CPE 323
MODULE 05
C Programming Language and MSP430 Assembly

Aleksandar Milenković

Email: milenka@uah.edu

Web: http://www.ece.uah.edu/~milenka

Overview

This module reviews C/C++ language for embedded systems and its relationship to MSP430 assembly language. Specifically, you will learn about the design flows for embedded software using C programming language, data types, storage classes, and storage modifiers, and how data is allocated and handled.

Objectives

Upon completion of this module learners will be able to:

- *Apply a proper design flow for embedded software using C/C++ programming languages*
- *Read and comprehend functionality of programs written in C/C++ for embedded systems*
- *Design and write programs in C/C++ to solve specific tasks*

Contents

1 Design Flow Using C/C++ Source Files

In this module we will discuss how high-level language constructs, specifically those found in C programming language, are translated into low-level MSP430 machine instructions. Embedded software developers typically use C programming language when developing their software solutions. In the previous module (MSP430 Assembly Language) we have discussed modern software development environments (SDEs) that integrate editors, assembler, compiler, linker, stand-alone simulator, embedded emulator or debugger, and flash programmer and you have learned how to use them for developing assembly language programs.

Figure 1 shows a typical development flow using C/C++ marked by gray area and it is very similar to the one discussed in Module 04. It starts from one or more C/C++ source files (with extensions .c or .cpp). These files are translated into assembly program files using a **C/C++ Compiler**. Assembly codes are translated using an **Assembler** into object files. The object files together with libraries are tied together by a **Linker** that produces an executable object file. The executable file is then downloaded into a MSP430 device using a flash programmer. Finally the target device executes the code. Below is a detailed description of all blocks from Figure 1, including side branches of development flow dealing with libraries.

- **The compiler** accepts C/C++ source files and produces MSP430 assembly language codes.
- **The assembler** translates assembly language files into machine language relocatable object files.
- **The linker** combines relocatable object files into a single absolute executable object file. As it creates the executable file, it performs relocation and resolves external references. The linker accepts relocatable object files and object libraries as input.
- **The archiver** allows you to collect a group of files into a single archive file, called a library. The archiver allows you to modify such libraries by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object files.
- **The library-build utility** automatically builds the run-time-support library if compiler and linker options require a custom version of the library. The run-time-support libraries contain the *standard ISO C and C++ library functions*, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler.
- **The hex conversion utility** converts an object file into other object formats. You can download the converted file to an EPROM programmer.
- **The absolute lister** accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations.
- **The cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files.

- **The C++ name demangler** is a debugging aid that converts names mangled by the compiler back to their original names as declared in the C++ source code. As shown in Figure 1-1, you can use the C++ name demangler on the assembly file that is output by the compiler; you can also use this utility on the assembler listing file and the linker map file.
- **The disassembler** decodes object files to show the assembly instructions that they represent.

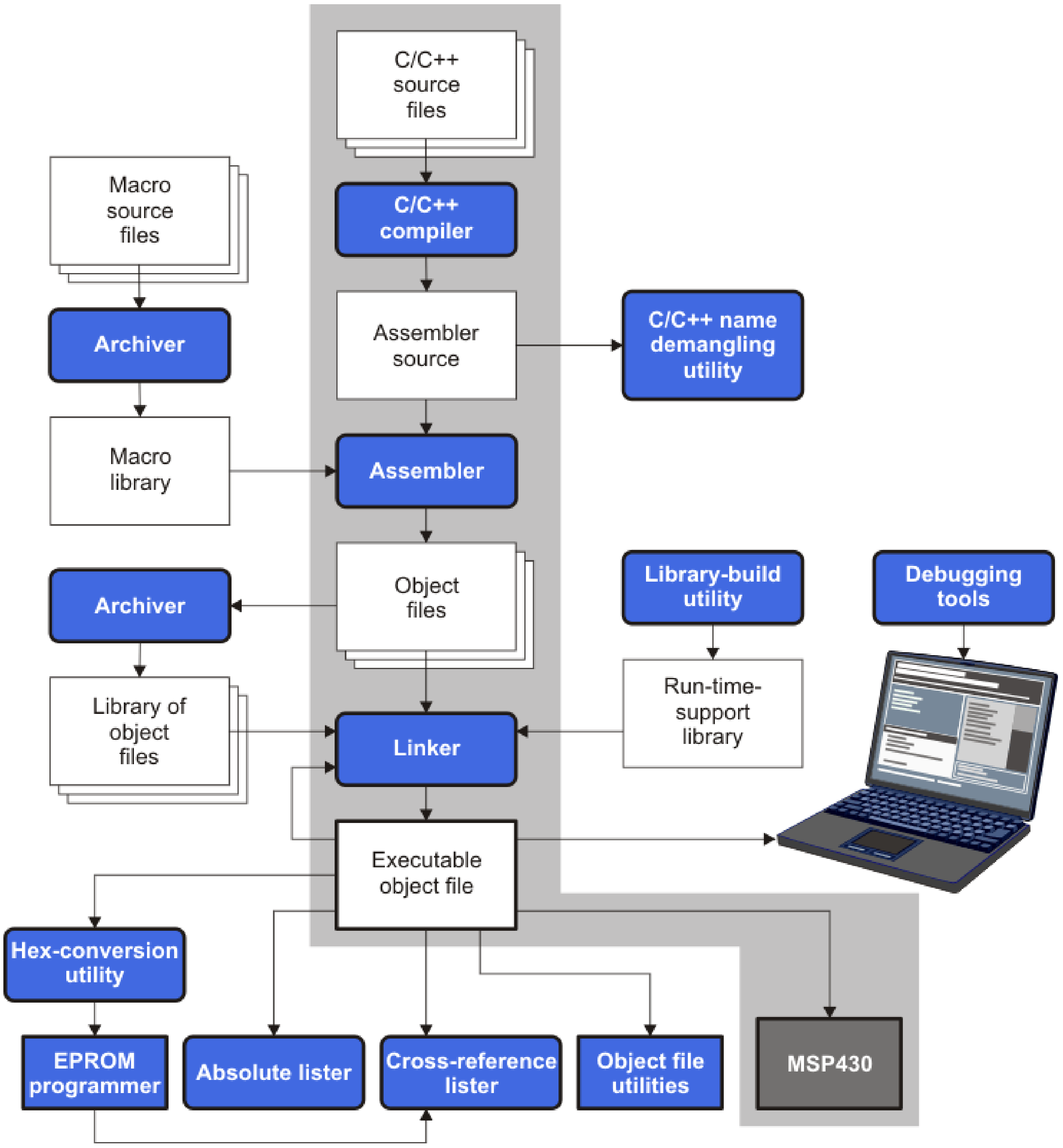

Figure 1. Design flow from C/C++ files to machine code.

