Application Report
SLAA376–November 2007

Extended Memory Access Using IAR v3.42A and CCE v2

Bhargavi Nisarga *MSP430 Applications*

ABSTRACT

This application report describes how to access the extended memory in devices with memory greater than 64 KB using the large code/small data model supported by the IAR C/C++ compiler v3.42A and Code Composer Essentials (CCE) v2. The purpose of this application report is to provide an understanding of the extended features of the MSP430X CPU, customizing the linker command file to fit the target system memory map, mixing C and assembly functions to assign data in the extended memory, and using intrinsic functions/user-defined assembly modules to access data in extended memory space. Also, brief descriptions of different methods to place and access data in the extended memory of such devices are covered. The IAR C/C++ compiler v4.0 does support a large code/large data model and is now available online.

Contents

List of Figures

1 Introduction

Recently, many devices with an extended MSP430X 16-bit RISC CPU with memory capacity greater than 64 KB have been released. The extended MSP430X instruction set gives the MSP430X CPU full access to its 20-bit address space. However, the IAR C/C++ compiler v3.42A and Code Composer Essentials (CCE) v2 employ a large code/small data model that does not support data storage in the extended memory. Interfacing C with assembly modules helps place data in extended memory. Compiler predefined intrinsic functions and user-defined assembly functions help access the data placed in the extended memory. Direct Memory Access (DMA) or an address variable can be used to efficiently access and move data in upper memory to different locations like the DAC registers, RAM, and so on. Also, continuous samples of data can be written into flash directly via the DMA or an address variable.

2 Memory Organization of MSP430X Devices

Devices with greater than 64 KB of flash integrate an extended MSP430X 16-bit RISC CPU with their memory map as shown in Figure 1. The lower 64 KB can be accessed with a 16-bit address. More address bits are required to access the extended upper memory. The memory range of the MSP430X devices vary with the device parameters. 20 address bits are available to access the upper extended memory. The interrupt vector is located at the upper end of the lower 64-KB memory, and it separates the upper memory from the lower memory.

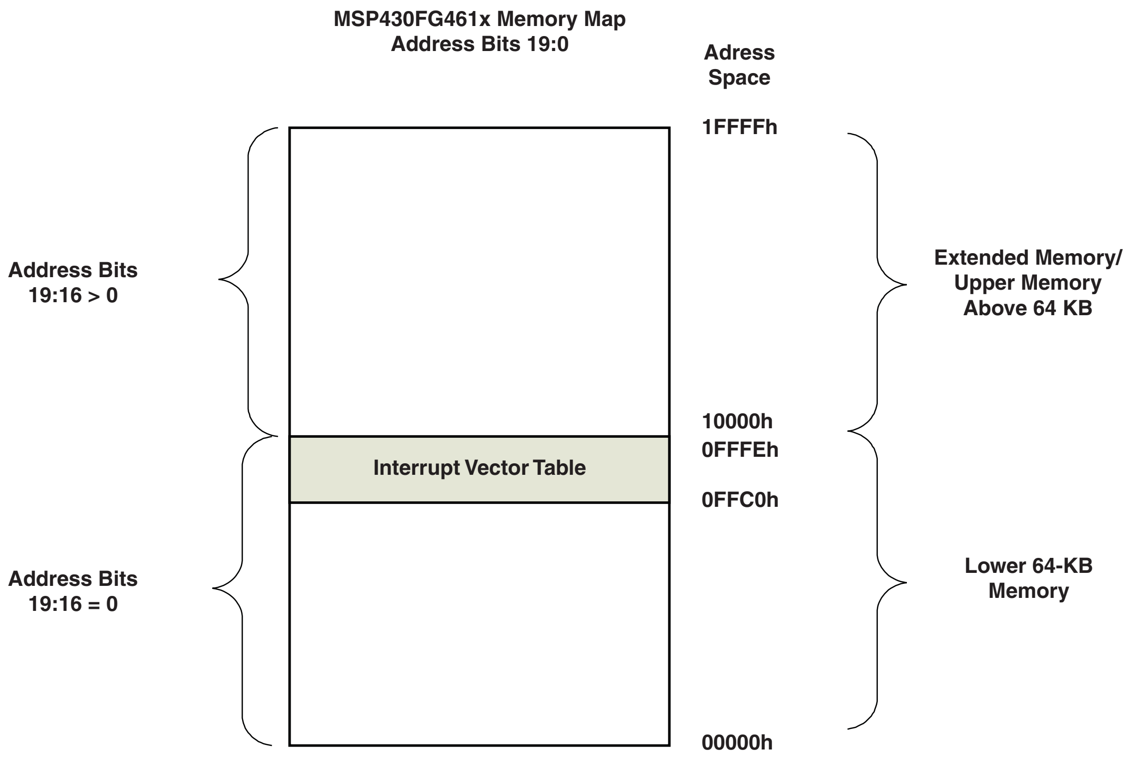

Figure 1. Memory Map of MSP430X Device

3 Extended MSP430X 16-Bit RISC CPU

The MSP430X CPU can address up to 1-MB address range without paging. In addition to the MSP430 CPU features, the MSP430X CPU includes:

- 20-bit address bus allowing direct access and branching throughout the entire memory range without paging
- Byte, word, and 20-bit address-word addressing

3.1 MSP430X CPU Registers

The block diagram of the MSP430X CPU is shown in Figure 2.

