host Pipe Object

A.2 Project Migration

We begin the project migration by creating a new directory that will be used to store the new C55x project.

The existing essential project files consists of:

- Copy.pjt
- Copy.c
- Copy.cdb
- Copycfg.s54
- Copycfg.h54
- Copycfg.cmd
- Copycfg_c.c

Create a new directory for the C55x project. Copy the following source files into a new directory:

- Copy.c

  Note: If there are custom libraries or object code, they should be migrated to the C55x before being added to the project.

Create a new project in Code Composer Studio. Copy the relevant project options and defines into the new project. When creating a new Code Composer Studio 2 project, the target setup in Code Composer Setup determines which platform the project will utilize. Before creating a project for this example, make sure Code Composer Studio is configured for the c55x Simulator.

For additional information on migrating a Code Composer Studio 1.0 or 1.2 project to Code Composer Studio 2, please refer to the application report *Migrating CCS 1.20/CCS 1.0 Projects to CCS 2.0* (SPRA745). For further information on upgrading from Code Composer V1, please refer to the application report *Upgrading to DSP/BIOS II* (SPRA675).

A.3 DSP/BIOS Migration

As can be seen in Figure A–5, when you create a new DSP/BIOS configuration database files (CDB) file, the configuration tool prompts the user for information on the target. This information is used to select a generic DSP/BIOS CDB file that is customized by the user for their application.

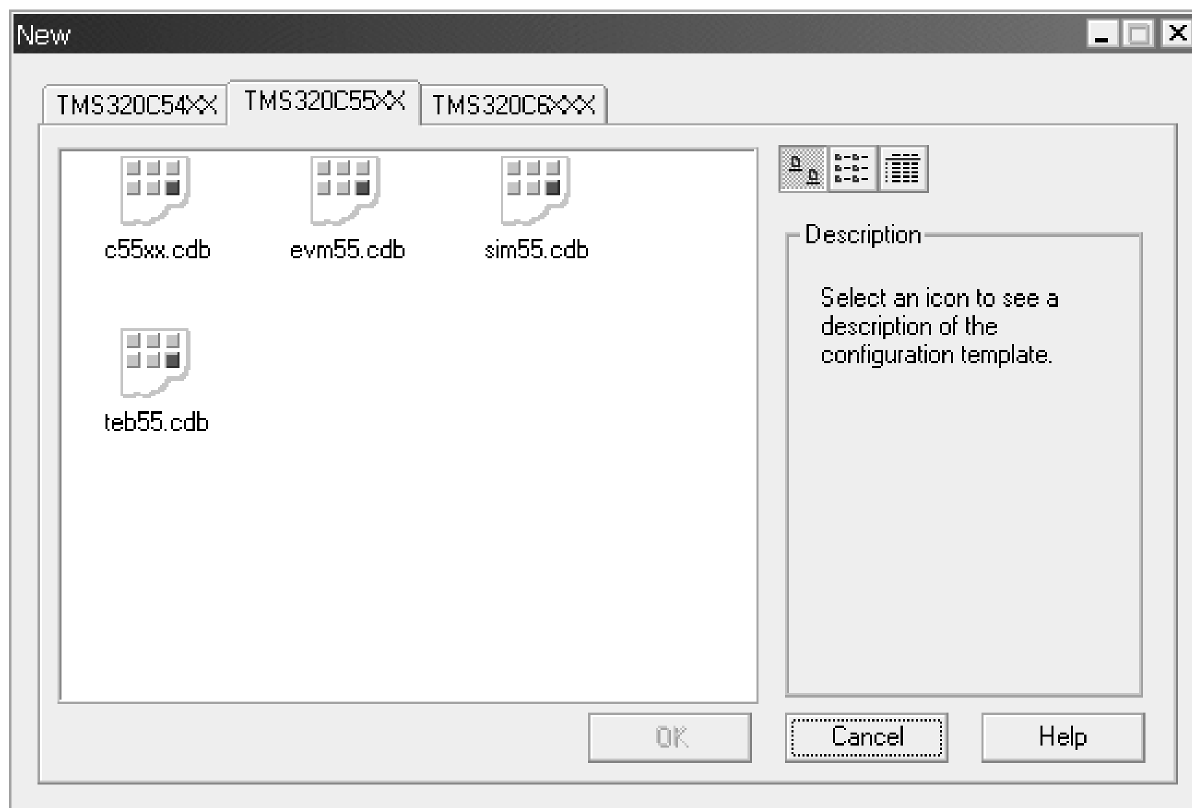


Figure A–5. Configuration Tool Target Selection

In addition to the DSP/BIOS application configuration, this configuration database file contains target specific information. Because the CDB file contains target specific information, the one from the existing CDB file cannot be used.

Create a new DSP/BIOS CDB for the appropriate target file and create DSP/BIOS objects with parameters similar to the ones in the existing C54x CDB file. In this example, the example will be migrated to the C55x simulator, so the C55x simulator template (sim55.cdb) should be chosen.

Note: The existing parameters are obtainable by either opening the existing CDB file, or by using the CDBprint utility to obtain the configuration parameters.

After saving the new C55x DSP/BIOS CDB file five files should result:

- Copy.cdb
- Copycfg.s54
- Copycfg.h54
- Copycfg.cmd
- Copycfg_c.c

Note: The five files resulting from the configuration tool can be checked into a configuration management system if one is used. Only the configuration tool should modify the CDB file. The other files should not be directly modified as any changes will be automatically overwritten if the CDB file is updated.

Particular DSP/BIOS modules should be checked to ensure proper migration.

A.3.1 Hardware Interrupt Manager (HWI)

Start by selecting the appropriate stack model. As can be seen in , the 32-bit stack model is initially selected. DSP/BIOS supports all of the different stack modes, and the application developer only needs to fill in the appropriate interrupt service routine.

A.3.2 MEM

Check to confirm that the memory sections are configured appropriately for the intended target.

A.3.3 Scheduling

Check to confirm that all scheduling objects are at the correct priority levels.

A.3.4 RTDX

Check to confirm that the RTDX mode is appropriate for the target. Since this example is based on the simulator, the simulator should already be selected.

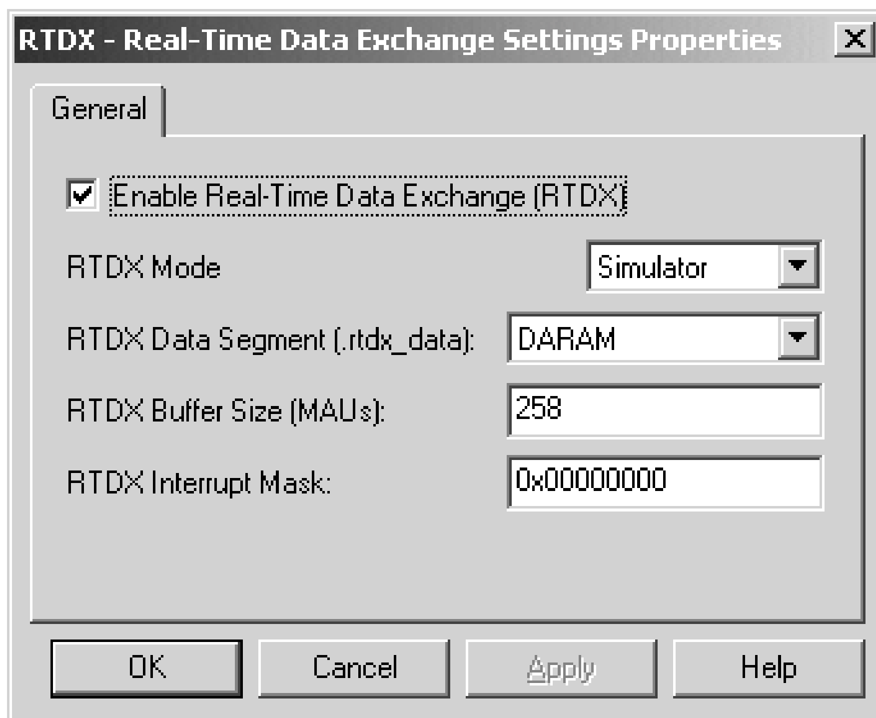


Figure A–6. RTDX Properties

A.3.5 Generic Arguments

In the copy example program, there are only two places where generic arguments are used: the HST and SWI modules. As explained earlier the Host notifier expects functions to be of the form *Fxn(Arg, Arg*). But, *SWI_andn(SWI_Handle swiPtr, Uns key)* is used to post the software interrupt *copySwi*. As mentioned in previous sections, on the C54x this does not cause any problems. On the C55x, the generic arguments are treated as 16-bit pointer quantities and are placed into locations different from those expected by the called function.

On the C55x, the HST pipe object's notifier is invoked by placing the two Arg type parameters in (X)AR0 and (X)AR1 registers per the compiler conventions. SWI_andn() is expecting the second argument to be Uns, and the compiler convention dictates that the second parameter should be in T0.

There are two possible solutions. One solution is to use stub functions that conform to the expected interface and call the DSP/BIOS API. The other solution is to use a new API that is designed for use with generic arguments.

A.3.5.1 Stub Function

A stub function can be called as the HST notify function. It should conform to the *Fxn(Arg, Arg)* prototype and it should, in turn, post the *copySwi*. If necessary, type conversion can be performed. Type conversion functions provided with DSP/BIOS can be found in Table 5.

The example application already utilizes stub functions and implements notify functions by calling the inputReady(Void) and outputReady(Void) functions. The source code for these functions can be seen in Figure A–7 and Figure A–8.

```
Void inputReady(Void)
{
     SWI_andn(&copySwi, 1); /* clear swi mbx bit position 0 */
}
```

Figure A–7. inputReady Stub Function

```
Void outputReady(Void)
{
     SWI_andn(&copySwi, 2); /* clear swi mbx bit position 1 */
}
```

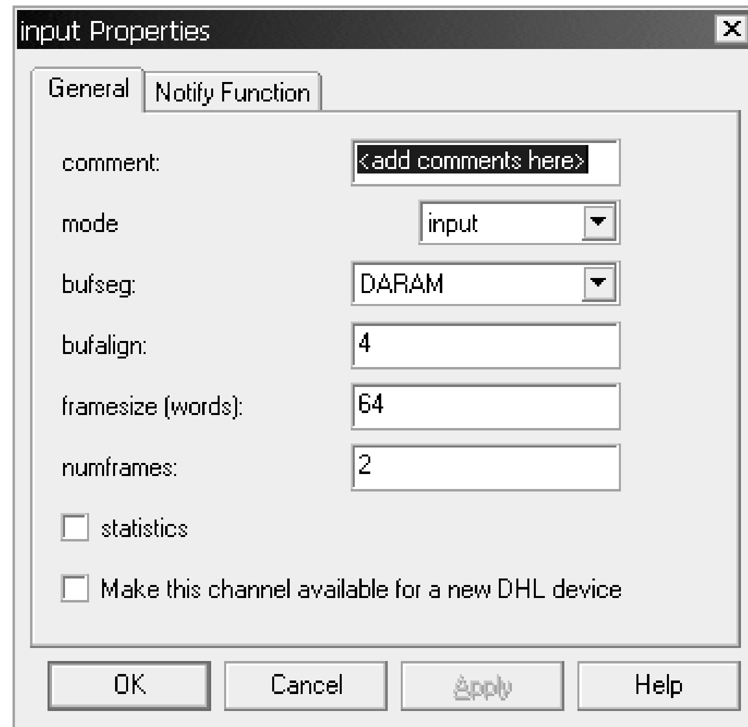
Figure A–8. outputReady Stub Function

A.3.5.2 Alternate API

An alternate approach is to use variants of function APIs that compensate for the difference in parameter placement.

DSP/BIOS provides special variations of 2 frequently used SWI APIs, as listed in Table 6. These 2 new DSP/BIOS APIs are equivalent in functionality to SWI_andn and SWI_or.

In the "Copy" example, it is possible to use SWI_andnHook() instead of SWI_andn() as the HST pipe notifier function. No modifications are necessary to the SWI object. The HST input object and HST output object are listed in Figure A–9 and Figure A–10.



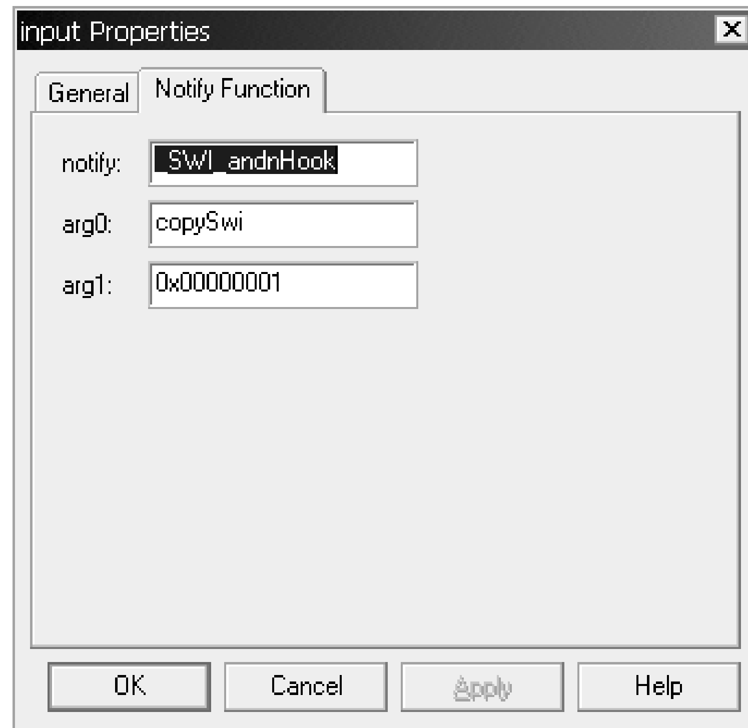
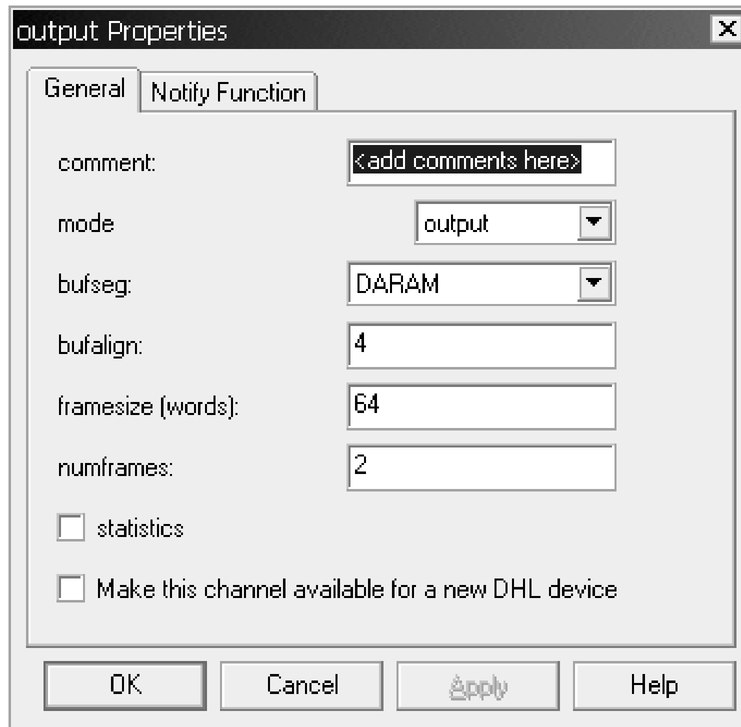


Figure A–9. C55x HST Input Object



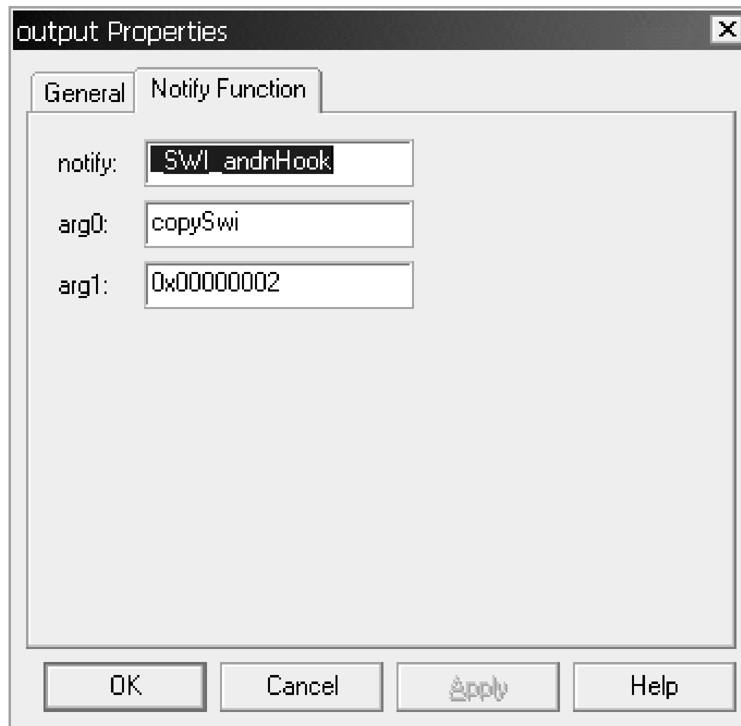


Figure A–10. C55x HST Output Object

A.4 Code Issues

After saving the DSP/BIOS CDB file, add the copy.cdb, and copycfg.cmd files to the project.

Table A–1 is a checklist of things that could be checked before building the program. This is only a guide.

Table A–1. Pre-build Checklist

Item	Note
Stack	Check to see if migrated code is compatible with selected stack mode.
Memory Model	Make sure source code is compatible with selected memory model. If in small memory model, check to ensure all data is contained inside a single 64k page.
Vector Table	Check to see if proper user interrupts are installed in DSP/BIOS HWI manager.
C interface to ASM code	Make sure the proper #pragma statements are used when calling assembly code from C. (Refer to the *TMS320C55x Optimizing C Compiler User's Guide*, literature number SPRU281, for details.)
Assembly code	Ensure that all C54x assembly code has been properly migrated to the C55x and assembles correctly. Separate functional verification eases project migration.
Generic Arguments	Check all function prototypes in user code that may rely on the function parameter passing model of the C54x C compiler. Also check for function prototype mismatches for all DSP/BIOS modules that utilize generic arguments, as listed in .
DSP/BIOS configuration	Make sure the DSP/BIOS configuration selected matches the intended target.
RTDX mode	Ensure that the operating mode of RTDX matches the target and the capabilities of the development platform.

After running through the checklist, re-build the project.

For large projects, configuration management is highly advisable. Migration of a large applications should be functionally incremental. Begin with the smallest subset of functionality and after verification incrementally add functions. Coupled to a successful configuration management strategy, this should enable successful problem resolution.

A.5 Conclusion

As seen in this example, migration of a DSP/BIOS C54x application to DSP/BIOS is possible with a minimum of effort. The example has also highlighted the similarities between the C54x and C55x DSP/BIOS API. This illustrates the ability to create a level of code common between the two platforms.

Appendix B
Migrating a C54x Code Composer Studio v1.2 DSP/BIOS Project

Code Composer Studio v2.0 features DSP/BIOS with some added functionality. For example, the Chip Support Library has been integrated with DSP/BIOS in Code Composer Studio 2. This section will discuss some additional issues specific to the migration of C54x projects from Code Composer Studio 1.2 to Code Composer Studio 2 on the C55x device.

It is highly recommend that a migration directly from Code Composer Studio 1.2 for the C54x be attempted in stages. The first stage should consist of migrating the Code Composer Studio project to Code Composer Studio 2. The second stage should then follow the procedures outlined in this application note for migrating the application from the C54x to C55x.

As mentioned in the introductory paragraph, there is a large amount of applications material available from Texas Instruments to expedite this process. Please refer to dspvillage.ti.com for more information.

B.1 Stage 1: Migration from Code Composer Studio v1.2 to Code Composer Studio v2.0

Migration from Code Composer Studio 1.2 to Code Composer Studio 2 requires examination of the items illustrated in Table B–1. After this stage, success should result in a successful program build of the C54x code.

Table B–1. Migration Items

Item	Notes
Project and build environment	Existing project files must be migrated to Code Composer Studio 2. Care must be taken to ensure the build environment (defines, compiler options, etc.) are preserved.
Code Generation	Code Composer Studio 2 contains a newer version of the code generation tools. Successful compilation of existing program files is essential to successful project migration.
DSP/BIOS	Code Composer Studio 2 contains a newer version of DSP/BIOS. Migrate the CDB file by opening the CDB file in the configuration tool. Menus will appear to guide the migration process. Once migration is complete, saving of the updated CDB is essential to cause the creation of new DSP/BIOS generated files (.s54, .cmd, etc.) Without the new generated files, a program re-build will fail.

B.2 Stage 2: Migration from C54x DSP/BIOS to C55x DSP/BIOS

After a successful re-build of the C54x code using the Code Composer Studio 2 components (code generation and DSP/BIOS libraries), the migration process from the C54x to the C55x can begin. A suitable process is outlined in Appendix A contained in this application report.

IMPORTANT NOTICE

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

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

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

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

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

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2001, Texas Instruments Incorporated