2 MSP430 C/C++ Language Implementation: Data Types

The TI C/C++ compiler used in Code Composer Studio supports C/C++ language standards developed by a committee of the American National Standards Institute (ANSI) and adopted by the International Standard Organization (ISO). The C compiler supports the 1989 (C89, ISO/IEC 9899:1990 C standard) and 1999 (C99, ISO/IEC 9899:1999) versions of the C language. The ANSI/ISO standard identifies some features of the C language that may be affected by characteristics of the target processor, run-time environment, or host environment. This set of features can differ among standard compilers. The MSP430 Optimizing C/C++ Compiler User's Guide is a great source of information about C/C++ language implementation and compiler specifics (the document for 21.6.0 LTS version of the compiler is available at https://www.ti.com/lit/pdf/SLAU132Y).

Table 1 lists C/C++ data types, the size, alignment, representation, and range of each scalar data type for the MSP430 compiler. Many of the ranges are available as standard macros in the header file *limits.h*. All single-byte data can be stored at any address in address space, even or odd (alignment is 8). All multi-byte data are stored at even address (alignment is 16).

Table 1. MSP430 C/C++ Data Types.

	Size [bits]	Align-ment	Representation	Range	
Type				**Minimum**	**Maximum**
signed char	8	8	Binary	-128	127
char	8	8	ASCII	0	255
unsigned char	8	8	Binary	0	255
bool (C99)	8	8	Binary	0 (false)	1 (true)
_Bool (C99)	8	8	Binary	0 (false)	1 (true)
bool (C++)	8	8	Binary	0 (false)	1 (true)
short, signed short	16	16	2s complement	-32,768	32,767
unsigned short	16	16	Binary	0	65,535
int, signed int	16	16	2's complement	-32,768	32,767
unsigned int	16	16	Binary	0	65,535
long, signed long	32	16	2's complement	-2,147,483,648	2,147,483,647
unsigned long	32	16	Binary	0	4,294,967,295
long long, signed long long	64	16	2's complement	-9,223,372,036, 854,775,808	9,223,372,036, 854,775,807
unsigned long long	64	16	Binary	0	18,446,744,073, 709,551,615
enum	varies	16	2's complement	varies	varies
float	32	16	IEEE 32-bit	1.175494e-38	3.40282346e+38
double	64	16	IEEE 64-bit	2.22507385e-308	1.79769313e+308

MSP430 devices support multiple data and code memory models. The code and data model affects the size, alignment, and storage space used for function pointers, data pointers, the size_t type, and ptrdiff_t type. For large code and data models (address space is 2^{20} bytes) pointers are always stored in units with a size of a power of 2; thus, 20-bit code and data pointers are stored in 32-bits. All multi-byte data are aligned to even addresses (alignment is 16). In this course all assignments will typically use small code and small data model, i.e., the baseline MSP430 ISA is will suffice to implement them.

Table 2. Data Sizes for MSP430 Pointers

Code and Data Model	Type	Size	Storage	Alignment
small code model	function pointer	16	16	16
large code model	function pointer	20	32	16
small data model	data pointer	16	16	16
small data model	size_t	16	16	16
small data model	ptrdiff_t	16	16	16
large data model	data pointer	20	32	16
large data model	size_t	32	32	16
large data model	ptrdiff_t	32	32	16

Things to remember 2-1. C/C++ common data types and code/data models.

Understand common data types in C/C++ and MSP430 code and data models.

3 Data Allocation Examples

Let us consider a C program that includes several declarations with scalar data types and several C statements that initialize the data as shown in Code 1. The program does not result in any useful output (either direct or indirect), so with optimization switches on, the compiler may entirely eliminate the code. If you turn the optimizations off and in the assembler options ask for assembly code to be generated, you will get an assembler file (CDataAllocationDemo.asm) as shown in Code 2.

```
/*------------------------------------------------------------------------------
 *   File:  CDataAllocationDemo.c (C Data Allocation Demo Program)
 *
 *   Board: MSP-EXP430F5529LP Development Kit
 *
 *   Description: Program illustrates how C compiler allocates space in memory
 *
*   Author: Aleksandar Milenkovic, milenkovic@computer.org
 *  Date:   September 2018
*-----------------------------------------------------------------------------*/

```

```c
#include <msp430.h>

int main(void) {
      int i1, i2;
      unsigned int ui1;
      short int sint1;
      long int lint2;
       int a[4];
      // Stop watchdog timer to prevent time out reset
      WDTCTL = WDTPW + WDTHOLD;
      i1 = 2; i2 = -2;
      ui1=65535;
      sint1=127;
      lint2=128243;
      a[0]=20; a[1]=9;
      return 0;
}
```

Code 1. C program that declares several scalar data types (CDataAllocationDemo.c).