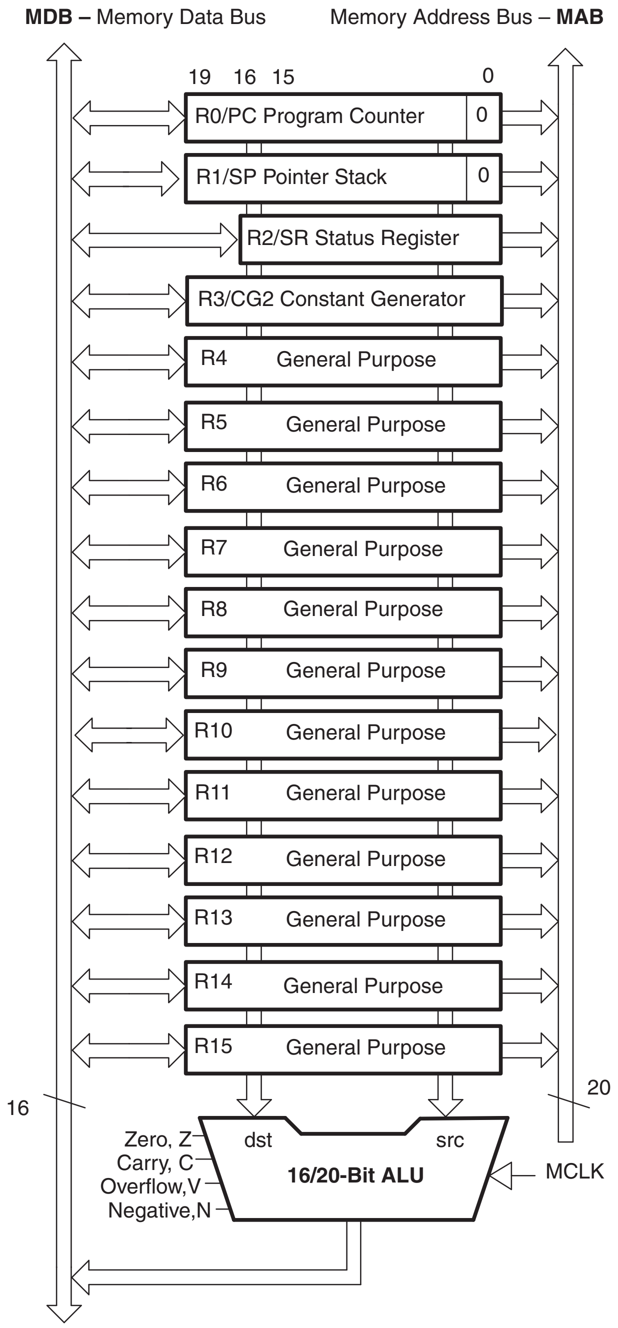

Figure 2. MSP430X CPU Block Diagram

It can be seen that the CPUX has a 20-bit memory address bus, 20-bit R0/PC (program counter), R1/SP (stack pointer), R3/CG (constant generator), and R4 to R15 general-purpose registers, as well as a 16/20-bit arithmetic logic unit (ALU).

MSP430X Extended Instructions: The extended MSP430X instruction set gives the MSP430X CPU full access to its 20-bit address space. MSP430X extended instructions require an additional word of op-code called the extension word. All addresses, indices, and immediate numbers have 20-bit values when preceded by the extension word. The MSP430 instructions are used throughout the memory range unless their 16-bit capability is exceeded. The MSP430X instructions are used when the addressing of the operands or the data length exceeds the 16-bit capability of the MSP430 instructions. Therefore, the MSP430X CPU is backwards compatible with the MSP430 CPU. [3]

Note: The additional extension-word op-code is not required in the MSP430X address instructions that support 20-bit operands (ADDA, CMPA, and SUBA). The addressing modes are restricted to the register mode and the immediate mode. Restricting the addressing modes removes the need for the additional extension-word op-code, thereby improving code density and execution time. Hence, address instructions should be used whenever an MSP430X instruction is needed with the corresponding restricted addressing mode. [3]

3.1.1 Program Counter

MSP430X CPU has a 20-bit program counter (PC/R0) that points to the next instruction to be executed.

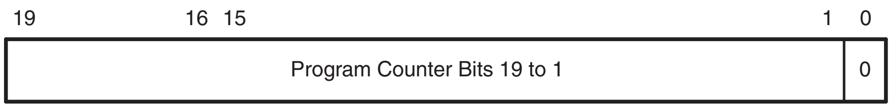

Figure 3. MSP430X Program Counter

Addresses in the lower 64-KB address range can be reached with the BR or CALL instruction. The BR and CALL instructions reset the upper four PC bits to 0. The BRA and CALLA instructions reach addresses beyond lower 64-KB range when branching or calling.

The program counter is automatically stored onto the stack with CALL or CALLA instruction. During an interrupt service routine, a CALL instruction stores only bits 15:0 of the PC onto the stack. However, during the execution of a CALLA instruction, bits 19:0 of the PC are stored in the stack. Figure 4 shows the storage of the PC contents with return address after a CALLA instruction. The RET instruction restores bits 15:0 to the program counter and adds two to the stack pointer. The RETI instruction restores bits 19:0 of the program counter and adds four to the stack pointer.

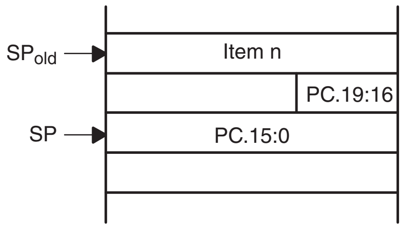

Figure 4. PC Contents Storage on Stack Upon Execution of CALLA

3.1.2 Stack Pointer

The 20-bit stack pointer (SP/R1) is used by the CPU to store the return addresses of subroutine calls and interrupts. The instruction PUSHX.A pushes 20-bit address word onto the stack as shown in Figure 5. During the execution of POPX.A instruction, the entire 20-bit address is restored to the destination from the stack.

