MTE 241 LAB 1 -- Debugging

Revised for online delivery by Bernie Roehl, April 2020

Learning Objectives

The purpose of this lab is to introduce you to techniques for debugging programs on embedded systems. After completing this lab, you should be able to:

- Identify and fix C syntax errors
- Understand and fix common compile time errors
- Identify code that does not conform to common C formatting standards and make the appropriate changes
- Use the debugger to
    - Watch local and global variables
    - Check the contents of memory
    - Set breakpoints
    - Step through code
- Correct logic errors

About the Project

In this lab, you'll be debugging a program that uses binary search trees.

The binary search tree struct stores the address of the root node and the number of nodes currently stored in the tree. This is implemented using the bst_t structure, as defined in bst.h.

Every node within the tree stores the addresses of both the left and right children (NULL if there is no child there) as well as the integer value stored in this node (of type S32). This is implemented using the bst_n structure, as defined in bst.h.

The bst.c file implements the following functions (with some errors you will need to fix) to manipulate the binary search tree with *n* nodes and a height *h*:

void bst_init(bst_t *);
Initialize the binary search tree so that it is empty. Run time: *Θ*(1)

U32 bst_size();
Return the number of nodes in the binary search tree. Run time: *Θ*(1)

bool bst_insert(bst_t *, S32);
Insert the given integer into the binary search tree and return false if the node is already in the tree (do not add a duplicate into the tree) and true otherwise. Run time: O(h)

S32 bst_min(bst_t *);
Returns the smallest integer in the binary search tree. Return

INT_MAX if the tree is empty. Run time: O(h)

S32 bst_max(bst_t *);
Returns the largest integer in the binary search tree. Return INT_MIN if the tree is empty. Run time: O(h)

bool bst_erase(bst_t *, S32);
If the object is in the binary search tree, remove it and return true; otherwise, return false and do nothing. Run time: O(h)

While completing the various exercises, you are welcome to create other helper functions and you are welcome to add additional fields onto any of the records as you find necessary.

#include <stdbool.h> has been added to allow access to the type `bool`.

#include <limits.h> is used to access `INT_MIN` and `INT_MAX`.

Goals

Your goals in this lab are as follows:

1. Fix the syntax errors so that the code compiles with 0 errors and 0 warnings.
2. Improve the C formatting and style, and fill in the comments block at the top of bst.c to document what you changed.
3. Fix the logic errors so the program produces the output shown below. There are four logic errors, but one of them (interestingly) has no impact on the results. Remember to fill in the comment block at the top of bst.c to document what you changed.

Expected output:

	Min	Max
Before first group erased:	-3	9593
After first group erased:	-3	9593
After second group erased:	23	9593
After third group erased:	140	9593
After fourth group erased:	140	9265
After fifth group erased:	2147483647	-2147483648

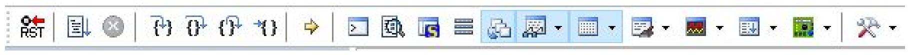
Marking

The lab will be marked as follows:

- 20% for getting the code to compile without errors or warnings.
- 10% for finding four formatting and style issues
- 50% for solving the logic bugs and getting the correct output
- 20% for being able to answer questions about the code (these questions are answered individually by each student in the group)

More Background Info on the Debugger

The debug controls are shown in the figure below. The first 8 buttons from the left allow you to start the program, stop the program and step through the code in a few different ways. Holding your mouse over each of the icons in *μ*Vision will show a tooltip explaining the functionality of the button.



Buttons 9 through 19 control what windows are visible in your debug view. The first time you start the debugger, you will likely have additional windows visible. Holding your mouse over each icon will open a tooltip with more information.

For Lab 1, we will introduce breakpoints, the Call Stack Window, Watch Windows and Memory Windows.

Breakpoints

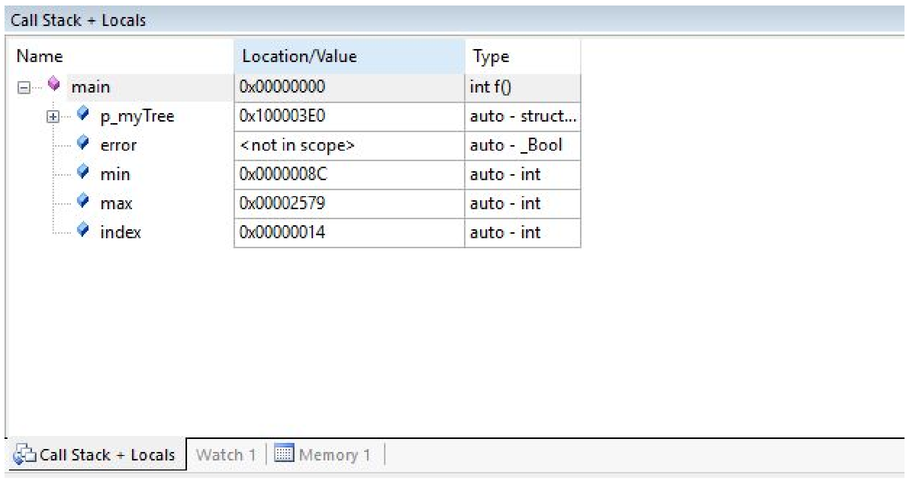
Being able to stop the code at a particular point in order to see what is going on is one of the primary uses of a debugger. This is done using breakpoints.