The commented line in the assembly file (starting with ";") contain a detailed compiler-generated commentary as well as C program statements inter-listed with assembly instructions. Thus, before the main program entry label (line 32), the compiler lists registers affected by this program (registers SP, SR, and R12) as well as the stack frame occupying 20 bytes. Going through the list of variables in the C code, the integer variables i1, i2 require 4 bytes (2 variables, 2 bytes each), ui1 requires 1x2 = 2 bytes, sint1 requires 1x2 bytes, lint2 requires 1x4=4 bytes, and integer array a requires 4x2=8 bytes, for the total of 20 bytes. The first instruction in line 34 reserves 20 bytes on the top of the stack by executing SUB.W #20, SP. After stopping the watchdog timer, a series of machines instructions initializes variables. By working line-by-line we can determine an exact order of variables on the stack. Table 3 illustrates the content of the stack once all variables are allocated and initialized. Its layout is inferred by analyzing the assembly instructions from Code 2. At the end of the program a constant 0 is placed into R12 (return 0), the stack frame is freed (which instruction is used for that?), and a RET instruction executed.

```
 1  ;******************************************************************************
 2  ;* MSP430 G3 C/C++ Codegen                                              PC
 3  v20.2.2.LTS *
 4  ;* Date/Time created: Sat Sep 26 15:08:21 2020                               *
 5  ;******************************************************************************
 6          .compiler_opts --abi=eabi --diag_wrap=off --hll_source=on --
 7  mem_model:code=small --mem_model:data=small --object_format=elf --
 8  silicon_errata=CPU21 --silicon_errata=CPU22 --silicon_errata=CPU40 --
 9  silicon_version=msp --symdebug:none
10  ;       C:\ti\ccs1010\ccs\tools\compiler\ti-cgt-msp430_20.2.2.LTS\bin\acpia430.exe -
11  @C:\\Users\\milenka\\AppData\\Local\\Temp\\{1AB06A9D-E49D-40E4-80D9-D2234B6A0F33}
12          .sect   ".text:main"
13          .clink
14          .global     main
15  ;----------------------------------------------------------------------
16  ;  24 | int main(void) {
17  ;  25 | int i1, i2;
```

```
;  26 | unsigned int ui1;
;  27 | short int sint1;
;  28 | long int lint2;
;  29 |  int a[4];
;  30 | // Stop watchdog timer to prevent time out reset
;----------------------------------------------------------------------

;***************************************************************************
;* FUNCTION NAME: main                                                     *
;*                                                                         *
;*   Regs Modified     : SP,SR,r12                                         *
;*   Regs Used         : SP,SR,r12                                         *
;*   Local Frame Size  : 0 Args + 20 Auto + 0 Save = 20 byte               *
;***************************************************************************
main:
;* -----------------------------------------------------------------------*
        SUB.W     #20,SP                ; []
;----------------------------------------------------------------------
;  31 | WDTCTL = WDTPW + WDTHOLD;
;----------------------------------------------------------------------
        MOV.W     #23168,&WDTCTL+0      ; [] |31|
;----------------------------------------------------------------------
;  32 | i1 = 2; i2 = -2;
;----------------------------------------------------------------------
        MOV.W     #2,12(SP)             ; [] |32|
        MOV.W     #65534,14(SP)         ; [] |32|
;----------------------------------------------------------------------
;  33 | ui1=65535;
;----------------------------------------------------------------------
        MOV.W     #65535,16(SP)         ; [] |33|
;----------------------------------------------------------------------
;  34 | sint1=127;
;----------------------------------------------------------------------
        MOV.W     #127,18(SP)           ; [] |34|
;----------------------------------------------------------------------
;  35 | lint2=128243;
;----------------------------------------------------------------------
        MOV.W     #62707,8(SP)          ; [] |35|
        MOV.W     #1,10(SP)             ; [] |35|
;----------------------------------------------------------------------
;  36 | a[0]=20; a[1]=9;
;----------------------------------------------------------------------
        MOV.W     #20,0(SP)             ; [] |36|
        MOV.W     #9,2(SP)              ; [] |36|
;----------------------------------------------------------------------
;  37 | return 0;
;----------------------------------------------------------------------
        MOV.W     #0,r12                ; [] |37|
        ADD.W     #20,SP                ; []
        RET       ; []
        ; []
;***************************************************************************
;* UNDEFINED EXTERNAL REFERENCES                                           *
;***************************************************************************
       .global      WDTCTL
```

```

;******************************************************************************
;* BUILD ATTRIBUTES                                                           *
;******************************************************************************
       .battr "TI", Tag_File, 1, Tag_LPM_INFO(1)
       .battr "TI", Tag_File, 1,
Tag_PORTS_INIT_INFO("012345678901ABCDEFGHIJ0000000000001111000000000000000000000000000
00")
       .battr "TI", Tag_File, 1, Tag_LEA_INFO(1)
       .battr "TI", Tag_File, 1, Tag_HW_MPY32_INFO(2)
       .battr "TI", Tag_File, 1, Tag_HW_MPY_ISR_INFO(1)
       .battr "TI", Tag_File, 1, Tag_HW_MPY_INLINE_INFO(1)
       .battr "mspabi", Tag_File, 1, Tag_enum_size(3)
```

Code 2. Assembly code produced by the C compiler (CDataAllocationDemo.asm).

Table 3. Variables allocated on the program stack for CDataAllocationDemo.c when executed on MSP430F5529.

Address	Memory[15:0] [hex]	Offset relative to current SP	Variable
0x4400	---		*Original Top of the Stack*
0x43FE	0x00FF	18	sint1
0x43FC	0xFFFF	16	ui1
0x43FA	0xFFFE	14	i2
0x43F8	0x0002	12	i1
0x43F6	0x0001	10	lint2 (upper)
0x43F4	0xF4F3	8	lint2 (lower)
0x43F2	-	6	a[3]
0x43F0	-	4	a[2]
0x43EE	0x0009	2	a[1]
0x43EC	0x0014	0 <= SP	a[0]

The code executed on MSP430F5529 with small code and data model (address space is 64 KB) will have the original top of the stack placed right above the physical RAM memory (address is 0x4400). Please note that from Table 3 you can determine the addresses of these variables in memory. Thus, the base address of the long integer lint2 is 0x43F4. Please note that though this variable occupies two words at address 0x43F4 (contains the lower 16 bits of the variable) and 0x43F6 (contains the upper 16 bits of the variable), the base address is 0x43F4 (address of the first byte a multi-byte object occupies). Also, you can see the little-endian placement policy in action. Next, please note how integer array *a* is placed in memory. Expectedly, the element with index 0, *a[0]*, is placed at the address 0x43EC, the element with index 1, *a[1]*, is placed at the next word address 0x43EE, the element with index 2, *a[2]*, is placed at the address 0x43F0, and the element with index 3, *a[3]*, is placed at the address 0x43F2.