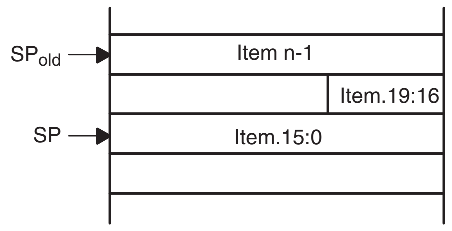

Figure 5. 20-Bit Address Storage on Stack Upon Execution of PUSHX.A

3.1.3 Working Registers R4 to R15

The twelve CPU working registers, R4 to R15, can contain 8-bit, 16-bit, or 20-bit values. Any byte written to a CPU register clears bits 19:8. Similarly, any word written to a register clears bits 19:16. The instructions with the ".A" suffix provide the 20-bit address-word handling. (Exception: SXT instruction extends the sign through the complete 20-bit register.)

3.2 Interrupt Handling by MSP430X CPU

The MSP430X uses the same interrupt structure as the MSP430:

- Vectored interrupts with no polling necessary
- Interrupt vectors are located in the interrupt vector table from addresses FFC0h to FFFEh.

Even in the MSP430X devices, all interrupt handlers must start in the lower 64-KB memory, since the interrupt vectors contain 16-bit addresses that point to the lower 64-KB memory.

3.2.1 Program Counter Storage on Stack for Interrupts

The program counter and the status register are pushed onto the stack during an interrupt. The MSP430X architecture stores the complete 20-bit PC value on the stack by automatically appending the PC bits 19:16 to the stored SR value as shown in Figure 6. During the execution of the RETI instruction the full 20-bit value of the PC is restored, making the return of control from an interrupt to any address in the memory range.

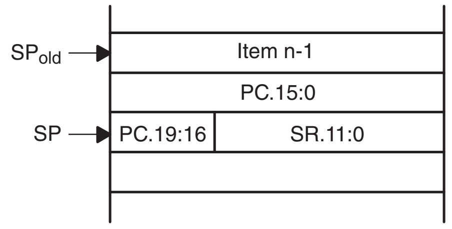

Figure 6. PC Contents Storage on Stack During an Interrupt

4 Accessing the 20-Bit Address Space

4.1 *IAR/CCE Assemblers and IAR/CCE C/C++ Compilers*

Both the IAR and CCE assemblers support the extended MSP430X instructions to provide complete access to the 20-bit address space of the MSP430X device. However, neither the IAR C/C++ compiler v3.42A nor the CCE v2 support extended memory access when programming in C alone. These versions of the compiler support a large code/small data model; i.e., the compiler supports large codes that can be placed in the extended memory but not large data. For applications that need to access the upper memory in the MSP430X devices, there arises a need to write parts of the code in assembly.

4.2 *Mixing C and Assembly Languages*

The combination of C and assembly benefits the programmer by providing the power of a high-level language as well as the speed, efficiency, and low-level control of assembly.

4.2.1 IAR v3.42A

The MSP430 IAR C/C++ compiler provides several ways to mix C and assembly languages, such as modules written entirely in assembly, intrinsic functions, and inline assembly.

4.2.1.1 *Intrinsic Functions*

Intrinsic functions are predefined functions that allow the compiler to have direct access to low-level processor operations without having to use assembly language. The advantage of an intrinsic function compared to using inline assembly is that the compiler has all necessary information to interface the sequence properly with register allocation and variables. Also, the compiler optimizes functions with such sequences, something the compiler is unable to do with inline assembly sequences. An intrinsic function is a built-in function that compiles into inline code, either as a single instruction or as a short sequence of instructions. The intrinsic functions that are used to access the upper 64-KB memory are:

```
/* Compiler Pre-defined Intrinsic functions in IAR C/C++, V3.42A */
/* The following functions are available in extended mode (--core=430X) to
 * access the upper memory above 64 KB as a data area. */

__intrinsic void __data20_write_char (unsigned long  __addr,
                                      unsigned char  __value);

__intrinsic void __data20_write_short(unsigned long  __addr,
                                      unsigned short __value);

__intrinsic void __data20_write_long (unsigned long  __addr,
                                      unsigned long  __value);

__intrinsic unsigned char  __data20_read_char (unsigned long __addr);
__intrinsic unsigned short __data20_read_short(unsigned long __addr);
__intrinsic unsigned long  __data20_read_long (unsigned long __addr);

/* The following two functions can be used to access 20-bit SFRs in the lower
 * 64kB. They are only available in extended mode (--core=430X). */

__intrinsic void __data16_write_addr (unsigned short __addr,
                                      unsigned long  __value);

__intrinsic unsigned long  __data16_read_addr (unsigned short __addr);
```

Note: To use intrinsic functions, the header file intrinsics.h should be included in the program.

4.2.1.2 Mixing C and Assembly Modules in IAR

It is possible to write parts of an application in assembly and mix them with the C modules. The calling conventions decide which registers are used to pass function parameters between C and assembly and how the return value is passed back to the caller. In IAR, an assembly routine that is to be called from C must:

- Conform to the calling convention
- Have a PUBLIC entry-point label
- Be declared as external before any call to allow type checking and optional promotion of parameters

4.2.2 Code Composer Essentials (CCE)

These are ways to use assembly language in conjunction with C code in CCE:

- Use separate modules of assembled code and link them with compiled C modules
- Use assembly language variables and constants in C source
- Use inline assembly language embedded directly in C source

Interfacing C with assembly language functions is straightforward if register conventions and calling conventions are followed. More information on these topics is presented in the *MSP430 Optimizing C/C++ Compiler v 2.0.1 User's Guide* (SLAU132). [6]

4.2.2.1 Mixing C and Assembly Modules in CCE

Unlike the IAR C/C++ compiler v3.42A, the CCE C/C++ compiler v2 does not support any compiler-predefined intrinsic functions to access data in the extended memory or access 20-bit SFRs in the lower 64 KB. So, in this application report separate assembly functions are used in CCE C programming to emulate the IAR compiler-predefined intrinsics. CCE C programs call these CCE assembly functions in the same way the predefined intrinsic functions are called by an IAR C program.

