CPE 325: Embedded Systems Laboratory
Laboratory #10 Tutorial
Software Reverse Engineering

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective

Introduce tools and methods for software reverse engineering in embedded systems

Contents

1 Introduction

In this section we will introduce basic concepts, tools, and techniques in software reverse engineering with a special emphasis on embedded computer systems.

Reverse engineering in general is a process of deconstructing man-made artifacts with a goal to reveal their designs and architecture or to extract knowledge. It is widely used in many areas of engineering, but here we are focusing on software reverse engineering. Note: hardware reverse engineering is another topic that may be of interest for electrical and computer engineers, but it is out of scope in this tutorial.

Software reverse engineering refers to a process of analyzing a software system in order to identify its components and their interrelationships and to create representations of the system in another form, typically at a higher level of abstraction. Two main components of software reverse engineering are re-documentation and design recovery. Re-documentation is a process of creating a new representation of the computer code that is easier to understand, often given at a higher level of abstraction. Design recovery is the use of deduction or reasoning from personal experience of the software system to understand its functionality. Software reverse engineering can be used even when the source code is available with the goal to uncover aspects of the program that may be poorly documented or are documented but no longer valid. More often though, software reverse engineering is used when source code is not available.

Software reverse engineering is used for the purpose of:
- Analyzing malware;
- Analyzing closed-source software to uncover vulnerabilities or interoperability issues;
- Analyzing compiler-generated code to validate performance and/or correctness;
- Debugging programs;

This tutorial focuses on reverse engineering of code written for the TI's MSP430 family of microcontrollers. It covers the following topics:
- Format of Executable Files
- GNU binary utilities typically used in software reverse engineering to understand executable files and disassemble executable programs;
- Extracting useful information from binaries;
- Retrieving programs from embedded platforms and analyzing them.

2 Format of Executable Files

In this section we will take a look at the format of executable files. Figure 1 illustrates a generalized flow of source code translation. User created source code written in high-level programming languages or an assembly language is translated into object files that are further linked with library files into executable files that are then loaded onto the target platform.

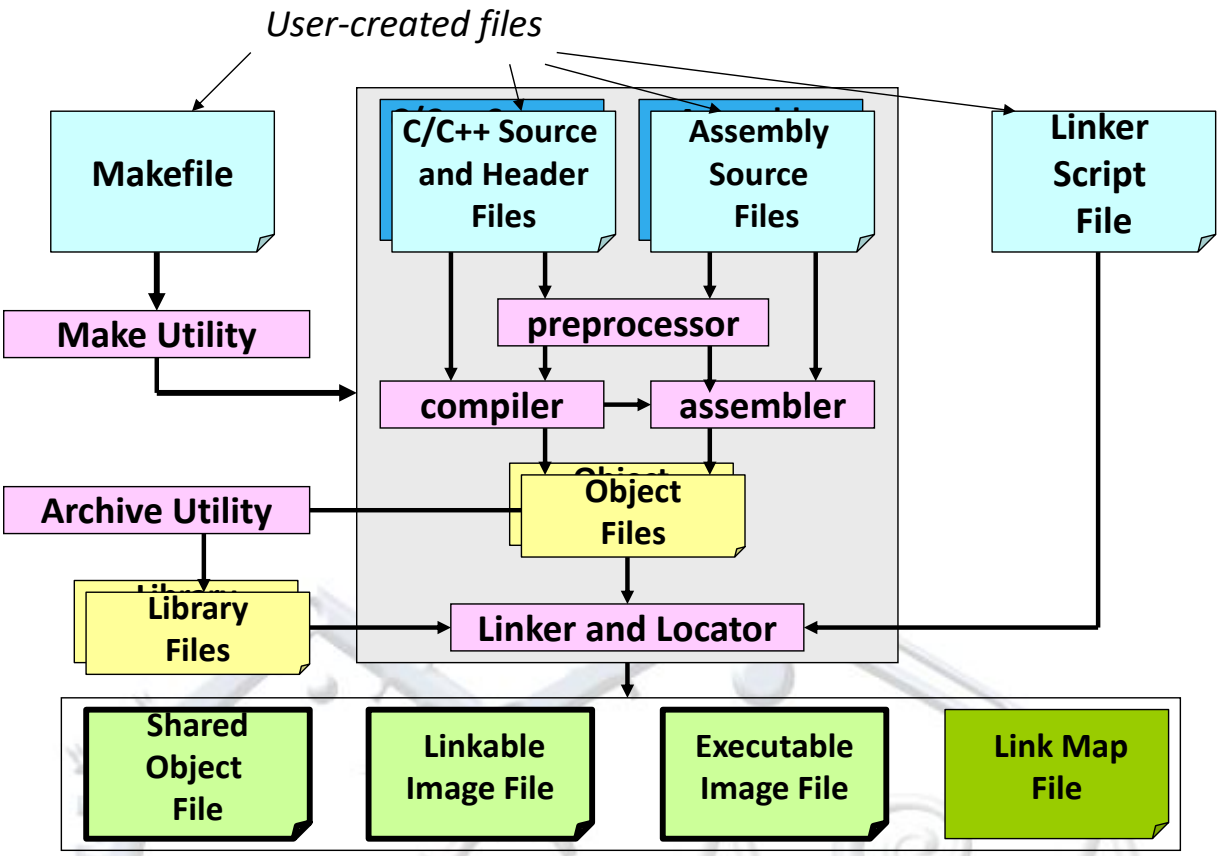


Figure 1. Source Translation Process.

Executable and Linkable File (ELF) format is a common standard file format used for executable files, object files, shared libraries, and core dumps. TI Code Composer Studio produces executable files in the ELF format, regardless of the compiler used (TI compiler or GNU MSP430 GCC compiler). The ELF format is not bound by the Instruction Set Architecture or operating systems. It defines the structure of the file, specifically the headers which describe actual binary content of the file. The structure of the ELF file is well defined and more information can be found at https://en.wikipedia.org/wiki/Executable_and_Linkable_Format. In brief, it is important to recognize the concept of segments and sections. The segments contain information that is needed for run time execution of the program, while sections contain important data needed for linking and relocation.

Figure 2 illustrates two views of ELF files: linkable and executable file formats. ELF files contain the following components:

- ELF file header
- Program header table: Describes zero or more memory segments; It tells loader how to create a process image in memory;
- Section header table: Describes zero or more sections that contain data referred to by entries in the program header tables and section header tables;
- Segments: contain info needed for run time execution;

- Sections: contain info for linking and relocation.

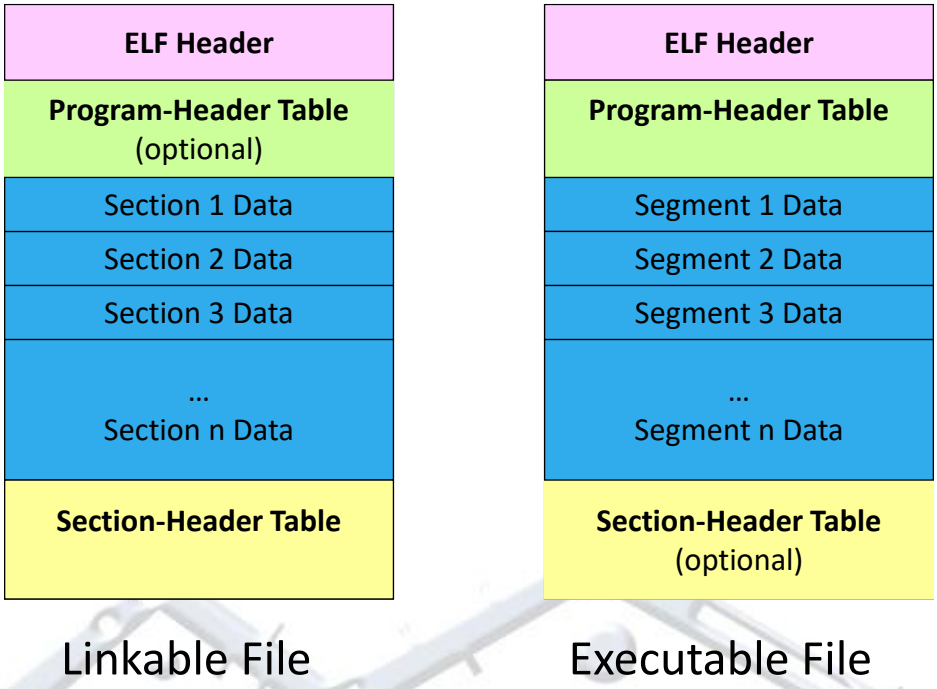


Figure 2. Linkable and Executable Views of ELF Files

ELF linkable files are divided into a collection of sections. Each section contains a single type of information and can contain flags (writable data, memory space during execution or executable machine instructions). Sections have:

- Name and type
- Requested memory location at run time
- Permissions (R, W, X).

Table 1 shows common sections of ELF linkable files.

Table 1. ELF Linking View: Common Sections.

Sections	Description
.interp	Path name of program interpreter
.text	Code (executable instructions) of a program; Typically stored in read-only memory.
.data	Initialized read/write data (global, static)
.bss	Uninitialized read/write data (global, static) Often it is initialized by the start-up code
.const/.rodata	Read-only data; typically stored in Flash memory

.init	Executable instructions for process initialization
.fini	Executable instructions for process termination
.ptl	Holds the procedure linkage table
.re.[x]	Relocation information for section [x]
.dynamic	Dynamic linking information
.symtab, .dynsym	Symbols (static/dynamic)
.strtab, .dynstr	String table
.stack	Stack

Linker is a utility program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file. Figure 3 illustrates linking multiple object files into a single executable file. The linker script defines the memory map of the device with respect to the compiled code sections. The linker needs to know where in memory to locate each of the sections of code based on the type of section and its attributes. Sometimes, these linker scripts can be modified by the developer to add custom sections for very specific purposes, but typically they are provided by software development environments.

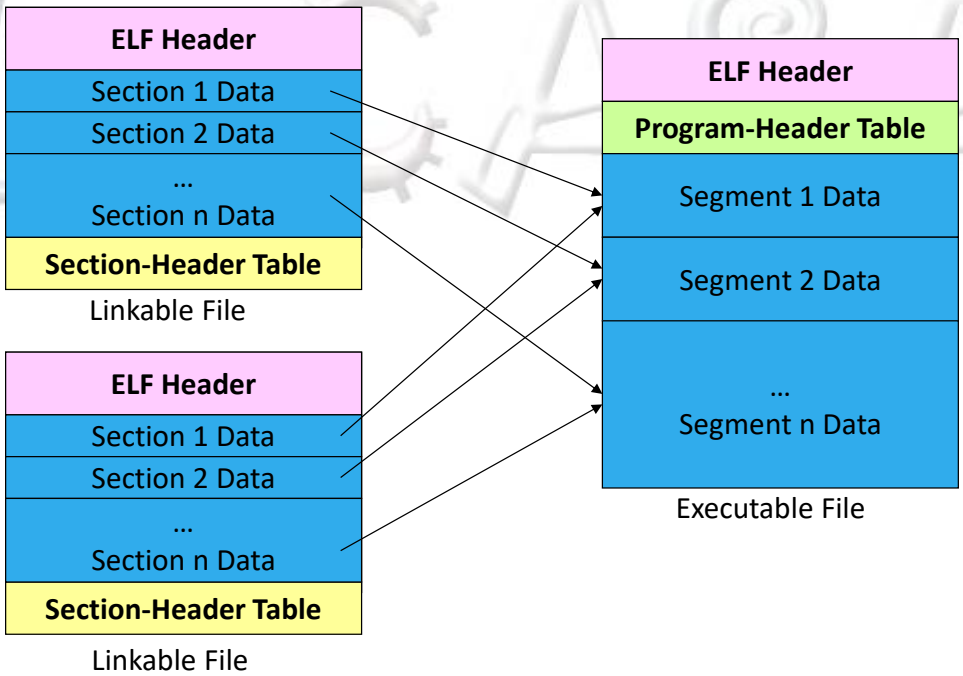


Figure 3. Linking object files into an executable file

The executable ELF file consists of segments. All loadable sections are packed into segments. Segments are parts with code and data that are loaded into memory at run time. Utility

programs that load executable files into memory and start program execution are called loaders. Segments have:

- Type
- Requested memory location
- Permissions (R, W, X)
- Size (in file and in memory)

Table 2 shows common segments in ELF executable files.