The C programming language supports storage class specifiers as shown in Table 4. We have already seen in action the *auto* specifier: all variables in the CDataAllocationDemo.c were deallocated before exiting the main function. Although the *register* specifier is also automatic, the compiler did not allocate any of the variables in general-purpose registers. Try to modify the CDataAllocationDemo.c program, line 15 as follows: register int i1, i2. Analyze the corresponding assembly code. Is the assembly code different? If yes, what are the differences? Also, changing optimization levels in the C/C++ compiler may also result in different data allocations.

Table 4. C Storage Class Specifiers

Storage Class Specifier	Description
auto	Variable is no longer required once a block has been left; default.
register	Hint to compiler to allocate a variable in a register; typically automatic, but there are not guarantees that compiler will do so. Such variables cannot be accessed by means of pointers.
static	Allows a local variable to retain its value when a block is reentered; initialized once by the compiler.
extern	Indicates that the variables is defined outside the current block

Table 5 shows C storage class modifiers with their description. Type conversions are dealing with C statements that perform operations on variables with different data types. E.g., what happens when you have an integer variable (2 bytes) that receives a value of a long integer (4 bytes)? C programming language typically allows automatic type conversions, but they may not always be a safe thing to use, especially when our understanding on how these conversions work does not match particular compiler implementations. Thus, using explicit type conversions is strongly encouraged (and many modern compilers will actually force you to do so).

Table 5. C Storage Class Modifiers

Storage Class Modifiers	Description
volatile	Placing keyword volatile in front of the declaration of a variable indicates that it can be changed externally (e.g., a location is associated with a peripheral). The compiler will never put such variables in processor registers.
const	Indicates that the variable may not be changed during program execution. Such variables cannot be changed unintentionally in a program, but can be changed externally as a result of an I/O operation
type conversions	Conversion between different data types are either done automatically or explicitly using cast operations.

`Things to remember 3-1. C/C++ storage class specifiers and modifiers.`

Understand storage class specifiers in C (auto, register, static, extern) and modifiers (volatile, const). Be aware of implicit type conversions – it is always better to handle type conversions explicitly.

4 Global and Local Variables, Passing Parameters by Value/Reference

Local variables are defined within functions or code sub-blocks within a function. Their scope is the particular function they are defined in (or the sub-block of a function) and cannot be used from outside the function. They are normally lost when a return from the function is made (an exception are local variable defined with the static modifier). Global variables are defined outside functions. They can be accessed both from inside and outside a function.

To illustrate global and local variables, we consider a C program shown in Code 3. In this program we first declare a global variable *gi* of type int initialized to 5. Inside the main, a local variable *li1* of type int, as well as two char variables *ch1* and *ch2* are declared. The main function calls two functions *lc2uc* and *plus10*. The prototype of the *lc2uc* function indicates that this function accepts a pointer to a character as an input and returns a character. The prototype of the *plus10* function indicates that it accepts one integer input parameter (int i) and returns an integer.

```
/*********************************************************************************
 *   File:  GLVarsFunctionsDemo.c (Illustrates global, local variables, function
calls)
 *
 *   Board: MSP-EXP430F5529LP Development Kit
 *
 *   Description: Program illustrates how C compiler allocates space in memory
 *
 *   Author: Aleksandar Milenkovic, milenkovic@computer.org
 *   Date:   September 2020
*********************************************************************************/
#include <msp430.h>
#include <stdio.h>

int gi = 5; // global variable, initialized to 5

char lc2uc(char *pc); // function prototype
int plus10(int i);   // function prototype

void main(void) {
  int li1 = 2;     // local var, li1=2
  char ch1 = 'a'; // local var, ch1='a'
  char ch2;       // local var, ch2 not initialized

  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;
  ch2 = lc2uc(&ch1);  // call lc2uc function
  li1 = li1 + gi;     // update li1
29    li1 = plus10(li1);  // call plus10 function
30    printf("li1=%d, gi=%d\n", li1, gi);
31    printf("ch1=%c, ch2=%c\n", ch1, ch2);
32  }
33
34  char lc2uc(char *pc) {
35      char tc;
36      tc = *pc;
37      if ((tc >= 'a') && (tc <= 'z')) tc = tc + ('A' - 'a'); // convert lowercase to
38  uppercase
39      *pc = tc;                                             //
40      return (tc+1);
41  }
42
43  int plus10(int i) {
44      i = i + 10;
45      gi = gi + 10;
46      return 20;
47  }
```

Code 3. C program illustrating global/local variables and function calls (GLVarsFunctionsDemo.c).

The function *lc2uc* declares a local variable *tc* of type char, which is initialized to take the value of a character passed as an input parameter passed by a reference (the *lc2uc* gets a pointer to the input parameter), `tc = *pc`. The code inside this function converts a lower case character into its upper case counterpart. Using the statement `*pc = tc`, the variable declared in a caller is updated by the new value. This is an example of passing parameters by reference, where statements inside the function have global effects. The function then returns (tc+1). The function *plus10* takes an input parameter (int i) passed by value, adds 10 to the input parameter, adds 10 to the global variable *gi*, and returns constant 20.