Any assembly routine that interfaces with an MSP430X C program is required to conform to the large code model:

- Use CALLA/RETA instead of CALL/RET
- Use PUSHM.A/POPM.A to save and restore any used save-on-entry registers. The entire 20-bit register must be stored/restored.
- Manipulation of function pointers requires 20-bit operations ([operand].A)

In CCE, the assembly functions that are to be called from the C program should be defined as ".global" in the ASM file. ".global" is a file reference directive that identifies one or more symbols that are defined in the current module and used in other modules. Also, these assembly functions should be declared as external in the C code using the EXTERN directive. EXTERN is a file reference directive that identifies one or more symbols used in the current module but defined in another module.

4.2.2.2 User-Defined Assembly Functions in CCE

The user-defined assembly functions used in the code examples of this application report that help access the upper memory above 64 KB are:

1. *unsigned char __data20_read_char(unsigned long __addr)* – This function reads an unsigned char present in the extended memory.

   Passing parameter:

   - *unsigned long __addr* – The 20-bit address that is passed is stored in a long integer as two words, the lower word in R12 and the upper word in R13, according to the CCE C compiler register and calling convention.

   Return parameter:

   - *unsigned char* – The 8-bit char value present at the 20-bit address location (passed during function call) is the return value. This return value is placed in the R12 register.

```
                     .global __data20_read_char
                     .text
__data20_read_char:
                     push.w   R13                 ;upper word
                     push.w   R12                 ;lower word
                     popx.a   R13                 ;20-bit address
                     ;Move 8-bit char value at the 20-bit addr locn to R12
                     movx.b  0x0(R13), R12        ;R12 contains the return value
                     reta                         ; return
```

2. *unsigned short __data20_read_short(unsigned long __addr)* – This function reads an unsigned short value from the extended memory.

   Passing parameter:

   - *unsigned long __addr* – The 20-bit address that is passed is stored in a long integer as two words, the lower word in R12 and the upper word in R13, according to the CCE C compiler register and calling convention.

   Return parameter:

   - *unsigned short* – The 16-bit short value present at the 20-bit address location (passed during function call) is the return value. This return value is placed in the register R12.

```
                     .global __data20_read_short
                     .text
__data20_read_short:
                     push.w  R13                 ;upper word
                     push.w  R12                 ;lower word
                     popx.a  R13                 ;20-bit address
                     ;Move 16-bit short value present at the 20-bit address
                     ;stored in R13 to R12
                     movx.w  @(R13), R12         ;R12 contains the return value
                     reta                        ; return
```

3. *void __data20_write_char(unsigned long __addr, unsigned char __value)* – This function is used to write an unsigned char value into the extended memory.

   Passing parameters:

   - *unsigned long __addr* – The 20-bit address to which an 8-bit char value is to be written is stored in long integer as two words, the lower word in R12 and the upper word in R13.
   - *unsigned char __value* – The 8-bit char value, passed in R14, is the data that is to be written into the 20-bit address location.

   Return parameter: void

```
                     .global __data20_write_char
                     .text
__data20_write_char:
                     push.w  R13                ;upper word
                     push.w  R12                ;lower word
                     popx.a  R13                ;20-bit address
                     ;Write the 8-bit char value in R14 onto the 20-bit
                     ;address stored in R13
                     movx.b  R14,0x0(R13)
                     reta                       ; return
```

4. *void __data20_write_short(unsigned long __addr, unsigned short __value)* – This function is used to write an unsigned short value into the extended memory.

   Passing parameters:

   - *unsigned long __addr* – The 20-bit address to which a 16-bit short value is to be written is stored in a long integer as two words, the lower word in R12 and the upper word in R13.
   - *unsigned short __value* – The 16-bit short integer, passed in R14, that is to be written into the 20-bit address.

   Return parameter: void

```
                     .global __data20_write_short
                     .text
__data20_write_short:
                     push.w   R13              ;upper word
                     push.w   R12              ;lower word
                     popx.a   R13              ;20-bit address
                     ;Write the 16-bit short value in R14 onto the 20-bit
                     ;address stored in R13
                     movx.w   R14,0x0(R13)
                     reta                      ; return
```

The following two user-defined assembly functions are used to access the 20-bit SFRs in lower 64 KB.

1. *void __data16_write_addr(unsigned short __addr, unsigned long __value)* – This function is used to write a 20-bit value into a special function register (SFR) in the lower 64-KB memory, which has a 16-bit address.

   Passing parameters:

   - *unsigned short __addr* – The 16-bit address passed in register R12, to which a 20-bit value is to be written.
   - *unsigned long __value* – This long integer contains the 20-bit value in two words, the lower word in R13 and the upper word in R14.

   Return parameter: void

```
                     .global __data16_write_addr
                     .text
__data16_write_addr:
                     push.w  R14                 ;upper word
                     push.w  R13                 ;lower word
                     popx.a  R14                 ;20-bit address
                     ;Move the 20-bit value in R14 to a 20-bit SFR in the lower 64 KB
                     ;memory location whose 16-bit address is in R12
                     movx.a  R14, 0x0(R12)
                     reta                        ; return
```

2. *unsigned long __data16_read_addr(unsigned short __addr)* – This function is used to read a 20-bit value present in the 20-bit SFR with a 16-bit address.