*μ*Vision supports execution, read/write access, and complex breakpoints. An execution breakpoint can be placed at any line of Assembly or C code. To place a breakpoint, find the line you want to stop on, then go to the Debug menu and select Insert/Remove Breakpoint. You can also click just to the left of the line number. A red circle on the left side of the intended code shows the breakpoint has been placed and the execution will stop there when the program is run. Breakpoints can be disabled or removed via the Debug menu or by clicking the red circle next to the line number.

Once execution is paused on a statement, the processing unit is stopped and the program counter does not proceed to the next statement, so you can see the call stack content, register values, watched variables, and port values. You can use the Step Into, Step Over and Step Out from the next statement buttons, or Run To Cursor Line to advance the program. The execution trace can also be continued using the Run command, or terminated using Stop command. You can also use the Reset button to restart the execution.

Call Stack and Local Window

The Call Stack and Locals window shows you two things: the sequence of function calls that have been made, and the values of variables that are currently in scope. An example is shown in the figure below.



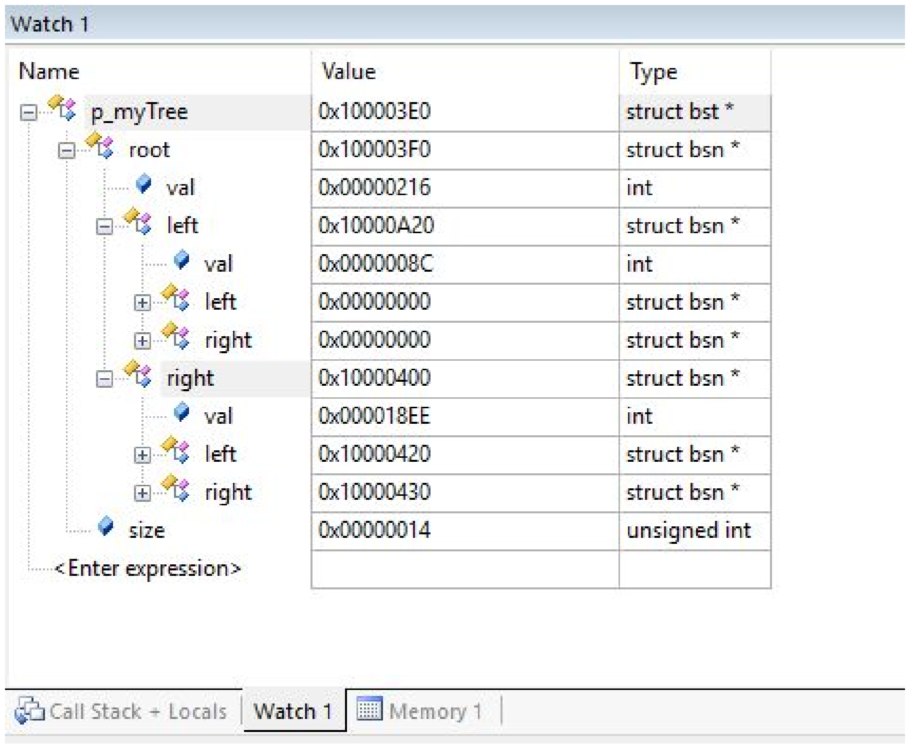
Call Stack + Locals

Name	Location/Value	Type
main	0x00000000	int f()
p_myTree	0x100003E0	auto - struct...
error	<not in scope>	auto - _Bool
min	0x0000008C	auto - int
max	0x00002579	auto - int
index	0x00000014	auto - int

Call Stack + Locals | Watch 1 | Memory 1

Watch Windows

Using the watch window, you can see the content of any variable at any time while the execution is stopped. To watch a variable, select the variable, right click on the selected name, and choose Add to Watch 1. The content of the variable becomes visible whenever the variable is in scope within the current trace. An example showing the binary tree from Project 1 is shown in the figure below.



Watch 1

Name	Value	Type
p_myTree	0x100003E0	struct bst *
root	0x100003F0	struct bsn *
val	0x00000216	int
left	0x10000A20	struct bsn *
val	0x0000008C	int
left	0x00000000	struct bsn *
right	0x00000000	struct bsn *
right	0x10000400	struct bsn *
val	0x000018EE	int
left	0x10000420	struct bsn *
right	0x10000430	struct bsn *
size	0x00000014	unsigned int
<Enter expression>		

Call Stack + Locals | Watch 1 | Memory 1

The p_myTree node has been expanded to show how this view can be used to trace the location and contents of each node in the tree. Notice that each variable has both a location, where it is physically located in memory, and a value.

Memory Windows

The memory window allows you to inspect the current contents of memory while program execution is stopped. Enter an address in the Address box at the top of the window. (Hint: for Lab 1 the start address of the memory you are managing is a good place to start). The memory contents starting from that address are displayed. Right click on the data to control the display format.

Given an address location, the memory window shows the data as bytes or words depending on the data format selected.

The memory content can also be shown in hexadecimal or decimal. As shown in the following figure, right clicking in the memory window opens a menu from which many formats can be selected. The appropriate choice will depend on the type of data you are trying to review. If your data is a string array, choose Unsigned->Char. If your data is an array of unsigned integers, choose Unsigned->Int.

Acknowledgments

Parts of the material shown here were adapted from material originally prepared by Douglas Harder. Adaptations and additions have been made by Allyson Giannikouris, Trevor Smouter and Bernie Roehl.