In line 27, `ch2 = lc2uc(&ch1)`, we call *lc2uc* and pass the address of the character *ch1*. Thus, the *lc2uc* is going to update *ch1*, so its original value 'a' is updated with its uppercase counterpart 'A'. In addition, *lc2uc* returns (tc+1), meaning that ch2 gets 'B'. In line 28, `li1 = li1+gi`, a new value of the variable *li1* is determined, li1=2+5=7. Next, in line 29, `li1 = plus10(li1)`, a copy of *li1* is passed into the *plus10* function. A copy of this variable is then increased by 10, the global variable *gi* is increased by 10, and the function returns constant 20. Thus, li1 is going to have value 20 (please note that actions on the local copy inside the function *plus10* do not have global effects). You should see the following output on the console:

```
li1=20, gi=15
ch1=A, ch2=B
```

Code 4 shows the interlisted assembly code created by the compiler for GLVarsFunctionsDemo.c. Please note how the global variable *gi* is defined in the initialized data section of the assembly program. As such, it is visible to all functions. Analyze assembly code for the *plus10* function (lines 32-38). How does it receive its input parameter? What happens with a C statement adding 10 to a local copy passed into the *plus10* function? How does the compiler pass the parameter into the *plus10* function? How does the compiler pass the parameter into *lc2uc*?

```
;******************************************************************************
;* MSP430 G3 C/C++ Codegen                                                  PC
v20.2.2.LTS *
;* Date/Time created: Sat Sep 26 23:43:47 2020                                 *
;******************************************************************************
      .compiler_opts --abi=eabi --diag_wrap=off --hll_source=on --
mem_model:code=small --mem_model:data=small --object_format=elf --
silicon_errata=CPU21 --silicon_errata=CPU22 --silicon_errata=CPU23 --
silicon_errata=CPU40 --silicon_version=msp --symdebug:none
      .global      gi
      .data
      .align 2
      .elfsym      gi,SYM_SIZE(2)
gi:
      .bits        0x5,16
                   ; gi @ 0

;     C:\ti\ccs1010\ccs\tools\compiler\ti-cgt-msp430_20.2.2.LTS\bin\opt430.exe
C:\\Users\\milenka\\AppData\\Local\\Temp\\{39F3C0AC-3032-4CCC-AAC6-6D4CECABD9DE}
C:\\Users\\milenka\\AppData\\Local\\Temp\\{8B4E6015-E28B-4E52-808E-80DCBFE96C48}
      .sect  ".text:plus10"
      .clink
      .global      plus10

;*****************************************************************************
;* FUNCTION NAME: plus10                                                     *
;*                                                                           *
;*   Regs Modified     : SP,SR,r12                                           *
;*   Regs Used         : SP,SR,r12                                           *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                   *
;*****************************************************************************
plus10:
;* --------------------------------------------------------------------------*
;** 43 -----------------------    gi += 10;
        ADD.W     #10,&gi+0             ; [] |43|
;** 44 -----------------------    return 20;
        MOV.W     #20,r12               ; [] |44|
        RET       ; []
        ; []
      .sect  ".text:lc2uc"
      .clink
      .global      lc2uc

;*****************************************************************************
;* FUNCTION NAME: lc2uc                                                      *
;*                                                                           *
;*   Regs Modified     : SP,SR,r12,r15                                       *
;*   Regs Used         : SP,SR,r12,r15                                       *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                   *
;*****************************************************************************
lc2uc:
;* --------------------------------------------------------------------------*
;** 35 -----------------------    if ( (tc = *pc) < 97 || tc > 122 ) goto g3;
```

```
        MOV.B     @r12,r15              ; [] |35|
        CMP.B     #97,r15               ; [] |35|
        JLO       $C$L1                 ; [] |35|
                                          ; [] |35|
;* --------------------------------------------------------------------------*
        CMP.B     #123,r15              ; [] |35|
        JHS       $C$L1                 ; [] |35|
                                          ; [] |35|
;* --------------------------------------------------------------------------*
;** 36 -----------------------    tc -= 32;
        SUB.B     #32,r15               ; [] |36|
;* --------------------------------------------------------------------------*
$C$L1:
;**    -----------------------g3:
;** 37 -----------------------    *pc = tc;
        MOV.B     r15,0(r12)            ; [] |37|
;** 38 -----------------------    return (unsigned char)(tc+1);
        MOV.W     #1,r12                ; [] |38|
        ADD.B     r15,r12               ; [] |38|
        RET       ; []
        ; []
       .sect  ".text:main"
       .clink
       .global      main

;****************************************************************************
;* FUNCTION NAME: main                                                      *
;*                                                                          *
;*   Regs Modified     : SP,SR,r10,r11,r12,r13,r14,r15                      *
;*   Regs Used         : SP,SR,r10,r11,r12,r13,r14,r15                      *
;*   Local Frame Size  : 6 Args + 2 Auto + 2 Save = 10 byte                 *
;****************************************************************************
main:
;* --------------------------------------------------------------------------*
        PUSH.W    r10                   ; []
        SUB.W     #8,SP                 ; []
;** 21 -----------------------    ch1 = 97u;
        MOV.B     #97,6(SP)             ; [] |21|
;** 25 -----------------------    WDTCTL = 23168u;
        MOV.W     #23168,&WDTCTL+0      ; [] |25|
;** 26 -----------------------    ch2 = lc2uc(&ch1);
        MOV.W     SP,r12                ; [] |26|
        ADD.W     #6,r12                ; [] |26|
        CALL      #lc2uc                ; [] |26|
                                          ; [] |26|
        MOV.W     r12,r10               ; [] |26|
;** 27 -----------------------    li1 = gi+2;
;** 28 -----------------------    li1 = plus10(li1);
        CALL      #plus10               ; [] |28|
                                          ; [] |28|
;** 29 -----------------------    printf[VA]("li1=%d, gi=%d\n", li1, gi);
        MOV.W     #$C$SL1+0,0(SP)       ; [] |29|
        MOV.W     r12,2(SP)             ; [] |29|
        MOV.W     &gi+0,4(SP)           ; [] |29|
        CALL      #printf               ; [] |29|
                                                ; [] |29|
;** 30 -----------------------    printf[VA]("ch1=%c, ch2=%c\n", (int)ch1, (int)ch2);
        MOV.W     #$C$SL2+0,0(SP)       ; [] |30|
        MOV.B     6(SP),r15             ; [] |30|
        MOV.W     r15,2(SP)             ; [] |30|
        MOV.B     r10,r10               ; [] |30|
        MOV.W     r10,4(SP)             ; [] |30|
        CALL      #printf               ; [] |30|
                                          ; [] |30|
;**    -----------------------    return;
        ADD.W     #8,SP                 ; []
        POP       r10                   ; []
        RET       ; []
        ; []
;******************************************************************************
;* STRINGS                                                                    *
;******************************************************************************
       .sect  ".const:.string"
       .align 2
$C$SL1:      .string      "li1=%d, gi=%d",10,0
       .align 2
$C$SL2:      .string      "ch1=%c, ch2=%c",10,0
;*****************************************************************************
;* UNDEFINED EXTERNAL REFERENCES                                             *
;*****************************************************************************
       .global      WDTCTL
       .global      printf

;******************************************************************************
;* BUILD ATTRIBUTES                                                           *
;******************************************************************************
       .battr "TI", Tag_File, 1, Tag_LPM_INFO(1)
       .battr "TI", Tag_File, 1,
Tag_PORTS_INIT_INFO("012345678901ABCDEFGHIJ0000000000001111000000000000000000000000
00")
       .battr "TI", Tag_File, 1, Tag_LEA_INFO(1)
       .battr "TI", Tag_File, 1, Tag_HW_MPY32_INFO(2)
       .battr "TI", Tag_File, 1, Tag_HW_MPY_ISR_INFO(1)
       .battr "TI", Tag_File, 1, Tag_HW_MPY_INLINE_INFO(1)
       .battr "mspabi", Tag_File, 1, Tag_enum_size(3)
```