   Passing parameter:

   - *unsigned short __addr* – The 16-bit address location of the 20-bit SFR stored in register R12.

   Return parameter:

   - *unsigned long __value* – The 20-bit value present at the 16-bit address location is returned as a long integer contained in two words, the lower word in R12 and the upper word in R13.

```
                     .global __data16_read_addr
                     .text
__data16_read_addr:
                     ;R12 has the 16-bit address of a 20-bit SFR
                     pushx.a @R12               ;push 20-bit addr to stack
                     ;pop as two words
                     pop.w R12                  ;lower word in R12
                     pop.w R13                  ;upper word in R13
                     reta                       ; return
```

Note: To use the above functions, the assembly file Ext_Intrinsics.asm must be included in the project build.

5 Data Initialization in Extended Memory

In assembly language programming, the programmer is responsible for declaring and naming relocatable segments/sections and determining how they are used. However, in C programming, the compiler creates and uses a set of predefined code and data segments; hence, the programmer uses a linker tool to define segments and their usage.

The MSP430 link step configures system memory by allocating output modules efficiently into the memory map using a linker command file. The link step combines object files and performs the following tasks:

- Links various modules together by resolving all global or external symbols that could not be resolved by the assembler or compiler
- Loads modules needed by the program from user-defined or IDE-supplied libraries
- Locates each segment of code or data at a user-specified address (specified within the linker command file)

5.1 IAR XLINK Linker

The XLINK linker tool performs the link step for the IAR Embedded Workbench. XLINK reads one or more relocatable object files produced by the IAR Systems assembler or compiler and produces absolute machine-code programs as output. It loads modules containing executable code or data from the input file(s).

Each module contains a number of segment parts. Each segment part belongs to a segment and contains either bytes of code or data or reserves space in RAM. Using the XLINK segment control command line options -Z, -P, and -b, the programmer can cause load addresses to be assigned to segments and segment parts. [2]

5.1.1 Customizing the IAR Linker Command File (*.xcl)

The `config` directory contains one ready-made linker command file for each MSP430 device. This linker command file contains the information required by the linker and is ready to be used. The only change made to the supplied linker command file is to customize it so it fits the target system memory map.

A segment is a logical entity containing a piece of data or code that should be mapped to a physical location in memory. The MSP430 IAR C/C++ compiler has a number of predefined segments for different purposes. Each segment has a name that describes the contents of the segment and a segment memory type that denotes the type of content. The ROM can be used for storing CONST and CODE segment memory types. The RAM memory can contain segments of DATA type. In addition to the predefined segments, users can define their own segments.

In applications where MSP430X devices are used, in order to place data in the upper memory, the placement directives are modified in the linker command file. Segments can be placed using the -Z and -P options. The former places the segment parts in the order they are found, while the latter rearranges them to make better use of the memory. The -P option is useful when the memory where the segment should be placed is not continuous. [1]

For the project in this application report, the linker command file is modified by adding the -Z placement directive to allocate the segment CONST_HIGH sequentially in the memory range 0x10000 to 0x1FFFF.

```
// File: lnk430FG4618_Custom.xcl (Linker Command file)
// Constant data.
// Modified below...
-Z(CONST)CONST_HIGH=10000-1FFFF
```

5.1.2 Data Table Placement

A data table can be placed in the extended memory using an assembly module. The programmer can declare and name relocatable segments and place data directly in these segments using assembly-level programming.

- In IAR code examples 1 to 4 of this application report, a 32-word sine lookup table (Sin_tab) is placed in the extended memory CONST_HIGH:

```
; File: ASM_func.s43
; Data Table in extended memory
                     NAME  Sin_tab
                     RSEG  CONST_HIGH
                     EVEN
Sin_tab              DW 2048, 2447, 2831, 3185, 3495, 3750, 3939, 4056
                     DW 4095, 4056, 3939, 3750, 3495, 3185, 2831, 2447
                     DW 2048, 1648, 1264,  910,  600,  345,  156,   39
                     DW    0,   39,  156,  345,  600,  910, 1264, 1648
```

5.1.3 Data Table Address Access

In IAR, the address location of the data table (Sin_tab) placed in the extended memory is accessed as follows:

- The 20-bit address of the data table, placed in the extended memory segment CONST_HIGH, is retrieved by calling the user-defined function *unsigned long __Get_Address20(void)*:

  Passing parameter: void

  Return parameter:

  – *unsigned long_value* – The 32-bit unsigned long return value is returned in two words, the lower word in R12 and the upper word in R13.

```
; File: ASM_func.s43
            PUBLIC   __Get_Address20
            RSEG CODE
__Get_Address20
            mov.w #LWRD Sin_tab, R12        ; Lower word - R12
            mov.w #HWRD Sin_tab, R13        ; Upper word - R13
            reta                            ; Return
```

- The function *__Get_Address20* is defined as PUBLIC in the ASM file. PUBLIC is a file reference directive that identifies one or more symbols defined in the current module and used in other modules.
- The function *__Get_Address20* is declared as external in the C file using the EXTERN directive.

5.2 CCE Linker

The CCE link step command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. Defining and creating a memory model aids the user to configure system memory. In CCE, the two directives, MEMORY and SECTIONS, allow the user to:

- Allocate sections into specific areas of memory
- Combine object file sections
- Define or redefine global symbols at link time

The MEMORY directive defines the target memory configuration and the SECTIONS directive controls how sections are built and allocated.