Table 2. ELF Executable View: Common Segments.

Common Segments	Description
LOAD	Portion of file to be loaded into memory
INTERP	Pointer to dynamic linker for this executable (.interp section)
DYNAMIC	Pointer to dynamic linking information (.dynamic section)

3 GNU Utilities

In this section we will give a brief introduction to GNU Binary Utilities, also known as binutils. Binutils are a set of programming tools for creating and managing binary programs, object files, profile data, and assembly source code. Table 3 shows a list of commonly used binutils.

Table 3. Common GNU utilities

Utility	Description
as	Assembler
elfedit	Edit ELF files
gdb	Debugger
gprof	Profiler
ld	Linker
objcopy	Copy object files, possibly making changes
objdump	Dump information about object files
nm	List symbols from object files
readelf	Display content of ELF files
strings	List printable strings
size	List total and section sizes

strip	Remove symbols from an object file

Texas Instruments partnered with a third party company to support open-source compiler called MSP430 GCC that originated from a community-driven MSPGCC. MSP430 GCC can be used as a stand-alone package or it can be used within Code Composer Studio (CCS) IDE v6.0 or later as an Add-On through the CCS's App Center.
You can locate various MSP430 GNU utilities from a Windows Command Prompt as shown in Figure 4. To learn more about each utility, run each of them with `--help` switch. Here we will take a closer look at several of these utilities of interest for software reverse engineering tasks:

- msp430-elf-readelf: displays information about executable files (Figure 5);
- msp430-elf-objdump: disassembler (Figure 6);
- msp430-elf-strings: displays printable strings (Figure 7).

NOTE: In newer version of CCS, the default installation path might be changed to:
C:\ti\ccs1010\ccs\tools\compiler\msp430-gcc-9.2.0.50_win64\bin

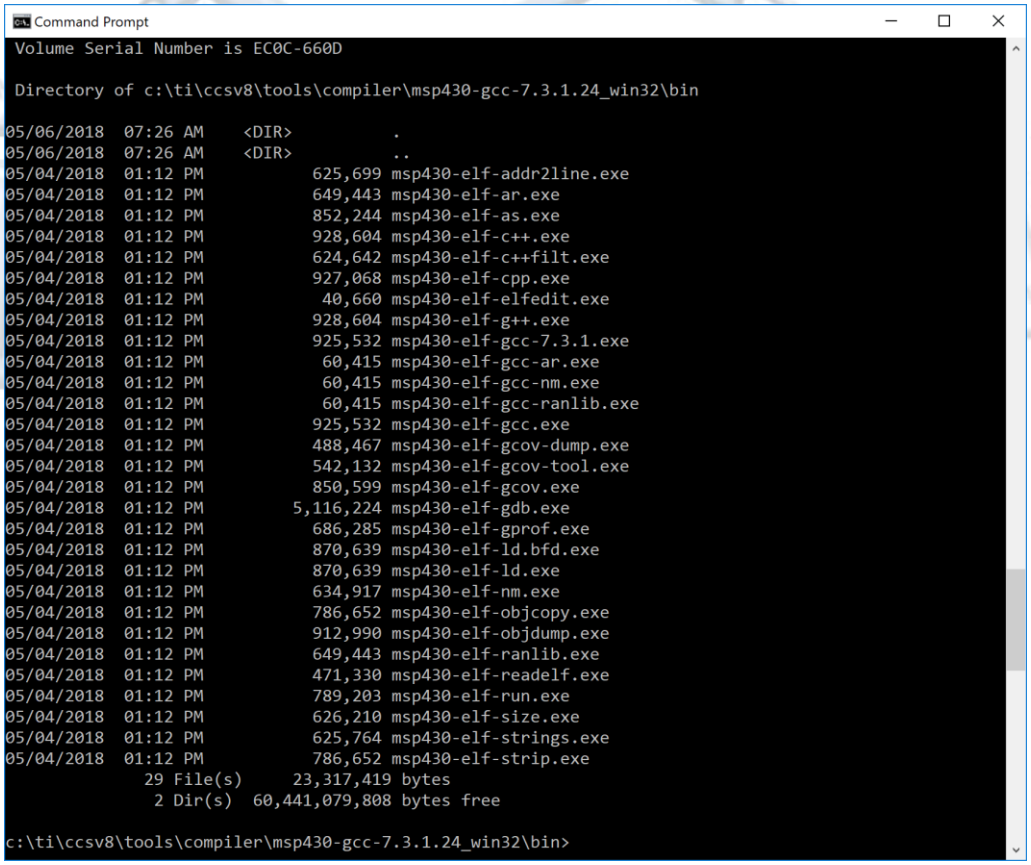


Figure 4. Windows Command Prompt: List of GNU Utilities

```
1  c:\ti\ccsv8\tools\compiler\msp430-gcc-7.3.1.24_win32\bin>msp430-elf-readelf.exe --help
2  Usage: readelf <option(s)> elf-file(s)
```

```
 Display information about the contents of ELF format files
 Options are:
  -a --all               Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header       Display the ELF file header
  -l --program-headers   Display the program headers
     --segments          An alias for --program-headers
  -S --section-headers   Display the sections' header
     --sections          An alias for --section-headers
  -g --section-groups    Display the section groups
  -t --section-details   Display the section details
  -e --headers           Equivalent to: -h -l -S
  -s --syms              Display the symbol table
     --symbols           An alias for --syms
  --dyn-syms             Display the dynamic symbol table
  -n --notes             Display the core notes (if present)
  -r --relocs            Display the relocations (if present)
  -u --unwind            Display the unwind info (if present)
  -d --dynamic           Display the dynamic section (if present)
  -V --version-info      Display the version sections (if present)
  -A --arch-specific     Display architecture specific information (if any)
  -c --archive-index     Display the symbol/file index in an archive
  -D --use-dynamic       Use the dynamic section info when displaying symbols
  -x --hex-dump=<number|name>
                         Dump the contents of section <number|name> as bytes
  -p --string-dump=<number|name>
                         Dump the contents of section <number|name> as strings
  -R --relocated-dump=<number|name>
                         Dump the contents of section <number|name> as relocated bytes
  -z --decompress        Decompress section before dumping it
  -w[lLiaprmfFsoRt] or
  --debug-dump[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
               =frames-interp,=str,=loc,=Ranges,=pubtypes,
               =gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
               =addr,=cu_index]
                         Display the contents of DWARF2 debug sections
  --dwarf-depth=N        Do not display DIEs at depth N or greater
  --dwarf-start=N        Display DIEs starting with N, at the same depth
                         or deeper
  -I --histogram         Display histogram of bucket list lengths
  -W --wide              Allow output width to exceed 80 characters
  @<file>                Read options from <file>
  -H --help              Display this information
  -v --version           Display the version number of readelf
Report bugs to <http://www.sourceware.org/bugzilla/>
```

Figure 5. msp430-elf-readelf Utility: Help System.

```
c:\ti\ccsv8\tools\compiler\msp430-gcc-7.3.1.24_win32\bin>msp430-elf-objdump.exe --help
Usage: msp430-elf-objdump.exe <option(s)> <file(s)>
 Display information from object <file(s)>.
 At least one of the following switches must be given:
  -a, --archive-headers    Display archive header information
  -f, --file-headers       Display the contents of the overall file header
  -p, --private-headers    Display object format specific file header contents
  -P, --private=OPT,OPT... Display object format specific contents
  -h, --[section-]headers  Display the contents of the section headers
  -x, --all-headers        Display the contents of all headers
  -d, --disassemble        Display assembler contents of executable sections
  -D, --disassemble-all    Display assembler contents of all sections
```

```
  -S, --source            Intermix source code with disassembly
  -s, --full-contents     Display the full contents of all sections requested
  -g, --debugging         Display debug information in object file
  -e, --debugging-tags    Display debug information using ctags style
  -G, --stabs             Display (in raw form) any STABS info in the file
  -W[lLiaprmfFsoRt] or
  --dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
          =frames-interp,=str,=loc,=Ranges,=pubtypes,
          =gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
          =addr,=cu_index]
                          Display DWARF info in the file
  -t, --syms              Display the contents of the symbol table(s)
  -T, --dynamic-syms      Display the contents of the dynamic symbol table
  -r, --reloc             Display the relocation entries in the file
  -R, --dynamic-reloc     Display the dynamic relocation entries in the file
  @<file>                 Read options from <file>
  -v, --version           Display this program's version number
  -i, --info              List object formats and architectures supported
  -H, --help              Display this information

 The following switches are optional:
  -b, --target=BFDNAME           Specify the target object format as BFDNAME
  -m, --architecture=MACHINE     Specify the target architecture as MACHINE
  -j, --section=NAME             Only display information for section NAME
  -M, --disassembler-options=OPT Pass text OPT on to the disassembler
  -EB --endian=big               Assume big endian format when disassembling
  -EL --endian=little            Assume little endian format when disassembling
      --file-start-context       Include context from start of file (with -S)
  -I, --include=DIR              Add DIR to search list for source files
  -l, --line-numbers             Include line numbers and filenames in output
  -F, --file-offsets             Include file offsets when displaying information
  -C, --demangle[=STYLE]         Decode mangled/processed symbol names
                                  The STYLE, if specified, can be `auto', `gnu',
                                  `lucid', `arm', `hp', `edg', `gnu-v3', `java'
                                  or `gnat'
  -w, --wide                     Format output for more than 80 columns
  -z, --disassemble-zeroes       Do not skip blocks of zeroes when disassembling
      --start-address=ADDR       Only process data whose address is >= ADDR
      --stop-address=ADDR        Only process data whose address is <= ADDR
      --prefix-addresses         Print complete address alongside disassembly
      --[no-]show-raw-insn       Display hex alongside symbolic disassembly
      --insn-width=WIDTH         Display WIDTH bytes on a single line for -d
      --adjust-vma=OFFSET        Add OFFSET to all displayed section addresses
      --special-syms             Include special symbols in symbol dumps
      --prefix=PREFIX            Add PREFIX to absolute paths for -S
      --prefix-strip=LEVEL       Strip initial directory names for -S
      --dwarf-depth=N        Do not display DIEs at depth N or greater
      --dwarf-start=N        Display DIEs starting with N, at the same depth
                             or deeper
      --dwarf-check          Make additional dwarf internal consistency checks.

msp430-elf-objdump.exe: supported targets: elf32-msp430 elf32-msp430 elf32-little elf32-big
plugin srec symbolsrec verilog tekhex binary ihex
msp430-elf-objdump.exe: supported architectures: msp:14 MSP430 MSP430x11x1 MSP430x12 MSP430x13
MSP430x14 MSP430x15 MSP430x16 MSP430x20 MSP430x21 MSP430x22 MSP430x23 MSP430x24 MSP430x26
MSP430x31 MSP430x32 MSP430x33 MSP430x41 MSP430x42 MSP430x43 MSP430x44 MSP430x46 MSP430x47
MSP430x54 MSP430X plugin
Report bugs to <http://www.sourceware.org/bugzilla/>.
```