Code 4. Assembly program created by the compiler by translating GLVarsFunctionsDemo.c.

Things to remember 4-1. Local vs. global variables, passing parameters by value vs. passing parameters by reference.

Understand difference between global and local variables. Understand the difference between passing parameters to subroutine by value and by reference and implications of each.

5 Translation of High-Level Language Constructs

In this section we will take a look how the TI C compiler translates high-level language constructs like a for loop and a switch statement into assembly code. Let us consider a source code shown in Code 5. We have a for loop that sums up first 10 unsigned integers and displays the result on port PB, which is a combination of ports P3 (lower 8 bits) and P4 (upper 8 bits). Next, we read an input port P1 (P1IN register), and depending on its input value, assign P2OUT one of 3 possible values in a switch statement. Please note that MSP430 parallel ports are 8-bit long and consist of PxOUT (output register), PxIN (input register), and PxDIR (direction register). On power-up all parallel ports are configured as inputs (PxDIR=0x00), meaning that the state of physical port pins is latched in the corresponding PxIN register – if an individual port pin is grounded the corresponding bit in the PxIN register is set to a logic ‘0’; if the port pin is on a high voltage, the corresponding bit in the PxIN register is set to a logic ‘1’. Please note that for ports P3 and P4, their corresponding P3DIR and P4DIR are not set to 0xFF, so the value written in registers P3OUT and P4OUT will not be visible on physical port pins.

```

/********************************************************************************
 *   File:  C2ASMDemo.c (Illustrates translation of HLL constructs into assembly)
 *
 *   Board: MSP-EXP430F5529LP Development Kit
 *
 *   Description: Program illustrates how the C compiler translates high-level
 *                language constructs into assembly code
 *
 *   Clocks: ACLK = 32.768kHz, MCLK = SMCLK = default DCO
 *
 *                 MSP430F5529
 *               -----------------
 *           /|\|                 |
 *            | |                 |
 *            --|RST              |
 *              |                 |
 *              |                 |
 *
 *   Author: Aleksandar Milenkovic, milenkovic@computer.org
 *   Date:   September 2020
*******************************************************************************/

#include <msp430.h>

int main(void) {
      WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    unsigned int i = 0;
    unsigned char ch;
    unsigned int sum = 0;

    for(i=0; i<10; i++) sum += i;
    PBOUT = sum;  // PB port includes P3 (lower byte) and P4 (upper byte)

    ch=P1IN;
37      switch(ch) {
38        case 0: P2OUT=0x01; break;
39        case 1: P2OUT=0x02; break;
40        default: P2OUT=0x80;
41      }
42    return 0;
43    }
```

Code 5. C2ASM.c Demo Program.