5.2.1 Customizing the CCE Linker Command File (*.cmd)

Linker command files in CCE allow the user to use the MEMORY and SECTIONS directives to customize an application. Detailed explanations of the linker command files and the directives are presented in the MSP430 Assembly Language Tools v 2.0.1 User's Guide (SLAU131). [5] These directives should be used in a command file and cannot be used as a command line.

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. The predefined MEMORY directive for a target device with extended memory contains a memory section defined for the upper memory above 64 KB. For example, the extended memory for the target device (MSP430FG4618) used in this application report is declared by default in the MEMORY directive as shown:

```
    FLASH2                  : origin = 0x10000, length = 0x10000
```

Memory allocation is performed according to the rules specified by the SECTIONS directive. For the project in this application report, the linker command file is modified by adding the SECTIONS directive definition to create a new section ".Chigh" in the extended memory range (0x10000 to 0x1FFFF), specified previously by the MEMORY derivative as FLASH2.

```
/* Modified Below....*/
.Chigh      : {} > FLASH2              /*CONSTANT DATA IN EXTENDED FLASH*/
```

Note: The link step allocates output sections from low to high addresses within a designated memory range by default. Alternatively, the user can cause the link step to allocate a section from high to low addresses within a memory range by using the HIGH location specifier in the SECTION directive declaration. For example, the above SECTIONS derivative can be declared with HIGH location specifier to allocate the section from a high to low address in the memory range specified.

```
.Chigh   : {} > FLASH2(HIGH)        /*CONSTANT DATA IN EXTENDED FLASH*/
```

5.2.2 Data Table Placement

A data table can be placed in the extended memory using an assembly module. The programmer defines the target memory configuration and controls allocation of different memory sections built using assembly level programming.

- In CCE code examples 1 to 3 of this application report, a 32-word sine lookup table (Sin_tab) is placed in the extended memory section ".Chigh" as shown:

```
            ; Data Table in extended memory
            .sect ".Chigh"
            .align 2
Sin_tab     .word 2048, 2447, 2831, 3185, 3495, 3750, 3939, 4056
            .word 4095, 4056, 3939, 3750, 3495, 3185, 2831, 2447
            .word 2048, 1648, 1264,  910,  600,  345,  156,   39
            .word    0,   39,  156,  345,  600,  910, 1264, 1648
```

5.2.3 Data Table Address Access

In CCE, the address location of the data table (Sin_tab) placed in the extended memory is accessed as follows:

- The 20-bit address of the data table, placed in the extended memory section, .Chigh, is retrieved by calling the user-defined function *unsigned long __Get_Address20(void)*:

  Passing parameter: void

  Return parameter:

  – *unsigned long_value* – The 32-bit unsigned long return value is returned in two words, the lower word in R12 and the upper word in R13.

```
; File: ASM_func.asm
           .global __Get_Address20
           .text
__Get_Address20
            movx.a  #Sin_tab, R12           ;R12 now has the 20-bit address
            ;Split into two 16-bit words to return data to C
            andx.a  #0x00FFFF, R12
            movx.a  #Sin_tab, R13
            ;Rotate 20-bit addr in R13, 16 times to its right arithmetically
            rpt      #16 ||
            rrax.a   R13
            reta                            ; Return
```

- The function *__Get_Address20* is defined as .global in the ASM file. .global is a file reference directive that identifies one or more symbols that are defined in the current module and used in other modules.
- The function *__Get_Address20* is declared as external in the C file using the EXTERN directive.

6 Code Example Description

This section briefly describes the associated code examples with this application report.

6.1 Before Running Code Examples

6.1.1 IAR Code Examples

For All IAR Code Examples

- In order to use intrinsic functions, intrinsics.h header file should be included in the code.

For IAR Code Examples CPUx_01 to CPUx_04

- The linker command file for this project has been modified by adding *-Z(CONST) CONST_HIGH = 10000–1FFFF* to create a new memory segment CONST_HIGH in the extended memory (0x10000 to 0x1FFFF).
- Do not change the original linker command file. It is recommended that a copy in the working directory is made and that the copy is modified instead.
- Link the modified linker command file (.xcl) to the active project.
  – Click Project/Options/Linker/Config/Override default.
  – Use the browse button to open the modified linker command file and click OK.
- The project builds must include the assembly file ASM_func.s43.

6.1.2 CCE Code Examples

For All CCE Code Examples

- The project builds must include the assembly file Ext_Intrinsics.asm, to define functions that access the upper extended memory and/or 20-bit SFRs in the lower 64-KB memory.

For CCE Code Examples CPUx_01 to CPUx_03

- The linker command file (*.cmd) for this project must be modified to create a new memory segment (.Chigh, in this project).
- The linker step command .Chigh : {} > FLASH2 is added to the SECTIONS directive in the linker command file to create a new MEMORY section in the extended memory. ".Chigh" ranges from address 10000h to 1FFFFh, in which constant data can be stored.
- Do not change the original linker command file. It is recommended that a copy in the working directory is made and that the copy is modified instead.
- Link the modified linker command file (.cmd) to the active project.
  – Create a New Managed C/ASM Project. Use the default linker command file for the selected device.
  – Add the modified linker command file to the project's working directory.
  – Delete the default linker command file. Note that this does not delete the original linker command file present in the path C:\Program Files\Texas Instruments\CC Essentials 2.0\tools\compiler\msp430\include. It deletes only the copy of the linker command file that was written into the project's working directory at project creation.
  – The modified linker command file is automatically linked to the project.

Note: Building the project with two linker command files results in memory overlap errors.