Figure 6. msp430-elf-objdump Utility: Help System.

```
c:\ti\ccsv8\tools\compiler\msp430-gcc-7.3.1.24_win32\bin>msp430-elf-strings.exe --help
Usage: msp430-elf-strings.exe [option(s)] [file(s)]
 Display printable strings in [file(s)] (stdin by default)
 The options are:
  -a - --all               Scan the entire file, not just the data section [default]
  -d --data                Only scan the data sections in the file
  -f --print-file-name     Print the name of the file before each string
  -n --bytes=[number]      Locate & print any NUL-terminated sequence of at
  -<number>                  least [number] characters (default 4).
  -t --radix={o,d,x}       Print the location of the string in base 8, 10 or 16
  -w --include-all-whitespace Include all whitespace as valid string characters
  -o                       An alias for --radix=o
  -T --target=<BFDNAME>    Specify the binary file format
  -e --encoding={s,S,b,l,B,L} Select character size and endianness:
                           s = 7-bit, S = 8-bit, {b,l} = 16-bit, {B,L} = 32-bit
  -s --output-separator=<string> String used to separate strings in output.
  @<file>                  Read options from <file>
  -h --help                Display this information
  -v -V --version          Print the program's version number
msp430-elf-strings.exe: supported targets: elf32-msp430 elf32-msp430 elf32-little elf32-big
plugin srec symbolsrec verilog tekhex binary ihex
Report bugs to <http://www.sourceware.org/bugzilla/>
```

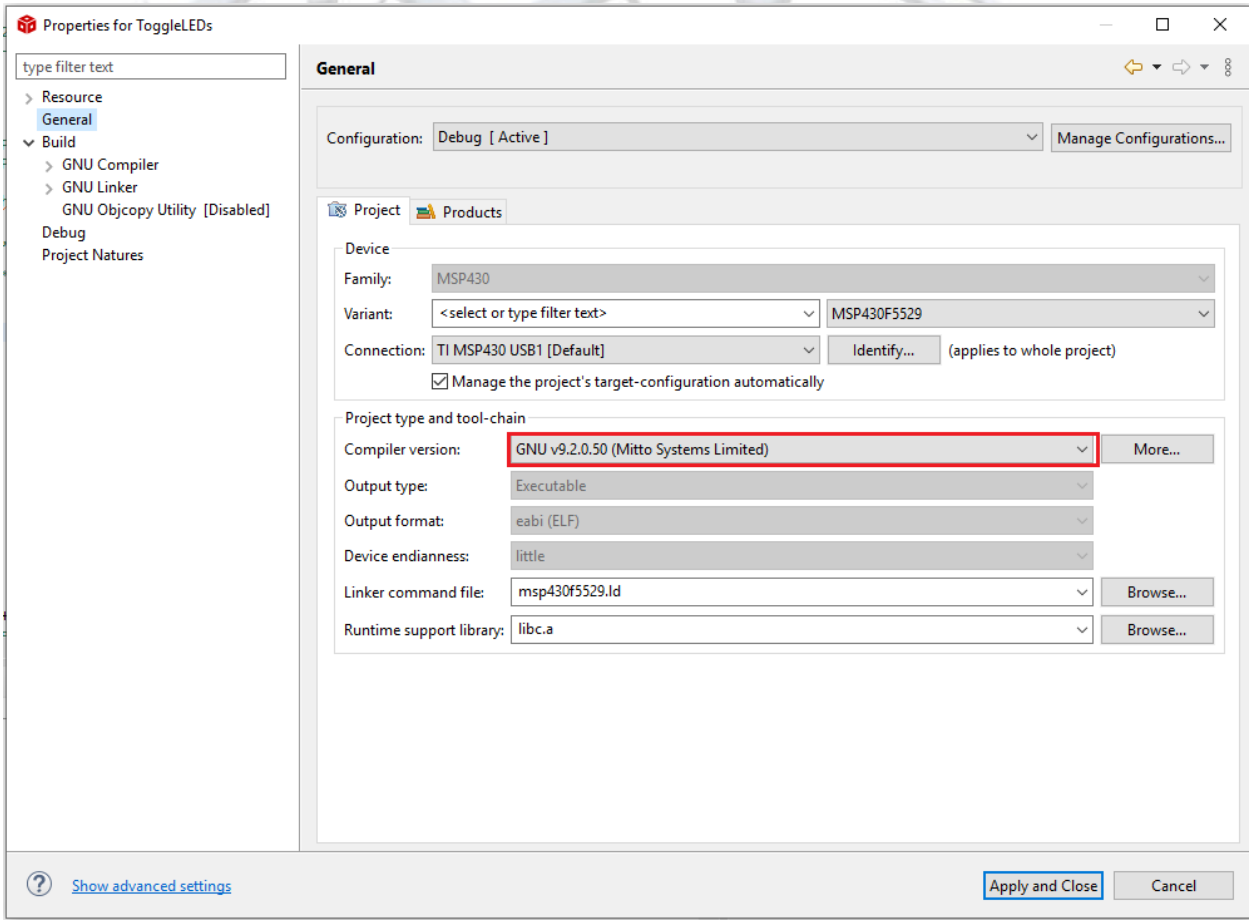
Figure 7. msp430-elf-strings Utility: Help System.

4 Deconstructing Executable Files: An Example

To demonstrate software engineering in practice, let us start from a C program described in Figure 8. This program toggles the LEDs connected to ports P2.1 and P2.2 on the TI Experimenter's board. Our first step is to compile this program using GNU C compiler that comes as an Add-on in TI's Code Composer Studio. To compile this program, we select the GNU C compiler and set appropriate compilation flags as shown in Figure 9.

```
/*******************************************************************************
 *   File:        ToggleLEDs.c
 *   Description: Program toggles LED1 and LED2 by
 *                xoring port pins inside of an infinite loop.
 *
 *   Board:       MSP-EXP430F5529 Experimenter Board
 *   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO
 *
 *                 MSP430F5529
 *             -----------------
 *         /|\|                 |
 *          | |                 |
 *          --|RST              |
 *            |                 |
 *            |             P1.0|--> LED1
 *            |             P4.7|--> LED2
 *
 *   Author: Alex Milenkovich, milenkovic@computer.org
 *   Date:   September 2010
 *   Modified: Prawar Poudel, prawar.poudel@uah.edu
 *   Date:   November 2020
```

```
22  ********************************************************************************/
23  #include  <msp430.h>
24
25  int main(void) {
26     WDTCTL = WDTPW + WDTHOLD;     // Stop watchdog timer
27
28     P1DIR |= BIT0;                // Set P1.0 to output direction (xxxx_xxx1) for LED1
29     P4DIR |= BIT7;                // Set P4.7 to output direction (1xxx_xxxx) for LED2
30
31     P1OUT |= BIT0;                // Set P1.0 ON (xxxx_xxx1) for LED1
32     P4OUT &= ~BIT7;               // Set P4.7 OFF (0xxx_xxxx) for LED2
33
34     for (;;) {
35       unsigned int i;
36
37       P1OUT ^= BIT0;              // toggle the LED1
38       P4OUT ^= BIT7;              // toggle the LED2
39
40       for(i = 0; i < 50000; i++); // Software delay (13 cc per iteration)
41       /* Total delay on average 13 cc*50,000 = 750,000; 750,000 * 1us = 0.75 s */
42     }
43     return 0;
44  }
45
```



46

Figure 8. ToggleLEDs Source Code.

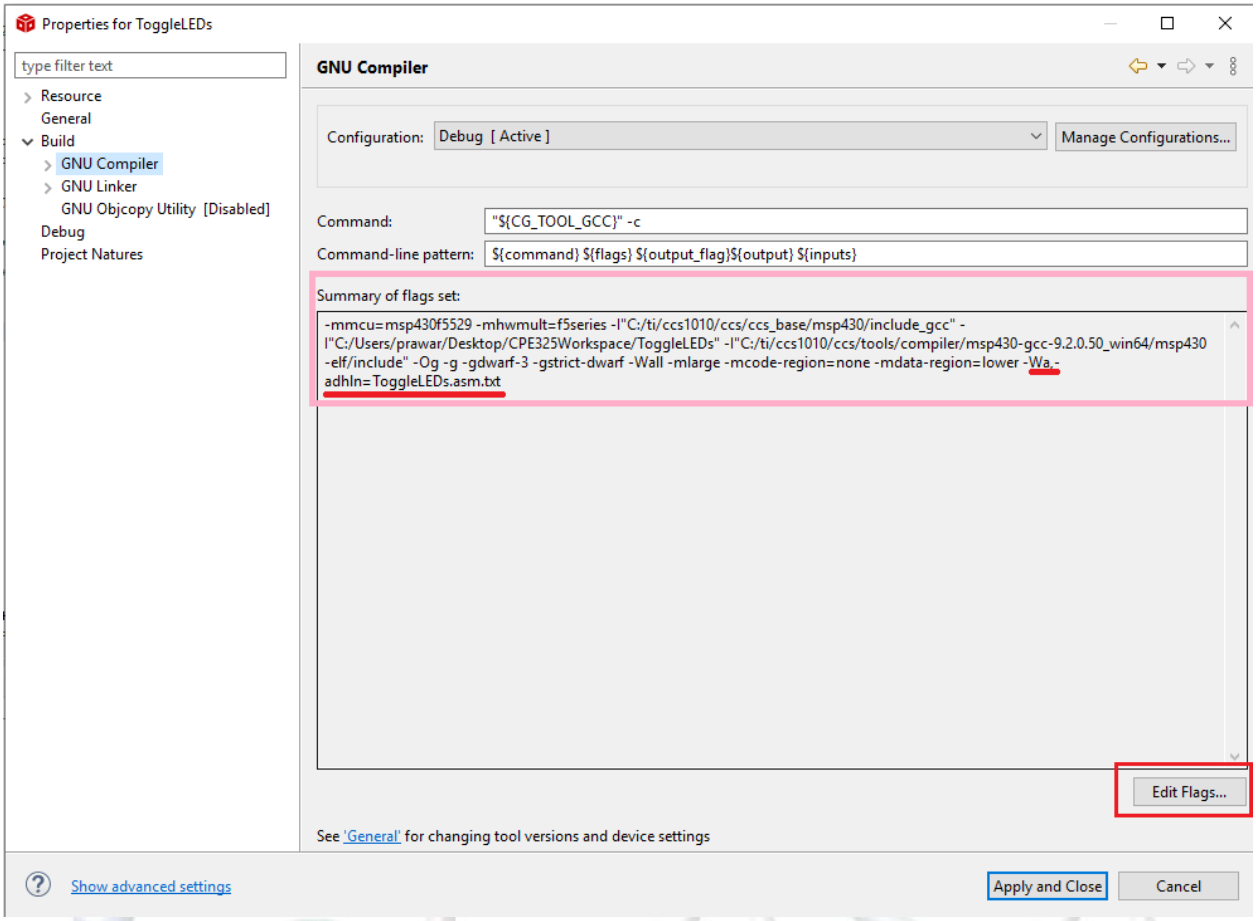


Figure 9. Settings for GNU C Compiler.

As the result of compilation you will notice ToggleLEDs.o (object file), ToggleLEDs.out (executable file), and ToggleLEDs.asm.txt (assembly code created by the compiler switches –Wa,-adhln=ToggleLEDs.asm.txt). You can click on the "Edit Flags" option in Figure 9 to add the flag to create assembly code manually.

Figure 11 shows the output list file "*ToggleLEDs.asm.txt*" that illustrates assembly code for each line of the source code in C. Figure 10 shows how to locate the file in *Project Explorer* window.