Code 6 shows the assembly program generated by the C compiler for the program shown in Code 5. Analyze the program and answer the following questions: (a) How does the compiler handle variables *i* and *sum*? (b) How does the compiler implement the *for* loop (hint: look at lines 43-60)? (c) How does the compiler test whether the input variable *ch* is equal to 0 or 1? Please note JHS assembly mnemonic is synonym with JC (jump if carry). Why do you think that instruction is used? Go to the MSP430 Compiler Optimization settings in the Project Properties and select Local Optimizations. Analyze the newly generated assembly code? How is the new code different from the one shown in Code 6?

```
1   ;******************************************************************************
2   ;* MSP430 G3 C/C++ Codegen                                                PC
3   v20.2.2.LTS *
4   ;* Date/Time created: Sat Sep 26 19:03:11 2020                                  *
5   ;******************************************************************************
6         .compiler_opts --abi=eabi --diag_wrap=off --hll_source=on --
7   mem_model:code=small --mem_model:data=small --object_format=elf --
8   silicon_errata=CPU21 --silicon_errata=CPU22 --silicon_errata=CPU23 --
9   silicon_errata=CPU40 --silicon_version=msp --symdebug:none
10  ;     C:\ti\ccs1010\ccs\tools\compiler\ti-cgt-msp430_20.2.2.LTS\bin\opt430.exe
11  C:\\Users\\milenka\\AppData\\Local\\Temp\\{4018CF22-9B41-4BAB-9D0A-4D4112BCA31A}
12  C:\\Users\\milenka\\AppData\\Local\\Temp\\{4744528D-4FD2-4348-BDEB-14CCCD4A0FC3}
13        .sect  ".text:main"
14        .clink
15        .global      main
16  ;----------------------------------------------------------------------
17  ;  18 | int main(void) {
18  ;----------------------------------------------------------------------
19
20  ;*****************************************************************************
21  ;* FUNCTION NAME: main                                                        *
22  ;*                                                                            *
23  ;*   Regs Modified     : SP,SR,r12,r14,r15                                    *
24  ;*   Regs Used         : SP,SR,r12,r14,r15                                    *
25  ;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                    *
26  ;*****************************************************************************
27  main:
28  ;* --------------------------------------------------------------------------*
29  ;----------------------------------------------------------------------
30  ;  19 | WDTCTL = WDTPW | WDTHOLD;       // stop watchdog timer
31  ;  21 | unsigned int i = 0;
32  ;  22 | unsigned char ch;
33  ;----------------------------------------------------------------------
34          MOV.W     #23168,&WDTCTL+0      ; [] |19|
;----------------------------------------------------------------------
;  23 | unsigned int sum = 0;
;----------------------------------------------------------------------
        MOV.W     #0,r14                ; [] |23|
;----------------------------------------------------------------------
;  25 | for(i=0; i<10; i++) sum += i;
;----------------------------------------------------------------------
        MOV.W     #0,r15                ; [] |25|
        CMP.W     #10,r15               ; [] |25|
        JHS       $C$L2                 ; [] |25|
                                          ; [] |25|
;* --------------------------------------------------------------------------*
;*   BEGIN LOOP $C$L1
;*
;*   Loop source line                : 25
;*   Loop closing brace source line  : 25
;*   Known Minimum Trip Count        : 1
;*   Known Maximum Trip Count        : 4294967295
;*   Known Max Trip Count Factor     : 1
;* --------------------------------------------------------------------------*
$C$L1:
        ADD.W     r15,r14               ; [] |25|
        ADD.W     #1,r15                ; [] |25|
        CMP.W     #10,r15               ; [] |25|
        JLO       $C$L1                 ; [] |25|
                                          ; [] |25|
;* --------------------------------------------------------------------------*
$C$L2:
;----------------------------------------------------------------------
;  26 | PBOUT = sum;  // PB port includes P3 (lower byte) and P4 (upper byte)
;----------------------------------------------------------------------
        MOV.W     r14,&PBOUT+0          ; [] |26|
;----------------------------------------------------------------------
;  28 | ch=P1IN;
;----------------------------------------------------------------------
        MOV.B     &PAIN_L+0,r15         ; [] |28|
;----------------------------------------------------------------------
;  29 | switch(ch) {
;  30 |   case 0: P2OUT=0x01; break;
;  31 |   case 1: P2OUT=0x02; break;
;----------------------------------------------------------------------
        MOV.B     r15,r15               ; [] |29|
        TST.W     r15                   ; [] |29|
        JEQ       $C$L4                 ; [] |29|
                                          ; [] |29|
;* --------------------------------------------------------------------------*
        SUB.W     #1,r15                ; [] |29|
        JEQ       $C$L3                 ; [] |29|
                                          ; [] |29|
;* --------------------------------------------------------------------------*
;----------------------------------------------------------------------
;  32 | default: P2OUT=0x80;
;----------------------------------------------------------------------
        MOV.B     #128,&PAOUT_H+0       ; [] |32|
        JMP       $C$L5                 ; [] |33|
                                            ; [] |33|
;* ----------------------------------------------------------------------------*
$C$L3:
        MOV.B     #2,&PAOUT_H+0           ; [] |31|
        JMP       $C$L5                   ; [] |31|
                                            ; [] |31|
;* ----------------------------------------------------------------------------*
$C$L4:
        MOV.B     #1,&PAOUT_H+0           ; [] |30|
;* ----------------------------------------------------------------------------*
$C$L5:
;----------------------------------------------------------------------
;  34 | return 0;
;----------------------------------------------------------------------
        MOV.W     #0,r12                  ; [] |34|
        RET       ; []
        ; []
;*****************************************************************************
;* UNDEFINED EXTERNAL REFERENCES                                             *
;*****************************************************************************
      .global     PAIN_L
      .global     PAOUT_H
      .global     PBOUT
      .global     WDTCTL

;******************************************************************************
;* BUILD ATTRIBUTES                                                           *
;******************************************************************************
      .battr "TI", Tag_File, 1, Tag_LPM_INFO(1)
      .battr "TI", Tag_File, 1,
Tag_PORTS_INIT_INFO("012345678901ABCDEFGHIJ0000000000001111000000000000000000000000
00")
      .battr "TI", Tag_File, 1, Tag_LEA_INFO(1)
      .battr "TI", Tag_File, 1, Tag_HW_MPY32_INFO(2)
      .battr "TI", Tag_File, 1, Tag_HW_MPY_ISR_INFO(1)
      .battr "TI", Tag_File, 1, Tag_HW_MPY_INLINE_INFO(1)
      .battr "mspabi", Tag_File, 1, Tag_enum_size(3)
```

Code 6. Compiler-generated assembly code for C2ASM.c Demo Program.

6 Pointers in C Language

Data pointers in C are variables that contain an address of an object of certain type. Let us consider a code snippet shown in Code 7. We declare a number of variables of different types. They all have a modifier volatile that will force the compiler to allocate them in the main memory rather than any of the general-purpose registers. For simplicity we are going to assume that SP initially points to 0x4400. In addition, we are going to assume that the variables are allocated on the stack in the order of appearance in the program (please note that we have already seen that the compiler usually does not do that, rather it has its own policy regarding ordering data on the stack).

```
int main(void) {
```

```
2       volatile unsigned int a = 4, b = 2;
3       volatile long int c = -4, d = 2;
4       volatile char mych]4] = {‘4’, ‘3’, ‘2’, ‘1’};
5       volatile long int *pli = &d;
6       volatile int *pi = &b;
7
8       pli = pli + 1;
9       pi = pi – 6;
10      *pi = a + *pi;
11  }
```

Code 7. Pointers Demo Code.