- The project builds must include the assembly file ASM_func.asm.

6.2 Code Example Description

6.2.1 IAR/CCE Example CPUx_01

This program demonstrates how a data lookup table placed in the extended memory is transferred word by word to the DAC by the DMA and is output as a 1-kHz sine wave.

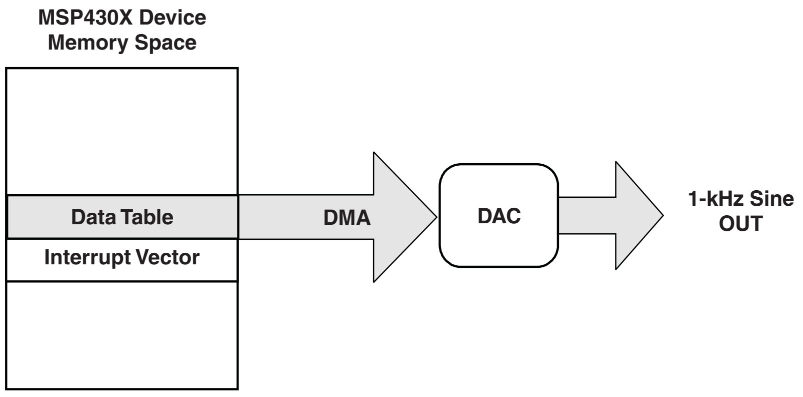

Figure 7. CPUx_01: Program Flow

6.2.2 IAR/CCE Example CPUx_02

This program demonstrates how a data lookup table placed in extended 20-bit address location is accessed via a 32-bit address variable. The contents of the data table are transferred word by word to the DAC and output as a 1-kHz sine wave.

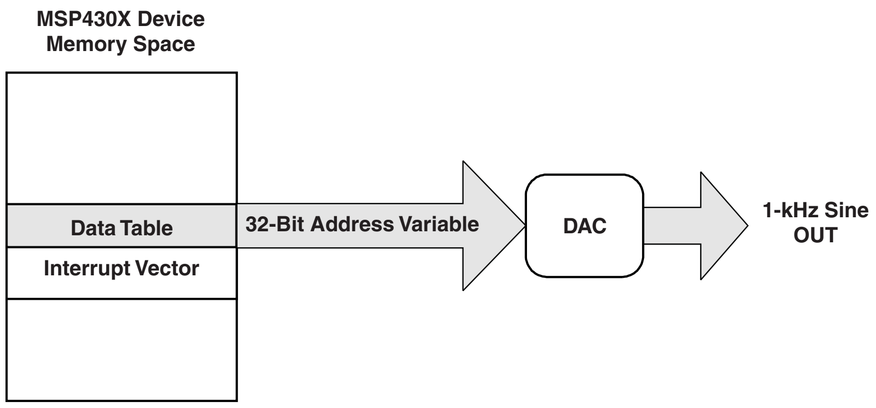

Figure 8. CPUx_02: Program Flow

6.2.3 IAR/CCE Example CPUx_03

This program demonstrates how a data lookup table placed in the extended memory is transferred as a single block to the RAM using DMA and moved word by word from RAM to DAC via a 16-bit address variable.

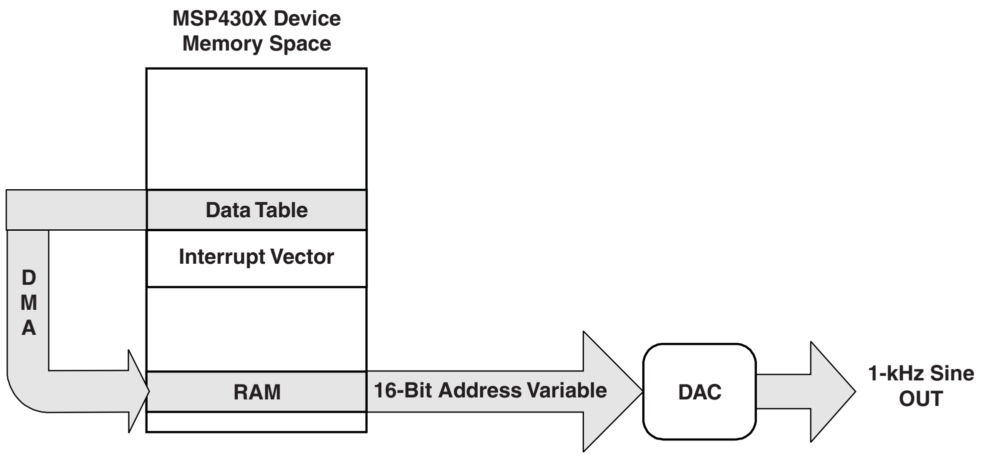

Figure 9. CPUx_03: Program Flow

6.2.4 IAR Example CPUx_04

This program demonstrates how register R4 can be used to store a 20-bit address variable to access the data lookup table placed in the extended memory. The contents of the data table are transferred word by word to the DAC and output as a 1-kHz sine wave.

Note: Register R4 is used exclusively as a 20-bit address variable in this code example and should be locked so that the compiler does not use this register for its operations. Therefore, the compiler command line option --lock_r4 must be enabled. This makes the module linkable with both modules that use R4 as __regvar (which reserves register R4 for use by global register variables) and modules that do not define its R4 usage. [1]

Enable the compiler command line option --lock_r4 by clicking Project>Options>C/C++ Compiler>Code>R4 utilization>Not used.

The R4 register lock feature is not present in CCE v2, so this code example cannot be built using CCE.