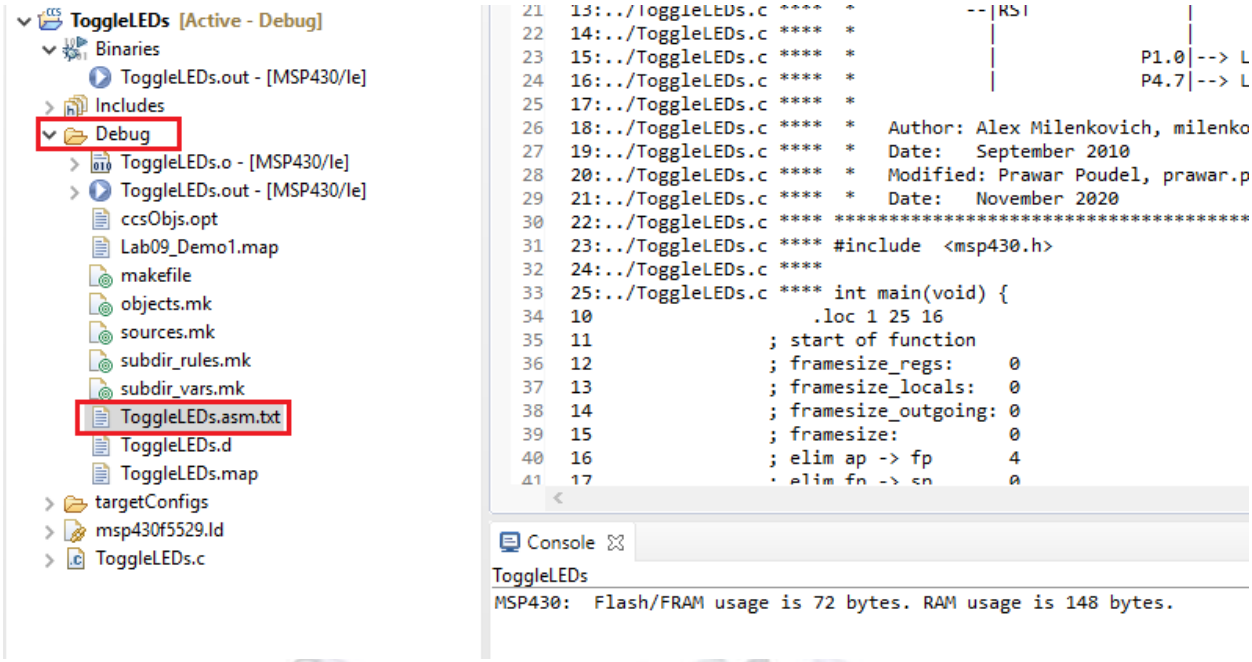


Figure 10. Locating output list files in Project Explorer window in CCS

```
     1                      .file   "ToggleLEDs.c"
     2                  .text
     3                  .Ltext0:
     4                         .balign 2
     5                         .global main
     7                  main:
     8                  .LFB0:
     9                         .file 1 "../ToggleLEDs.c"
     1:../ToggleLEDs.c ****
/*****************************************************************************
     2:../ToggleLEDs.c ****  *   File:       ToggleLEDs.c
     3:../ToggleLEDs.c ****  *   Description: Program toggles LED1 and LED2 by
     4:../ToggleLEDs.c ****  *                xoring port pins inside of an infinite loop.
     5:../ToggleLEDs.c ****  *
     6:../ToggleLEDs.c ****  *   Board:       MSP-EXP430F5529 Experimenter Board
     7:../ToggleLEDs.c ****  *   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO
     8:../ToggleLEDs.c ****  *
     9:../ToggleLEDs.c ****  *                 MSP430F5529
    10:../ToggleLEDs.c ****  *             -----------------
    11:../ToggleLEDs.c ****  *          /|\|                 |
    12:../ToggleLEDs.c ****  *           | |                 |
    13:../ToggleLEDs.c ****  *           --|RST              |
    14:../ToggleLEDs.c ****  *             |                 |
    15:../ToggleLEDs.c ****  *             |             P1.0|--> LED1
    16:../ToggleLEDs.c ****  *             |             P4.7|--> LED2
    17:../ToggleLEDs.c ****  *
    18:../ToggleLEDs.c ****  *   Author: Alex Milenkovich, milenkovic@computer.org
    19:../ToggleLEDs.c ****  *   Date:   September 2010
    20:../ToggleLEDs.c ****  *   Modified: Prawar Poudel, prawar.poudel@uah.edu
    21:../ToggleLEDs.c ****  *   Date:   November 2020
    22:../ToggleLEDs.c ****
*****************************************************************************/
    23:../ToggleLEDs.c **** #include  <msp430.h>
    24:../ToggleLEDs.c ****
    25:../ToggleLEDs.c **** int main(void) {
```

```
  10                            .loc 1 25 16
  11                    ; start of function
  12                    ; framesize_regs:     0
  13                    ; framesize_locals:   0
  14                    ; framesize_outgoing: 0
  15                    ; framesize:          0
  16                    ; elim ap -> fp       4
  17                    ; elim fp -> sp       0
  18                    ; saved regs:(none)
  19                            ; start of prologue
  20                            ; end of prologue
  26:../ToggleLEDs.c ****    WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
  21                            .loc 1 26 4
  22                            .loc 1 26 11 is_stmt 0
  23 0000 B240 805A             MOV.W  #23168, &WDTCTL
  23      0000
  27:../ToggleLEDs.c ****
  28:../ToggleLEDs.c ****    P1DIR |= BIT0;               // Set P1.0 to output direction
(xxxx_xxx1) for LED1
  24                            .loc 1 28 4 is_stmt 1
  25                            .loc 1 28 10 is_stmt 0
  26 0006 D2D3 0000             BIS.B  #1, &PADIR_L
  29:../ToggleLEDs.c ****    P4DIR |= BIT7;               // Set P4.7 to output direction
(1xxx_xxxx) for LED2
  27                            .loc 1 29 4 is_stmt 1
  28                            .loc 1 29 10 is_stmt 0
  29 000a F2D0 80FF             BIS.B  #-128, &PBDIR_H
  29      0000
  30:../ToggleLEDs.c ****
  31:../ToggleLEDs.c ****    P1OUT |= BIT0;               // Set P1.0 ON (xxxx_xxx1) for LED1
  30                            .loc 1 31 4 is_stmt 1
  31                            .loc 1 31 10 is_stmt 0
  32 0010 D2D3 0000             BIS.B  #1, &PAOUT_L
  32:../ToggleLEDs.c ****    P4OUT &= ~BIT7;              // Set P4.7 OFF (0xxx_xxxx) for LED2
  33                            .loc 1 32 4 is_stmt 1
  34                            .loc 1 32 10 is_stmt 0
  35 0014 F2F0 7F00             AND.B  #127, &PBOUT_H
  35      0000
  36 001a 8000 0000             BRA    #.L4
  37                    .LVL0:
  38                    .L3:
  39                    .LBB2:
  33:../ToggleLEDs.c ****
  34:../ToggleLEDs.c ****    for (;;) {
  35:../ToggleLEDs.c ****      unsigned int i;
  36:../ToggleLEDs.c ****
  37:../ToggleLEDs.c ****      P1OUT ^= BIT0;             // toggle the LED1
  38:../ToggleLEDs.c ****      P4OUT ^= BIT7;             // toggle the LED2
  39:../ToggleLEDs.c ****
  40:../ToggleLEDs.c ****      for(i = 0; i < 50000; i++); // Software delay (13 cc per
iteration)
  40                            .loc 1 40 32 is_stmt 1
  41                            .loc 1 40 28
  42                            .loc 1 40 29 is_stmt 0
  43 001e 1C53                  ADD.W  #1, R12
  44                    .LVL1:
  45                    .L2:
  46                            .loc 1 40 17 is_stmt 1
  47                            .loc 1 40 6 is_stmt 0
  48 0020 3D40 4FC3             MOV.W  #-15537, R13
  49 0024 0D9C 002C             CMP.W  R12, R13 { JHS .L3
```

```
  50                 .LVL2:
  51                 .L4:
  52                 .LBE2:
  34:../ToggleLEDs.c ****      unsigned int i;
  53                         .loc 1 34 4 is_stmt 1
  54                 .LBB3:
  35:../ToggleLEDs.c ****
  55                         .loc 1 35 6
  37:../ToggleLEDs.c ****      P4OUT ^= BIT7;             // toggle the LED2
  56                         .loc 1 37 6
  37:../ToggleLEDs.c ****      P4OUT ^= BIT7;             // toggle the LED2
  57                         .loc 1 37 12 is_stmt 0
  58 0028 D2E3 0000          XOR.B  #1, &PAOUT_L
  38:../ToggleLEDs.c ****
  59                         .loc 1 38 6 is_stmt 1
  38:../ToggleLEDs.c ****
  60                         .loc 1 38 12 is_stmt 0
  61 002c F250 80FF          ADD.B  #-128, &PBOUT_H
  61      0000
  62                         .loc 1 40 6 is_stmt 1
  63                 .LVL3:
  64                         .loc 1 40 12 is_stmt 0
  65 0032 4C43               MOV.B  #0, R12
  66                         .loc 1 40 6
  67 0034 8000 0000          BRA    #.L2
  68                 .LBE3:
  69                 .LFE0:
  97                 .Letext0:
  98                         .file 2
"C:/ti/ccs1010/ccs/ccs_base/msp430/include_gcc/msp430f5529.h"
```

Figure 11. Output Assembly Code Generated by GNU GCC Compiler

For the moment, let us assume that we are given the executable file and that we have no prior knowledge what that executable code is doing. Here we will demonstrate steps we can take to deconstruct or reverse engineer code from the executable file.

Step #1: **Examine ELF header to determine type of machine code, data representation, entry points and more**. We can use msp430-elf-readelf to learn more about the executable file. Switch *--file-header* displays information about the ELF header: this is an ELF32 executable file, containing code for MSP430 microcontroller, the entry program point is at address 0x4400, and so on (see Figure 12). In the following figure, the highlighted test is the command that is executed in command line or PowerShell in windows, while contents from Line 3 and onwards are the outputs generated.

```
1  C:\ti\ccs1010\ccs\tools\compiler\msp430-gcc-9.2.0.50_win64\bin> msp430-elf-readelf.exe -h
2  ToggleLEDs.out
3  ELF Header:
4    Magic:   7f 45 4c 46 01 01 01 ff 00 00 00 00 00 00 00 00
5    Class:                             ELF32
6    Data:                              2's complement, little endian
7    Version:                           1 (current)
8    OS/ABI:                            Standalone App
9    ABI Version:                       0
10   Type:                              EXEC (Executable file)
11   Machine:                           Texas Instruments msp430 microcontroller
12   Version:                           0x1
```

```
  Entry point address:               0x4400
  Start of program headers:          52 (bytes into file)
  Start of section headers:          21604 (bytes into file)
  Flags:                             0x2d: architecture variant: MSP430X
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         26
  Section header string table index: 25
```

Figure 12. msp430-elf-readelf –file-header (-h): ELF Header Content for ToggleLEDs.out

Step #2. **Examine ELF file sections**.

We can use msp430-elf-readelf utility with --section-headers switch to display information about all sections. Figure 13 shows the output of this command for ToggleLEDs.out. A similar information can be obtained using objdump utility with –h switch as shown in Figure 15. The list of sections includes the section name, the starting addresses (VMA – virtual and LMA – load memory address), the offset of the section in the actual file, the size of the section, the section attributes, and the alignment in memory.

The __reset_vector, .rodata, and .text sections reside in the Flash memory (read only). The .lowtext starts at 0x4400 and has the capacity of 0xa bytes (10 bytes). It is followed by the .text section that starts at 0x440a and contain 0x38 bytes. The RAM memory region consists of .data and .bss sections. The .bss section starts at address 0x2400, followed by the .heap section that also starts at 0x2400 since our .bss is empty. (Figure 14 shows another case where .bss is not empty, and the memory addresses are different. Do you think Figure 14 belongs to the same microcontroller as in Figure 13?) Another noteworthy entry is __reset_vector that sits at the address 0xFFFE.

```

C:\ti\ccs1010\ccs\tools\compiler\msp430-gcc-9.2.0.50_win64\bin> msp430-elf-readelf.exe --
section-headers ToggleLEDs.out
There are 26 section headers, starting at offset 0x5464:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] __reset_vector    PROGBITS        0000fffe 0000d6 000002 00   A  0   0  1
  [ 2] .lower.rodata     PROGBITS        00004400 0000d8 000000 00   W  0   0  1
  [ 3] .rodata           PROGBITS        00004400 0000d8 000000 00  WA  0   0  1
  [ 4] .rodata2          PROGBITS        00004400 0000d8 000000 00   W  0   0  1
  [ 5] .data             PROGBITS        00002400 0000d8 000000 00  WA  0   0  1
  [ 6] .bss              NOBITS          00002400 000000 000000 00  WA  0   0  1
  [ 7] .noinit           PROGBITS        00002400 0000d8 000000 00   W  0   0  1
  [ 8] .heap             NOBITS          00002400 000094 000004 00  WA  0   0  1
  [ 9] .lowtext          PROGBITS        00004400 000094 00000a 00  AX  0   0  1
  [10] .lower.text       PROGBITS        0000440a 0000d8 000000 00   W  0   0  1
  [11] .text             PROGBITS        0000440a 00009e 000038 00  AX  0   0  2
  [12] .upper.text       PROGBITS        00010000 0000d8 000000 00   W  0   0  1
  [13] .MSP430.attribute MSP430_ATTRIBUT 00000000 0000d8 000026 00      0   0  1
  [14] .comment          PROGBITS        00000000 0000fe 000039 01  MS  0   0  1
  [15] .debug_aranges    PROGBITS        00000000 000137 000020 00      0   0  1
  [16] .debug_info       PROGBITS        00000000 000157 002af6 00      0   0  1
  [17] .debug_abbrev     PROGBITS        00000000 002c4d 00009d 00      0   0  1
  [18] .debug_line       PROGBITS        00000000 002cea 000152 00      0   0  1
  [19] .debug_frame      PROGBITS        00000000 002e3c 000024 00      0   0  4
```

```
  [20] .debug_str        PROGBITS        00000000 002e60 001c38 01  MS  0   0  1
  [21] .debug_loc        PROGBITS        00000000 004a98 000013 00      0   0  1
  [22] .debug_ranges     PROGBITS        00000000 004aab 000018 00      0   0  1
  [23] .symtab           SYMTAB          00000000 004ac4 000620 10     24  77  4
  [24] .strtab           STRTAB          00000000 0050e4 000280 00      0   0  1
  [25] .shstrtab         STRTAB          00000000 005364 0000fd 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

Figure 13. msp430-elf-readelf –section-headers (-S): ELF section headers for ToggleLEDs.out (I)

```
C:\Users\milenka\workspace_cpe325\ToggleLEDs\Debug__GNU>msp430-elf-readelf --section-headers
ToggleLEDs.out
There are 25 section headers, starting at offset 0x3278:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] __reset_vector    PROGBITS        0000fffe 00028e 000002 00   A  0   0  1
  [ 2] .lower.rodata     PROGBITS        00003100 000290 000000 00   W  0   0  1
  [ 3] .rodata           PROGBITS        00003100 000290 000000 00  WA  0   0  1
  [ 4] .rodata2          PROGBITS        00003100 0000d4 00000c 00  WA  0   0  4
  [ 5] .data             PROGBITS        00001100 000290 000000 00  WA  0   0  1
  [ 6] .bss              NOBITS          00001100 0000e0 000012 00  WA  0   0  2
  [ 7] .noinit           PROGBITS        00001112 000290 000000 00   W  0   0  1
  [ 8] .heap             NOBITS          00001112 0000e2 000004 00  WA  0   0  1
  [ 9] .lowtext          PROGBITS        0000310c 0000e0 000066 00  AX  0   0  1
  [10] .lower.text       PROGBITS        00003172 000290 000000 00   W  0   0  1
  [11] .text             PROGBITS        00003172 000146 000146 00  AX  0   0  2
  [12] .upper.text       PROGBITS        00010000 000290 000000 00   W  0   0  1
  [13] .MSP430.attribute MSP430_ATTRIBUT 00000000 000290 000017 00      0   0  1
  [14] .comment          PROGBITS        00000000 0002a7 000039 01  MS  0   0  1
  [15] .debug_aranges    PROGBITS        00000000 0002e0 000020 00      0   0  1
  [16] .debug_info       PROGBITS        00000000 000300 000d42 00      0   0  1
  [17] .debug_abbrev     PROGBITS        00000000 001042 0000a6 00      0   0  1
  [18] .debug_line       PROGBITS        00000000 0010e8 0000a5 00      0   0  1
  [19] .debug_frame      PROGBITS        00000000 001190 000024 00      0   0  4
  [20] .debug_str        PROGBITS        00000000 0011b4 0007f8 01  MS  0   0  1
  [21] .debug_loc        PROGBITS        00000000 0019ac 000013 00      0   0  1
  [22] .shstrtab         STRTAB          00000000 003187 0000ef 00      0   0  1
  [23] .symtab           SYMTAB          00000000 0019c0 000f20 10     24 196  4
  [24] .strtab           STRTAB          00000000 0028e0 0008a7 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Figure 14. msp430-elf-readelf –section-headers (-S): ELF section headers for ToggleLEDs.out (II)

```
C:\ti\ccs1010\ccs\tools\compiler\msp430-gcc-9.2.0.50_win64\bin> msp430-elf-objdump.exe -h
ToggleLEDs.out

C:\Users\prawar\Desktop\CPE325Workspace\ToggleLEDs\Debug\ToggleLEDs.out:     file format
elf32-msp430

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
```

```
  0 __reset_vector 00000002  0000fffe  0000fffe  000000d6  2**0
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .lower.rodata 00000000  00004400  00004400  000000d8  2**0
                   CONTENTS
  2 .rodata       00000000  00004400  00004400  000000d8  2**0
                   CONTENTS, ALLOC, LOAD, DATA
  3 .rodata2      00000000  00004400  00004400  000000d8  2**0
                   CONTENTS
  4 .data         00000000  00002400  00002400  000000d8  2**0
                   CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000000  00002400  00004400  00000000  2**0
                   ALLOC
  6 .noinit       00000000  00002400  00002400  000000d8  2**0
                   CONTENTS
  7 .heap         00000004  00002400  00004400  00000094  2**0
                   ALLOC
  8 .lowtext      0000000a  00004400  00004400  00000094  2**0
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  9 .lower.text   00000000  0000440a  0000440a  000000d8  2**0
                   CONTENTS
 10 .text         00000038  0000440a  0000440a  0000009e  2**1
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .upper.text   00000000  00010000  00010000  000000d8  2**0
                   CONTENTS
 12 .MSP430.attributes 00000026  00000000  00000000  000000d8  2**0
                   CONTENTS, READONLY
 13 .comment      00000039  00000000  00000000  000000fe  2**0
                   CONTENTS, READONLY
 14 .debug_aranges 00000020  00000000  00000000  00000137  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 15 .debug_info   00002af6  00000000  00000000  00000157  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 16 .debug_abbrev 0000009d  00000000  00000000  00002c4d  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 17 .debug_line   00000152  00000000  00000000  00002cea  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 18 .debug_frame  00000024  00000000  00000000  00002e3c  2**2
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 19 .debug_str    00001c38  00000000  00000000  00002e60  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 20 .debug_loc    00000013  00000000  00000000  00004a98  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 21 .debug_ranges 00000018  00000000  00000000  00004aab  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
```

Figure 15. msp430-elf-objdump –h: ELF section headers for ToggleLEDs.out

Step #3. **Display ELF symbols**.

We use msp430-elf-readelf utility with --symbols switch (or –s) to display all symbols in the ELF file. Figure 16 shows a filtered output of this utility for ToggleLEDs.out (the full list contains 98 symbols). A similar output can be obtained by using msp430-elf-objdump utility with switch –t or by using a separate binutils utility msp430-elf-nm. By searching the output you can identify important symbols such as '_start', '__stack', '__heap_start__', '_bssstart', 'main', and others. These sections and their locations are defined in the linker script file for the given microcontroller as the placement is a function of the size and mapping of Flash and RAM memory.

```
C:\ti\ccs1010\ccs\tools\compiler\msp430-gcc-9.2.0.50_win64\bin> msp430-elf-readelf.exe --
symbols ToggleLEDs.out

Symbol table '.symtab' contains 98 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000fffe     0 SECTION LOCAL  DEFAULT    1
     2: 00004400     0 SECTION LOCAL  DEFAULT    2
     3: 00004400     0 SECTION LOCAL  DEFAULT    3
     4: 00004400     0 SECTION LOCAL  DEFAULT    4
     5: 00002400     0 SECTION LOCAL  DEFAULT    5
     6: 00002400     0 SECTION LOCAL  DEFAULT    6
            ….
            ….
    75: 00004432     0 NOTYPE  LOCAL  DEFAULT   11 .LBB3
    76: 00004442     0 NOTYPE  LOCAL  DEFAULT   11 .LBE3
    77: 0000015c     0 NOTYPE  GLOBAL DEFAULT  ABS WDTCTL
    78: 00004400     4 FUNC    GLOBAL DEFAULT    9 __crt0_start
    79: 00002404     0 NOTYPE  GLOBAL DEFAULT    8 __HeapLimit
    80: 00002404     0 NOTYPE  GLOBAL DEFAULT    8 __heap_end__
    81: 00000202     0 NOTYPE  GLOBAL DEFAULT  ABS PAOUT_L
    82: 00004400     0 NOTYPE  GLOBAL DEFAULT    9 _start
    83: 00000204     0 NOTYPE  GLOBAL DEFAULT  ABS PADIR_L
    84: 00000000     0 NOTYPE  WEAK   DEFAULT  ABS __rom_highdatacopysize
    85: 0000440a    56 FUNC    GLOBAL DEFAULT   11 main
    86: 00002400     0 NOTYPE  GLOBAL DEFAULT    8 __heap_start__
    87: 00000000     0 NOTYPE  WEAK   DEFAULT  ABS __high_bsssize
    88: 00000000     0 NOTYPE  WEAK   DEFAULT  ABS __rom_highdatastart
    89: 00000223     0 NOTYPE  GLOBAL DEFAULT  ABS PBOUT_H
    90: 00000000     0 NOTYPE  WEAK   DEFAULT  ABS __high_datastart
    91: 00000000     0 NOTYPE  WEAK   DEFAULT  ABS __upper_data_init
    92: 00004400     0 NOTYPE  GLOBAL DEFAULT    8 __stack
    93: 00002400     0 NOTYPE  GLOBAL DEFAULT    5 _edata
    94: 00002400     0 NOTYPE  GLOBAL DEFAULT    8 _end
    95: 00000000     0 NOTYPE  WEAK   DEFAULT  ABS __high_bssstart
    96: 00004404     6 FUNC    GLOBAL DEFAULT    9 __crt0_call_main
    97: 00000225     0 NOTYPE  GLOBAL DEFAULT  ABS PBDIR_H
```

Figure 16. msp430-elf-readelf --symbols: ELF symbols for ToggleLEDs.out

Step #4. **Display ELF segments**.

We use msp430-elf-readelf utility with --program-headers switch (or –segments) to display all segments that are loadable into the memory. Figure 17 shows the output of this utility for ToggleLEDs.out. It shows information about loadable segments in the memory, .bss and .heap (RAM memory), and .lowtext, .text and __reset_vector (Flash memory).

```
C:\ti\ccs1010\ccs\tools\compiler\msp430-gcc-9.2.0.50_win64\bin> msp430-elf-readelf.exe --
program-headers ToggleLEDs.out

Elf file type is EXEC (Executable file)
Entry point 0x4400
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x0000236c 0x0000436c 0x00094 0x00098 RW  0x4
```

```
  LOAD           0x000094 0x00004400 0x00004400 0x00042 0x00042 R E 0x4
  LOAD           0x0000d6 0x0000fffe 0x0000fffe 0x00002 0x00002 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00     .bss .heap
   01     .lowtext .text
   02     __reset_vector
```

Figure 17. msp430-elf-readelf: ELF Program Headers or Segments for ToggleLEDs.out (-l or –program-headers)

Step #5. **Disassemble the code**.

Now we are ready to take additional steps toward deconstructing the text segment that contains the code. We use msp430-elf-objdump –S to dump source code together with disassembly. Note this is a slight deviation from our assumption that source code is not available. Similar results can be obtained using –d (disassembly) that does not assume that source code is present. Figure 18 shows the result of disassembling operation of the text segment of ToggleLEDs.out executable file. The first thing we can notice is that the first instruction differs from the one shown in Figure 11. The entry point in the program is as expected 0x4400, but the first instruction is the one to initialize the stack pointer, rather than to stop the watchdog timer. This is because the compiler inserts so-called start-up code that proceed the main code. Thus, first instruction is actually moving the symbol that corresponds to the label __stack (the location above physical RAM) into R1 (stack pointer).

Next, '__crt0_call_main' is called, and finally, the main function is executed, starting at the address 0x440a.

```
C:\ti\ccs1010\ccs\tools\compiler\msp430-gcc-9.2.0.50_win64\bin> msp430-elf-objdump.exe -S
ToggleLEDs.out

C:\Users\prawar\Desktop\CPE325Workspace\ToggleLEDs\Debug\ToggleLEDs.out:     file format
elf32-msp430


Disassembly of section .lowtext:

00004400 <__crt0_start>:
    4400:       81 00 00 44     mova    #17408, r1      ;0x04400

00004404 <__crt0_call_main>:
    4404:       0c 43           clr     r12             ;

00004406 <.Loc.254.1>:
    4406:       b0 13 0a 44     calla   #17418          ;0x0440a

Disassembly of section .text:

0000440a <main>:
 *   Date:   November 2020
******************************************************************************/
#include  <msp430.h>

int main(void) {
   WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
```

```
    440a:       b2 40 80 5a     mov     #23168, &0x015c ;#0x5a80
    440e:       5c 01

00004410 <.Loc.28.1>:

   P1DIR |= BIT0;               // Set P1.0 to output direction (xxxx_xxx1) for LED1
    4410:       d2 d3 04 02     bis.b   #1,     &0x0204 ;r3 As==01

00004414 <.Loc.29.1>:
   P4DIR |= BIT7;               // Set P4.7 to output direction (1xxx_xxxx) for LED2
    4414:       f2 d0 80 ff     bis.b   #-128,  &0x0225 ;#0xff80
    4418:       25 02

0000441a <.Loc.31.1>:

   P1OUT |= BIT0;               // Set P1.0 ON (xxxx_xxx1) for LED1
    441a:       d2 d3 02 02     bis.b   #1,     &0x0202 ;r3 As==01

0000441e <.Loc.32.1>:
   P4OUT &= ~BIT7;              // Set P4.7 OFF (0xxx_xxxx) for LED2
    441e:       f2 f0 7f 00     and.b   #127,   &0x0223 ;#0x007f
    4422:       23 02
    4424:       80 00 32 44     mova    #17458, r0      ;0x04432

00004428 <.L3>:
     unsigned int i;

     P1OUT ^= BIT0;             // toggle the LED1
     P4OUT ^= BIT7;             // toggle the LED2

     for(i = 0; i < 50000; i++); // Software delay (13 cc per iteration)
    4428:       1c 53           inc     r12             ;

0000442a <.L2>:
    442a:       3d 40 4f c3     mov     #-15537,r13     ;#0xc34f
    442e:       0d 9c           cmp     r12,    r13     ;
    4430:       fb 2f           jc      $-8             ;abs 0x4428

00004432 <.L4>:
     P1OUT ^= BIT0;             // toggle the LED1
    4432:       d2 e3 02 02     xor.b   #1,     &0x0202 ;r3 As==01

00004436 <.Loc.38.1>:
     P4OUT ^= BIT7;             // toggle the LED2
    4436:       f2 50 80 ff     add.b   #-128,  &0x0223 ;#0xff80
    443a:       23 02

0000443c <.Loc.40.1>:
     for(i = 0; i < 50000; i++); // Software delay (13 cc per iteration)
    443c:       4c 43           clr.b   r12             ;

0000443e <.Loc.40.1>:
    443e:       80 00 2a 44     mova    #17450, r0      ;0x0442a
```

Figure 18. msp430-elf-objdump –S for ToggleLEDs.out

By analyzing the sequence of instructions in the main code, we should deduce what our program is doing. Figure 19 shows the disassembled code for the main code using msp430-objdump –d (there is no C statements displayed in the disassembled code). We can walk

through the code one by one instruction, write comments, and then tie everything together into a functional description of what this code does.

- Line 22 is a MOV instruction that moves immediate #23168 into the address 0x015c in the address space. This address represents the control register of the watchdog timer. Where do you find this information about register addresses?
- By analyzing the format of this register we can deduce that this instruction stops the watchdog timer.
- The next instruction is bis.b #1, &0x0204. At the address 0x0204 we have a P1DIR register, and this instruction will set port pins at bit positions 0 be output.
- Similarly, the next instruction sets the port pin of Port 4 at bit position 7 to output.
- The next instruction at 0x441a sets 0^{th} bit at the address 0x0202, which represents P1OUT.
- Similarly, the next instruction ANDs content of 0x0223 with 127. Can you guess what this statement does?
- The next instruction moves 0x4432 to R0 (PC). The means the PC points to 0x4432. Where command is executed after this?
- Commands from address 0x4428 to 0x4430 are the assembly instructions for our delay loop.
- Toggling of LEDs occur at instructions at 0x4432 and 0x4436.
- Finally, 0x442a is put into the PC at 0x443e thus calling the delay loop.

Thus, we can finally deduce that this code periodically toggles port pins P1.0 and P4.7 with a certain period.

```
1   C:\ti\ccs1010\ccs\tools\compiler\msp430-gcc-9.2.0.50_win64\bin> msp430-elf-objdump.exe -d
2   ToggleLEDs.out
3
4   C:\Users\prawar\Desktop\CPE325Workspace\ToggleLEDs\Debug\ToggleLEDs.out:     file format
5   elf32-msp430
6
7
8   Disassembly of section .lowtext:
9
10  00004400 <__crt0_start>:
11      4400:       81 00 00 44     mova    #17408, r1      ;0x04400
12
13  00004404 <__crt0_call_main>:
14      4404:       0c 43           clr     r12             ;
15
16  00004406 <.Loc.254.1>:
17      4406:       b0 13 0a 44     calla   #17418          ;0x0440a
18
19  Disassembly of section .text:
20
21  0000440a <main>:
22      440a:       b2 40 80 5a     mov     #23168, &0x015c ;#0x5a80
23      440e:       5c 01
24
25  00004410 <.Loc.28.1>:
26      4410:       d2 d3 04 02     bis.b   #1,     &0x0204 ;r3 As==01
27
```

```
00004414 <.Loc.29.1>:
    4414:       f2 d0 80 ff     bis.b   #-128,  &0x0225 ;#0xff80
    4418:       25 02

0000441a <.Loc.31.1>:
    441a:       d2 d3 02 02     bis.b   #1,     &0x0202 ;r3 As==01

0000441e <.Loc.32.1>:
    441e:       f2 f0 7f 00     and.b   #127,   &0x0223 ;#0x007f
    4422:       23 02
    4424:       80 00 32 44     mova    #17458, r0      ;0x04432

00004428 <.L3>:
    4428:       1c 53           inc     r12             ;

0000442a <.L2>:
    442a:       3d 40 4f c3     mov     #-15537,r13     ;#0xc34f
    442e:       0d 9c           cmp     r12,    r13     ;
    4430:       fb 2f           jc      $-8             ;abs 0x4428

00004432 <.L4>:
    4432:       d2 e3 02 02     xor.b   #1,     &0x0202 ;r3 As==01

00004436 <.Loc.38.1>:
    4436:       f2 50 80 ff     add.b   #-128,  &0x0223 ;#0xff80
    443a:       23 02

0000443c <.Loc.40.1>:
    443c:       4c 43           clr.b   r12             ;

0000443e <.Loc.40.1>:
    443e:       80 00 2a 44     mova    #17450, r0      ;0x0442a
```

Figure 19. msp430-elf-objdump –d for ToggleLEDs.out (main section)

A useful exercise is to select the TI compiler instead of MSP430 GCC, create a new executable file, and repeat the analysis of the executable using utilities discussed above: msp430-elf-readelf, msp430-elf-nm, msp430-elf-symbols, and msp430-elf-objdump. What insights can you gain from your analysis?

5 Working with HEX Files and MSP430Flasher Utility

5.1 Downloading HEX File to the Platform

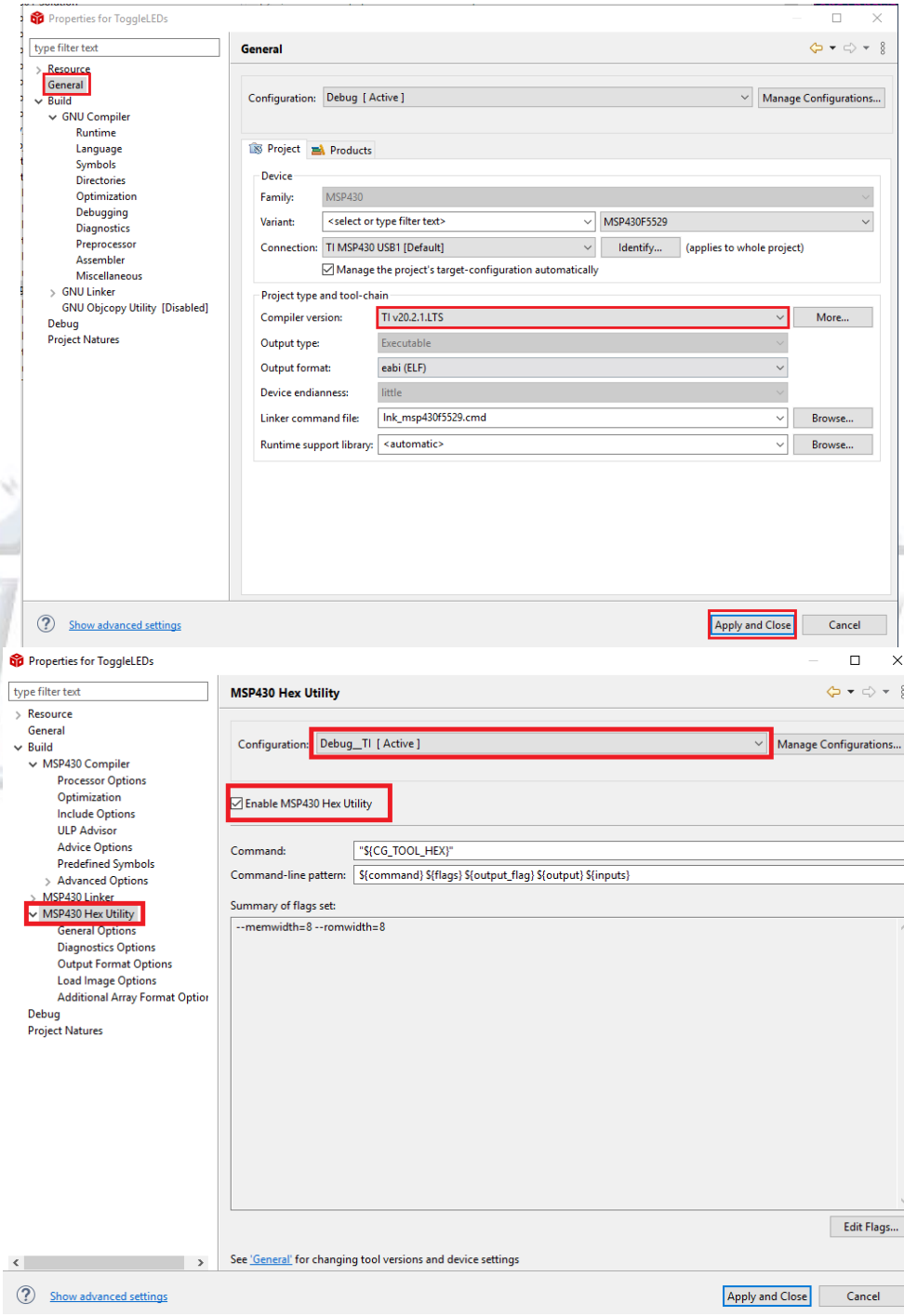
In this section we use ToggleLEDs.c program to demonstrate how to create a HEX file with executable and how to flash it on the target platform using a TI's MSP430Flasher utility program.

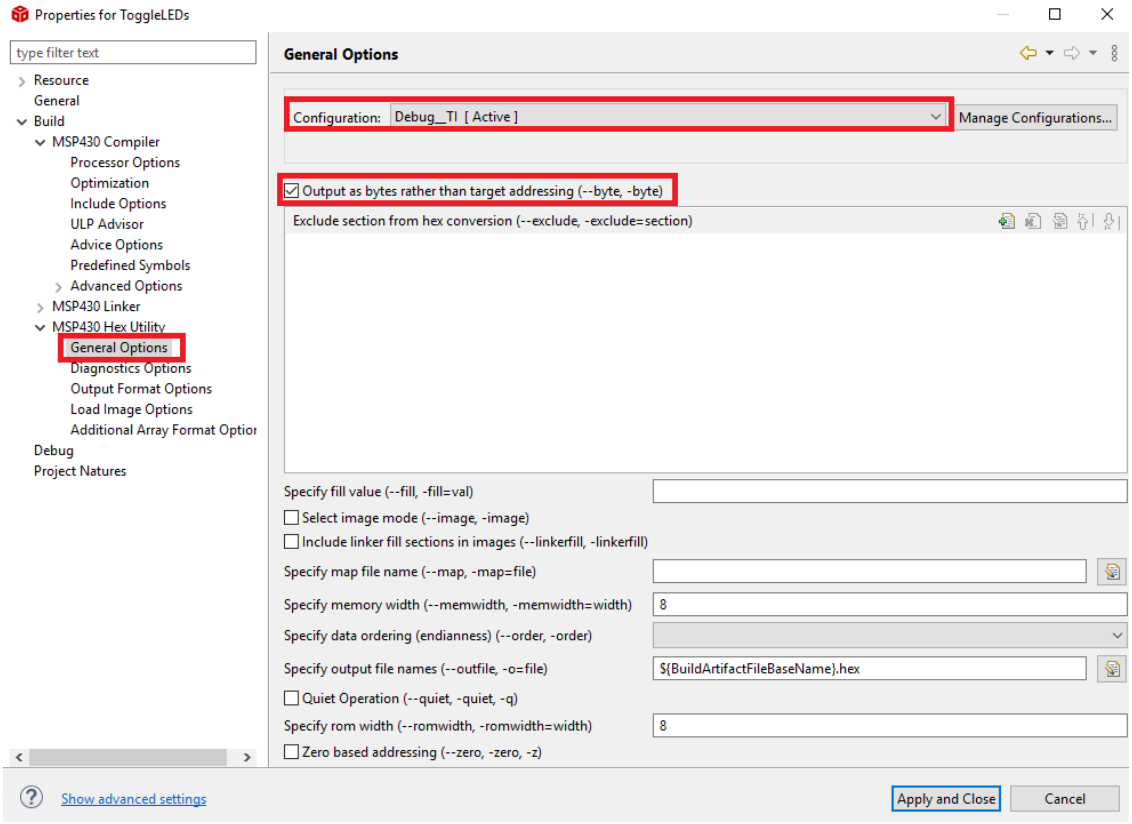
- We select the TI compiler in the CCS,
- Enable MSP430 HEX Utility,
- Set General Options and Output Format Options as shown in Figure 20.

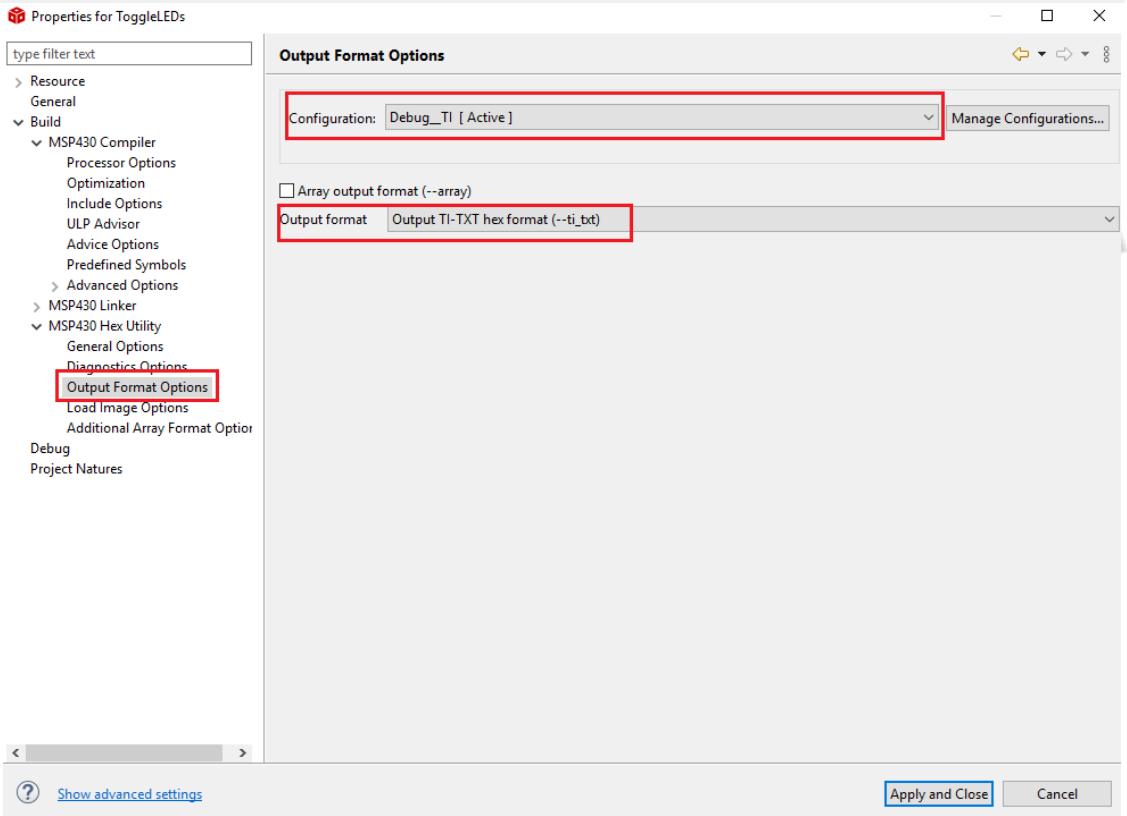
An output HEX file is created (ToggleLEDs.txt) and its content is shown in Figure 21. This file can be downloaded on the target platform using a TI utility called MSP430Flasher as shown in Figure 23. You can download the flasher tool from TI website at

http://www.ti.com/tool/MSP430-FLASHER. You may need to create an account and log-in. While installing, please make sure you note the installation folder. The default folder might be *C:\ti\MSPFlasher_1.3.20* in newer versions.

If everything goes all right, you should see the green and yellow LEDs flashing alternately.







And, finally after building the program, you can find the hex file as shown below:

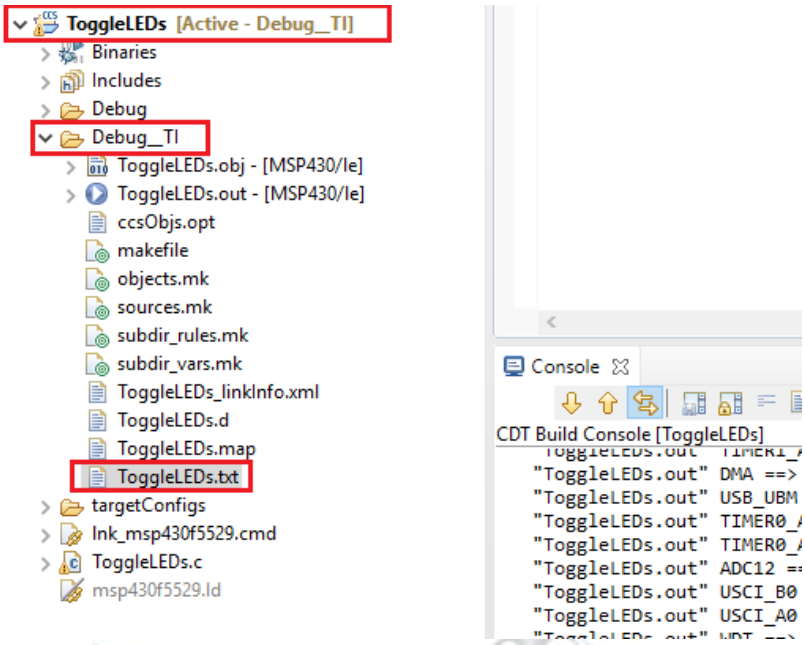


Figure 20. ToggleLEDs Project Properties for Generating HEX files

```
1   @4400
2   81 00 00 44 B1 13 3E 00 0C 43 B1 13 00 00 1C 43
3   B1 13 38 00 32 D0 10 00 FD 3F 03 43
4   @ffd2
5   14 44 14 44 14 44 14 44 14 44 14 44 14 44 14 44
6   14 44 14 44 14 44 14 44 14 44 14 44 14 44 14 44
7   14 44 14 44 14 44 14 44 14 44 14 44 00 44 B2 40
8   80 5A 5C 01 D2 D3 04 02 F2 D0 80 00 25 02 D2 D3
9   02 02 F2 F0 7F 00 23 02 D2 E3 02 02 F2 E0 80 00
10  23 02 0F 43 3F 90 50 C3 F7 2F 1F 53 3F 90 50 C3
11  F3 2F FB 3F 03 43 03 43 FF 3F 03 43 1C 43 10 01
12  q
```

Figure 21. ToggleLEDs.txt: Executable in HEX

Alternately to program the MSP430 microcontroller, you can also use Code Composer Studio if you have the Hex file as shown in Figure 22.

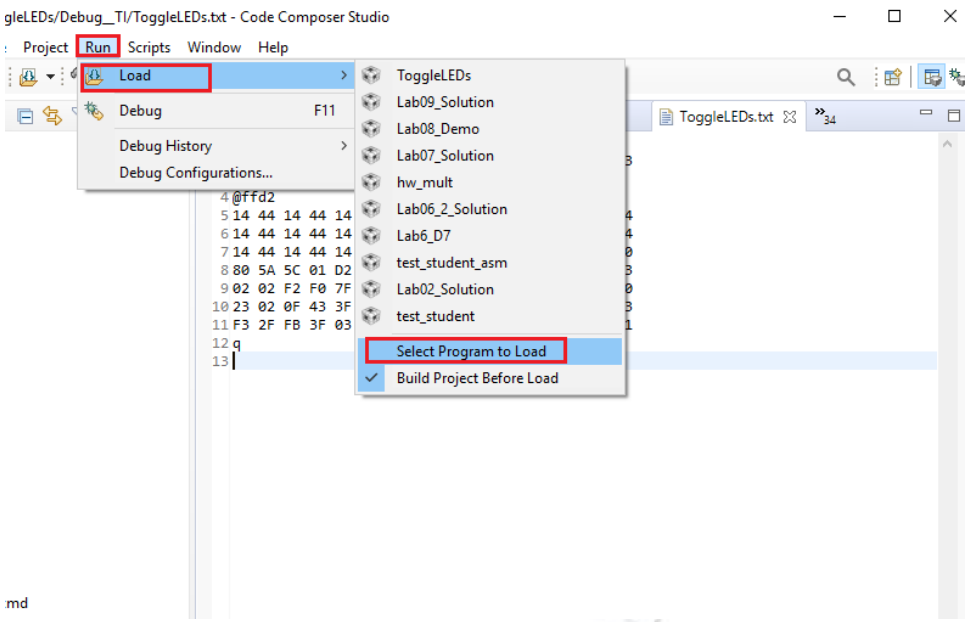


Figure 22. Using CCS to Download Code to the Platform (given a hex file)

```
C:\ti\MSPFlasher_1.3.20> MSP430Flasher.exe -n MSP430F5529 -w ToggleLEDs.txt -v -z [Vcc]

* Unable to access log file. Creating folder...done
* -----/|---------------------------------------------------------------------- *
*     / |__                                                                    *
*    /_   /   MSP Flasher v1.3.20                                              *
*      | /                                                                     *
* -----|/---------------------------------------------------------------------- *
*
* Evaluating triggers...done
* Checking for available FET debuggers:
* Found USB FET @ COM38 <- Selected
* Initializing interface @ COM38...done
* Checking firmware compatibility:
* FET firmware is up to date.
* Reading FW version...
* Debugger does not support target voltages other than 3000 mV!
* Setting VCC to 3000 mV...done
* Accessing device...done
* Reading device information...done
* Loading file into device...done
* Verifying memory
(C:\Users\prawar\Desktop\CPE325Workspace\ToggleLEDs\Debug__TI\ToggleLEDs.txt)...done
*
* ------------------------------------------------------------------------------
* Arguments   : -n MSP430F5529 -w
C:\Users\prawar\Desktop\CPE325Workspace\ToggleLEDs\Debug__TI\ToggleLEDs.txt -v -z [Vcc]
* ------------------------------------------------------------------------------
* Driver      : loaded
* Dll Version : 31400000
* FwVersion   : 31200000
* Interface   : TIUSB
* HwVersion   : E 3.0
* JTAG Mode   : AUTO
* Device      : MSP430F5529
* EEM         : Level 7, ClockCntrl 2
* Erase Mode  : ERASE_ALL
```

```
* Prog.File   : C:\Users\prawar\Desktop\CPE325Workspace\ToggleLEDs\Debug__TI\ToggleLEDs.txt
* Verified    : TRUE
* BSL Unlock  : FALSE
* InfoA Access: FALSE
* VCC ON      : 3000 mV
* ------------------------------------------------------------------------------
* Starting target code execution...done
* Disconnecting from device...done
*
* ------------------------------------------------------------------------------
* Driver      : closed (No error)
* ------------------------------------------------------------------------------
*/
```

Figure 23. Running MSP430Flasher to Download Code to the Platform (given a hex file)

The command used to download the code into platform using MSP430Flasher is highlighted in the text above.

5.2 Retrieving Code from the Platform

MSP430Flasher utility supports many useful functions in addition to downloading code as shown in Figure 23. Here, we are especially interested in an option of retrieving the machine code from the actual platform and storing it into an output file in either HEX or text format. Figure 24 shows the process of extracting the code from the platform.

Make sure you are aware of where the command prompt or powershell is located while you are running the command. In the Figure 24 below, it is run from desktop (as highlighted in green), so the output file will be created in desktop.

```

C:\Users\prawar\Desktop> C:\ti\MSPFlasher_1.3.20\MSP430Flasher.exe -r [RetrievedHEX.txt,MAIN]

* Unable to access log file. Creating folder...done
* -----/|---------------------------------------------------------------- *
*     / |__                                                               *
*    /_   /   MSP Flasher v1.3.20                                         *
*      | /                                                                 *
* -----|/---------------------------------------------------------------- *
*
* Evaluating triggers...done
* Checking for available FET debuggers:
* Found USB FET @ COM38 <- Selected
* Initializing interface @ COM38...done
* Checking firmware compatibility:
* FET firmware is up to date.
* Reading FW version...done
* Setting VCC to 3000 mV...done
* Accessing device...done
* Reading device information...done
* Dumping memory from MAIN into RetrievedHEX.txt...done
*
* ------------------------------------------------------------------------------
* Arguments   : -r [RetrievedHEX.txt,MAIN]
* ------------------------------------------------------------------------------
* Driver      : loaded
* Dll Version : 31400000
* FwVersion   : 31200000
* Interface   : TIUSB
```

```
* HwVersion   : E 3.0
* JTAG Mode   : AUTO
* Device      : MSP430F5529
* EEM         : Level 7, ClockCntrl 2
* Read File   : RetrievedHEX.txt (memory segment = MAIN)
* VCC OFF
* ------------------------------------------------------------------------------
* Powering down...done
* Disconnecting from device...done
*
* ------------------------------------------------------------------------------
* Driver      : closed (No error)
* ------------------------------------------------------------------------------
*/
```

Figure 24. Running MSP430Flasher to Retrieve Code from the Platform

The output file RetrivedHEX.txt contains the hexadecimal content of the entire Flash memory starting from the address 0x4400. This file is relatively big as it includes the content of the entire Flash memory. The memory locations that contain 0xFF are actually erased locations that do not contain any useful information and thus can be removed manually from the file. The resulting file without erased Flash locations is named RetrievedHEX_Stripped.txt.

The stripped file has the following contents:

```
@4400
B2 40 80 5A 5C 01 D2 D3 04 02 F2 D0 80 00 25 02
D2 D3 02 02 F2 F0 7F 00 23 02 D2 E3 02 02 F2 E0
80 00 23 02 0F 43 3F 90 50 C3 F7 2F 1F 53 3F 90
50 C3 F3 2F FB 3F 03 43 31 40 00 44 B0 12 52 44
0C 43 B0 12 00 44 1C 43 B0 12 4C 44 03 43 FF 3F
03 43 1C 43 30 41 32 D0 10 00 FD 3F 03 43
q
```

Figure 25. Stripped File after Retrieving the Code from the Platform

The next step is to run a disassembler that takes a HEX file as an input (the stripped file) and produces assembly code that can be inspected and reverse engineered manually. For this purpose, we use naken_util disassembler developed by Michael Kohn and Joe Davisson. You can download the naken_util from :

```
C:\Users\prawar\Desktop> naken_util.exe -msp430 -disasm RetrievedHEX_stripped.txt >
ReverseMe.txt
```

Figure 26 shows the resulting assembly code created by naken_util. The next step is to analyze the code line-by-line as shown in Figure 27. We can easily recognize that this code corresponds to ToggleLEDs program. Note: This implementation differs from the one we analyzed above because this one is created using TI Compiler instead of MSP430 GCC.

```

naken_util - by Michael Kohn
                Joe Davisson
    Web: http://www.mikekohn.net/
  Email: mike@mikekohn.net
```

```
Version: April 25, 2020

Loaded ti_txt C:\Users\prawar\Desktop\RetrievedHEX_stripped.txt from 0x4400 to 0x445d
Type help for a list of commands.

Addr    Opcode Instruction                          Cycles
------- ------ -----------------------------------  ------
0x4400: 0x40b2 mov.w #0x5a80, &0x015c                5
0x4402: 0x5a80
0x4404: 0x015c
0x4406: 0xd3d2 bis.b #1, &0x0204                     4
0x4408: 0x0204
0x440a: 0xd0f2 bis.b #0x80, &0x0225                  5
0x440c: 0x0080
0x440e: 0x0225
0x4410: 0xd3d2 bis.b #1, &0x0202                     4
0x4412: 0x0202
0x4414: 0xf0f2 and.b #0x7f, &0x0223                  5
0x4416: 0x007f
0x4418: 0x0223
0x441a: 0xe3d2 xor.b #1, &0x0202                     4
0x441c: 0x0202
0x441e: 0xe0f2 xor.b #0x80, &0x0223                  5
0x4420: 0x0080
0x4422: 0x0223
0x4424: 0x430f mov.w #0, r15                         1
0x4426: 0x903f cmp.w #0xc350, r15                    2
0x4428: 0xc350
0x442a: 0x2ff7 jhs 0x441a  (offset: -18)             2
0x442c: 0x531f add.w #1, r15                         1
0x442e: 0x903f cmp.w #0xc350, r15                    2
0x4430: 0xc350
0x4432: 0x2ff3 jhs 0x441a  (offset: -26)             2
0x4434: 0x3ffb jmp 0x442c  (offset: -10)             2
0x4436: 0x4303 nop   --  mov.w #0, CG                1
0x4438: 0x4031 mov.w #0x4400, SP                     2
0x443a: 0x4400
0x443c: 0x12b0 call #0x4452                          5
0x443e: 0x4452
0x4440: 0x430c mov.w #0, r12                         1
0x4442: 0x12b0 call #0x4400                          5
0x4444: 0x4400
0x4446: 0x431c mov.w #1, r12                         1
0x4448: 0x12b0 call #0x444c                          5
0x444a: 0x444c
0x444c: 0x4303 nop   --  mov.w #0, CG                1
0x444e: 0x3fff jmp 0x444e  (offset: -2)              2
0x4450: 0x4303 nop   --  mov.w #0, CG                1
0x4452: 0x431c mov.w #1, r12                         1
0x4454: 0x4130 ret   --  mov.w @SP+, PC              3
0x4456: 0xd032 bis.w #0x0010, SR                     2
0x4458: 0x0010
0x445a: 0x3ffd jmp 0x4456  (offset: -6)              2
0x445c: 0x4303 nop   --  mov.w #0, CG                1
```

Figure 26. Disassembled Code in ReverseMe.asm.txt Created Using naken_util

```
Addr    Opcode Instruction                          Cycles
------- ------ -----------------------------------  ------
```

```
0x4400: 0x40b2 mov.w #0x5a80, &0x015c                    5 // moves 0x5a80 to 0x015c(WDTCTL)
0x4402: 0x5a80
0x4404: 0x015c
0x4406: 0xd3d2 bis.b #1, &0x0204                         4 // set 0x01(BIT0) at 0x0204 (PADIR_L
ie P1DIR)
0x4408: 0x0204
0x440a: 0xd0f2 bis.b #0x80, &0x0225                      5 // set 0x80(BIT7) at 0x0225 (P4DIR)
0x440c: 0x0080
0x440e: 0x0225
0x4410: 0xd3d2 bis.b #1, &0x0202                         4 // set 0x01(BIT0) at 0x0202 (P1OUT)
0x4412: 0x0202
0x4414: 0xf0f2 and.b #0x7f, &0x0223                      5 // and 0x7f (all bits except BIT7)
at 0x0223 (P4OUT)
0x4416: 0x007f
0x4418: 0x0223
0x441a: 0xe3d2 xor.b #1, &0x0202                         4 // toggle BIT0 at 0x0202
0x441c: 0x0202
0x441e: 0xe0f2 xor.b #0x80, &0x0223                      5 // toggle BIT7 at 0x0223
0x4420: 0x0080
0x4422: 0x0223
0x4424: 0x430f mov.w #0, r15                             1 // delay loop
0x4426: 0x903f cmp.w #0xc350, r15                        2 // delay loop
0x4428: 0xc350
0x442a: 0x2ff7 jhs 0x441a  (offset: -18)                 2 // delay loop
0x442c: 0x531f add.w #1, r15                             1 // delay loop
0x442e: 0x903f cmp.w #0xc350, r15                        2 // delay loop terminating condition
0x4430: 0xc350
0x4432: 0x2ff3 jhs 0x441a  (offset: -26)                 2 // escape from delay loop
0x4434: 0x3ffb jmp 0x442c  (offset: -10)                 2 // delay loop
0x4436: 0x4303 nop   --  mov.w #0, CG                    1
```

Figure 27. Reverse Engineering of the Code in Disassembled Code Using naken_util

6 To Learn More

1. Texas Instruments, MSP430 GCC User's Guide: http://www.ti.com/lit/ug/slau646c/slau646c.pdf
2. MSP430 Flasher: http://www.ti.com/tool/MSP430-FLASHER (should be installed on your workstation and its exe directory, e.g. c:\ti\MSP430Flasher_1.3.18, should be in the PATH system environment variable)
3. Mike Kohn's Naken_asm: https://www.mikekohn.net/micro/naken_asm.php (should be installed on your workstation and its exe directory, e.g., c:\ti\naken_asm, should be in the PATH system environment variable)