Table 6 illustrates the content of stack once all variables are allocated and initialized (lines 1-6 in Code 7). Line 5 declares a variable with the name pli of type “pointer to a long int” and initializes it with the address of the *long int* variable d. Please note that in this example we assume the small data model – address space is 64KB and pointers are 16-bit long. Thus, to allocate space for pli one word is used and its initial value is the address of the variable d. In our case it is 0x43F4 (the base address of d). Similarly, line 6 declares a variable with the name pi of type “point to an int” and initializes with the address of variable b. Again, the size of variable pi is the size of the address (one word), and the the address of b is 0x43FC.

Table 6. Variables allocated on the program stack for example in Code 7 after lines 1-6 are carried out.

Address	Memory[15:0] [hex]	Offset relative to current SP	Variable
0x4400	---		*Original Top of the Stack*
0x43FE	0x0004	18	a
0x43FC	0x0002	16	b
0x43FA	0xFFFF	14	c, upper word
0x43F8	0xFFFC	12	c, lower word
0x43F6	0x0000	10	d, upper word
0x43F4	0x0002	8	d, lower word
0x43F2	0x3132	6	mych[3], mych[2]
0x43F0	0x3334	4	mych[1], mych[0]
0x43EE	0x43F2	2	pli
0x43EC	0x43FC	0 <= SP	pi

Let us now consider C statement is lines 8-10. In line 8, pli = pli + 1, the pointer pli is incremented by 1. The meaning of this statement in C is “increment the pointer so it points to the next object of the declared type.” As the size of the *long int* type is 4 bytes, in practice we want to add 1x4=4 bytes to the current value of pli. Thus the statement in line 8 can be translated into assembly as follows:

```
ADD.W #4, 2(SP) ; move to the next element of type long int
```

What if we have a statement pli = pli + 4? In this case we will say pli is of type long int, the size of long int is 4 bytes, thus, we want to add 4x4=16 bytes to the current value of pli. As you can see pointers are blind – they see entire address space as a collection of objects of their type. Thus, if original pli points to an element of a long integer array with index i, say lia[i], after adding 4 to pli, it will point to an element of the long integer array with index i+4.

In line 9, pi = pi – 6, we decrement 6 from pi. The variable pi is declared as a pointer to int and the sizeof(int)=2 bytes. Thus, this statement will subtract 6x2=12 from the current value of pi, thus the new value of pi is 0x43FC-0x000C = 0x43F0. This way, we have *pi* to point to a location where we originally have the first two characters of *mych*. The equivalent assembly language statement is as follows:

```
SUB.W #12, 0(SP)  ; move back for 6 elements of type int
```

In line 10, we have *pi = a + *pi. The expression *pi indicates an integer variable reached through the pointer pi. The statement says, "add the variable a to the variable the pointer pi is pointing to and store the result of addition back to the variable the pointer pi is pointing to." In our example, we will have the following:

```
 0x3334   ; *pi
+0x0004   ; a
-----------
  0x3338   ; *pi
M[0x43F0] = 0x3338
```

If you want to print the content of mych[0] you will see that it now contains an ASCII code for digit 8. Table 7 shows the updated view of the stack once the statements in lines 8-10 are executed.

How would compiler translate the statement in line 10 into assembly code. The compiler knows where *pi* and *a* are located. The following assembly code will do the work (this is a bit optimized version and the compiler may initially use more assembly instructions):

```
 MOV.W 0(SP), R12    ; R12 get the value of pi
 ADD.W 18(SP), 0(R12);  add a to the variable pi is pointing to
```

Table 7. Variables allocated on the program stack for example in Code 7 after statements in lines 8,9, and 10 are carried out.

Address	Memory[15:0] [hex]	Offset relative to current SP	Variable
0x4400	---		*Original Top of the Stack*
0x43FE	0x0004	18	a
0x43FC	0x0002	16	b
0x43FA	0xFFFF	14	c, upper word
0x43F8	0xFFFC	12	c, lower word

0x43F6	0x0000	10	d, upper word
0x43F4	0x0002	8	d, lower word
0x43F2	0x3132	6	mych[3], mych[2]
0x43F0	0x3338	4	mych[1], mych[0]
0x43EE	0x43F8	2	pli
0x43EC	0x43F0	0 <= SP	pi

`Things to remember 6-1. Pointers and pointer arithmetic.`

Pointers are variables stored in memory that contain addresses of other variables. When we increment a pointer variable, the intent is to update the value of the pointer, so it points to the next variable of the same type in memory. In practice this means that if the size of your variable that pointer is pointing to is 4 bytes, the pointer value after the increment operation will the original value + 4.

7 Exercises

Problem 1. Pointers

Consider the following C program. Assume that the register SP at the beginning points to 0x1100. Answer the following questions. Assume all variables are allocated on the stack, and in the order as they appear in the program (a, b, uli, mych, puli, pc). ASCII code for character ‘0’ is 48 (0x30).

```
int main( void )  {
  volatile unsigned long int uli = 5;
  volatile int a= -4, b= -3;
  volatile char mych[4] = {‘0’, ‘2’, ‘4’, ‘6’};
  volatile unsigned long int *puli = &uli;
  volatile char *pc = mych;
  pc = pc + 2;   //
  *pc = 4 + *pc;
  puli = puli - 1;
  *puli = *puli + uli;
}
```

Fill in the following table with answers to the questions.

#	Question?	Answer

1	The number of bytes allocated on the stack for the variables declared in line 2.	
2	The number of bytes allocated on the stack for the character array declared in line 4.	
3	The number of bytes allocated on the stack for all variables declared in lines 2-6.	
4	Value of mych[2] after initialization performed in line 4.	
5	Address of variable uli (&uli).	
6	Value of pc at the moment after the statement in line 6 is executed.	
7	Value of pc at the moment after the statement in line 7 is executed.	
8	Value of mych[2] at the moment after the statement in line 8 is executed.	
9	Value of puli after the statement in line 9 is executed.	
10	Value of variable b after the statement in line 10 is executed.	