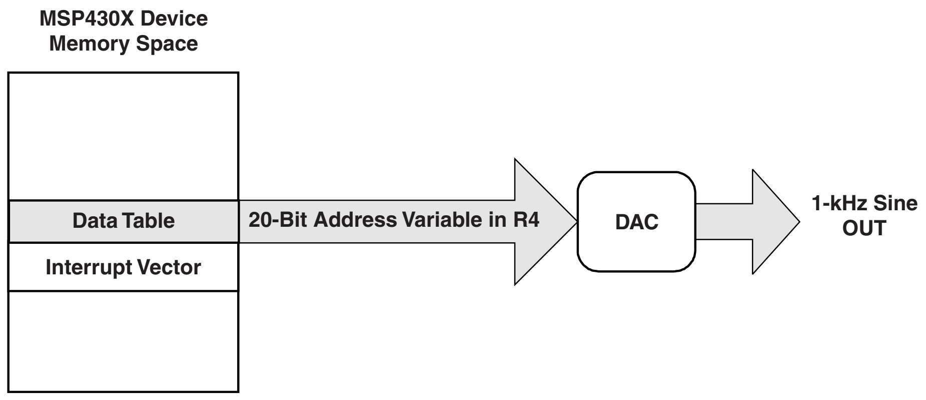

Figure 10. CPUx_04: Program Flow

User-defined assembly modules are used in this code to write to and read from the locked register R4:

- *void __Write_R4(unsigned long value)* – This function is used to write a 20-bit address in register R4.

  Passing parameter: The 32-bit unsigned long value is passed to the function in two words, the lower word in R12 and the upper word in R13.

  Return parameter: void
- *unsigned int __Read_at_R4_plus(void)* – This function is used to read the data located at the 20-bit address passed in register R4, and then word-increment the contents of R4.

```
; File: ASM_func.s43
            PUBLIC  __Write_R4
            PUBLIC  __Read_at_R4_plus

     RSEG CODE
__Write_R4
            ; Write a 20-bit address in register R4
            push.w  R13                    ; Push two words into stack
            push.w  R12                    ; and pop a 20-bit address
            popx.a  R4                     ; R4 has the 20-bit address
            reta

__Read_at_R4_plus
            ; Read the data located at the 20-bit address stored in register R4
            ; and word increment R4
            mov.w  @R4+, R12               ; R4 acts as address variable
            reta
```

6.2.5 IAR/CCE CPUx_05

This program demonstrates how ADC12 conversion results are written directly to the extended flash via the DMA.

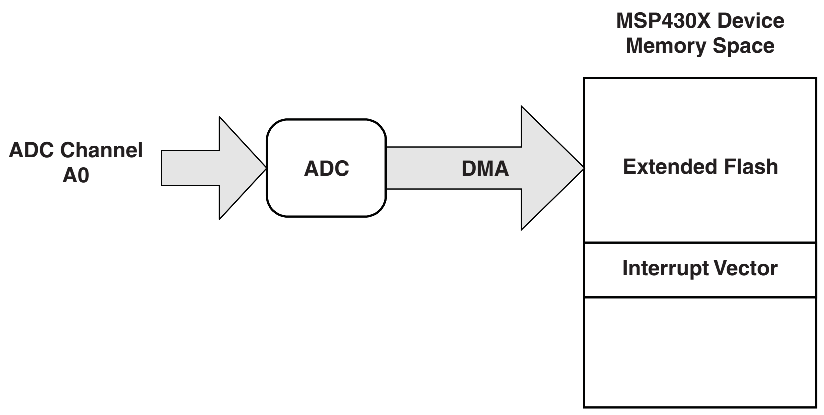

Figure 11. CPUx_05: Program Flow

6.2.6 IAR/CCE Example CPUx_06

This program demonstrates how ADC12 conversion results are written directly to the extended flash via a 32-bit address variable.

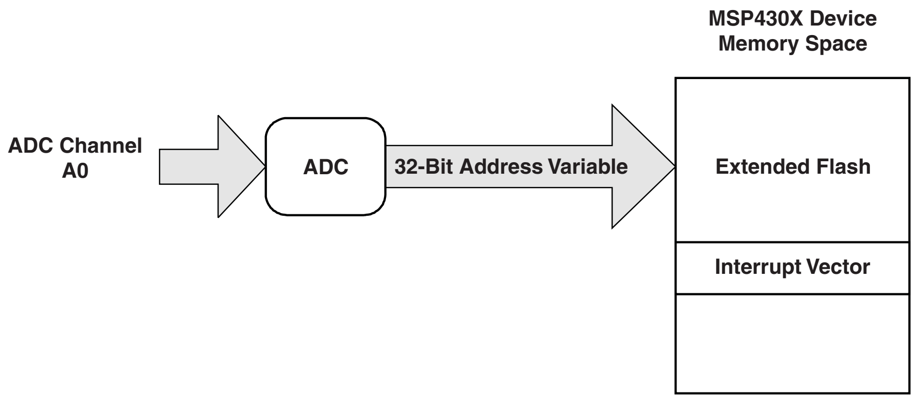

Figure 12. CPUx_06: Program Flow

7 References

1. *IAR MSP430 C/C++ Compiler Reference Guide* (http://www.iar.com/)
2. *IAR MSP430 Linker and Library Tools Reference Guide* (http://www.iar.com/)
3. *MSP430x4xx Family User's Guide* (SLAU056)
4. *MSP-FET430 Flash Emulation Tool (FET) User's Guide* (SLAU138 and SLAU157)
5. *MSP430 Assembly Language Tools v 2.0.1 User's Guide* (SLAU131)
6. *MSP430 Optimizing C/C++ Compiler v 2.0.1 User's Guide* (SLAU132)

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless