



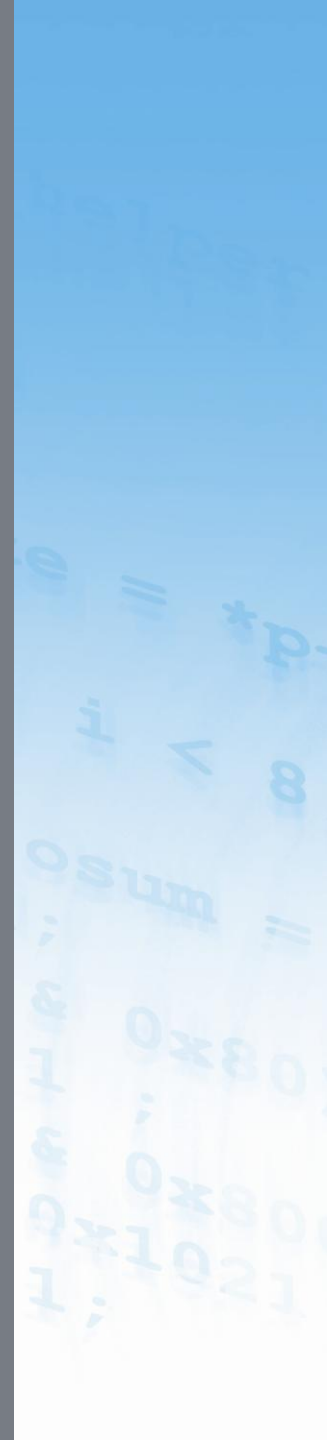
Working with the Stack and Heap

Agenda

- What is the heap?
- Determining heap size
- Potential Problems in Working with the Heap
- Heap Layout Considerations
- Working with the Stack
- Determining stack size
- Potential Problems in working with the Stack
- Static stack checkers
- Stack plug-in in the Embedded Workbench
- Demonstration

What is the heap?

- The heap is an area of memory reserved for dynamic memory allocation
- When an application needs to use a certain amount of memory temporarily it can allocate, or borrow, this memory from the heap by calling the `malloc()` function in C or by using the 'new' operator in C++
- When this memory is no longer needed it can be freed up by calling `free()` or by using the `delete` operator. Once the memory is freed it can be reused by future allocations
- The location and size of the heap are set statically at compile-time
- It is important that you allocate enough heap for your application, else it will crash during execution!



Determining heap size

The first question to be answered when considering an application using dynamic memory is how much heap do I need?

- This is really more of an estimate than a determination
- If you know how your application behaves then you should have some idea of how much memory needs to be allocated at any one time



Determining heap size

Another factor to consider is that there will be some overhead in the management of the heap.

- The heap manager will need to keep track of the amount of heap used or remaining, the size of the block being allocated and will usually contain a pointer to the next available memory location available.
- One more factor is that the tools may reserve a larger block of memory than requested to accommodate the memory architecture. For instance, EWARM compiler always allocates blocks of memory in multiples of 8 bytes to maintain stack alignment

Potential problems

One of the most common problems with dynamic memory allocation occurs when blocks of memory of varying size are frequently allocated and freed

- When memory is freed, there will be a memory hole
- This can be a problem if the next allocation is larger than any of the available holes
- This can lead to difficulties in debugging since the total amount of free space on the heap may be sufficient for a desired allocation but allocations may fail since the free space is not contiguous.

Potential problems

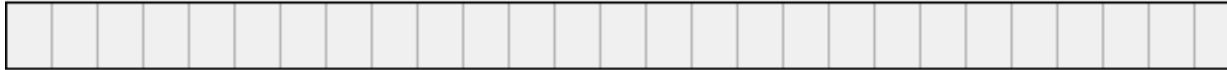
For example, assume the following:

- The heap lies from 0x20000 - 0x20FFF
- A small block of memory, 8 bytes, is allocated at the beginning of the stack
- Immediately following this block of memory is a block of 1024 bytes

At some later time, the first block is freed and can be reused. However, the application needs to allocate a block of memory larger than 8 bytes so the free block at the beginning of the heap cannot be used. This shows how the heap can become fragmented, if the application never again requests a block of 8 bytes or less, then the block at the beginning of the heap is wasted. This example also demonstrates how small blocks of memory on the heap are inefficient, the overhead is equal in size to the amount of data available to the application.

Potential problems

The heap is initially empty:



We allocate an object "Foo":



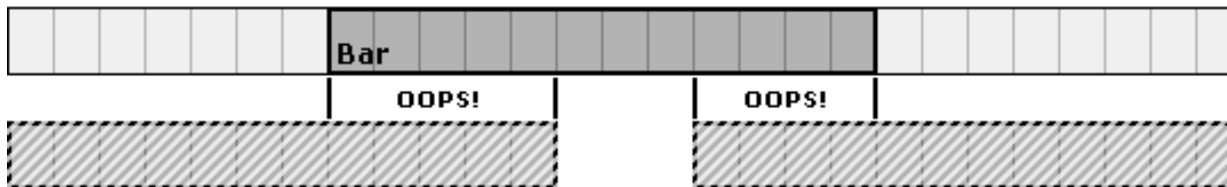
Then another object "Bar":



We delete Foo, leaving the heap fragmented into two parts:



When we try to allocate another "Bar" now, it won't fit in either part:



Layout of the heap

- It may be useful in some situations to know exactly what is on the heap and where it is located.
- The heap contains more than just the data allocated
 - There is some overhead to maintain the heap
 - Each block of memory allocated will contain two integers, the size of the pointer will depend on the computer architecture, e.g. an ARM device will use 32-bit integers.
 - The first of these integers will indicate the size, in bytes, of the block of memory allocated
 - There will be a 32 bit value following the last heap object indicating how much heap space is remaining.
 - The 32 bit value at the beginning of the last heap allocation indicates the address of this data.

Layout of the heap

The heap also keeps track of the next available location for allocations in a structure which resides outside of the heap.

```
typedef struct {  
    __data_Cell __data * __data *_Plast;  
    __data_Cell __data *_Head; }  
__data_Altab;
```

The `_Head` pointer indicates the current status of the heap. The `_Plast` pointer indicates where to resume scanning the heap for available blocks and is not currently used.

Layout of the heap

The `__data_Cell` structure is defined as:

```
typedef struct __data_Cell
{
    __data_size_t _Size;
    struct __data_Cell __data *_Next;
} __data_Cell;
```

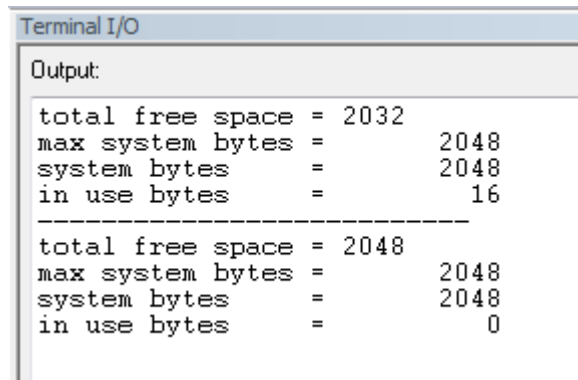
where `_Size` indicates the amount of heap remaining and `_Next` indicates the next address available for memory allocation. If the heap is exhausted the `_Next` pointer will be `NULL`.

Final thoughts on the heap

- It is difficult to estimate how much heap space you will need without some sort of tool to help analyze your dynamic memory needs
 - There exist such tools for desktop Java (HAT, Heap Analysis Tool)
 - No such tool as yet for Embedded C/C++
- As embedded systems have limited resources, dynamic memory should be used sparingly due to overhead and the possibility of heap fragmentation
 - Examine ways in your code (test cases) that you can create multiple instantiations of structs or objects
 - MISRA C requires that all memory be statically allocated at compile-time such that you can **never** run out of heap

Heap statistics in EWARM

- The C-Spy debugger in EWARM gives you the ability to see how much of your heap is being currently used as well as its maximum usage
- To see this, you must include `dlmalloc.c` in your project
- Calling `__iar_dlmallinfo()` and `__iar_dlmalloc_stats()` will print your heap statistics to the Terminal I/O window

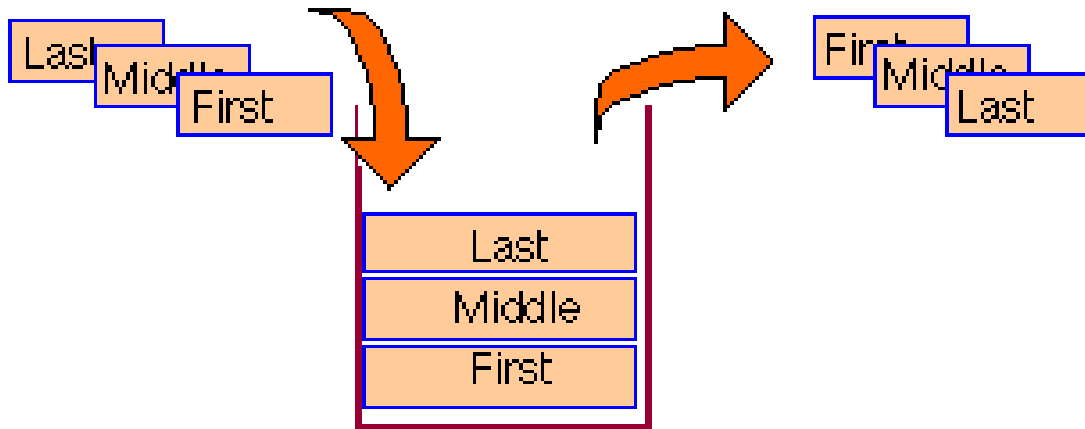
A screenshot of the 'Terminal I/O' window in the C-Spy debugger. The window has a title bar 'Terminal I/O' and a label 'Output:'. It displays two sets of heap statistics. The first set shows 'total free space = 2032', 'max system bytes = 2048', 'system bytes = 2048', and 'in use bytes = 16'. A horizontal line separates this from the second set, which shows 'total free space = 2048', 'max system bytes = 2048', 'system bytes = 2048', and 'in use bytes = 0'.

```
Terminal I/O
Output:
total free space = 2032
max system bytes =      2048
system bytes     =      2048
in use bytes     =         16
-----
total free space = 2048
max system bytes =      2048
system bytes     =      2048
in use bytes     =          0
```

- For more information, see <http://supp.iar.com/Support/?note=28545>

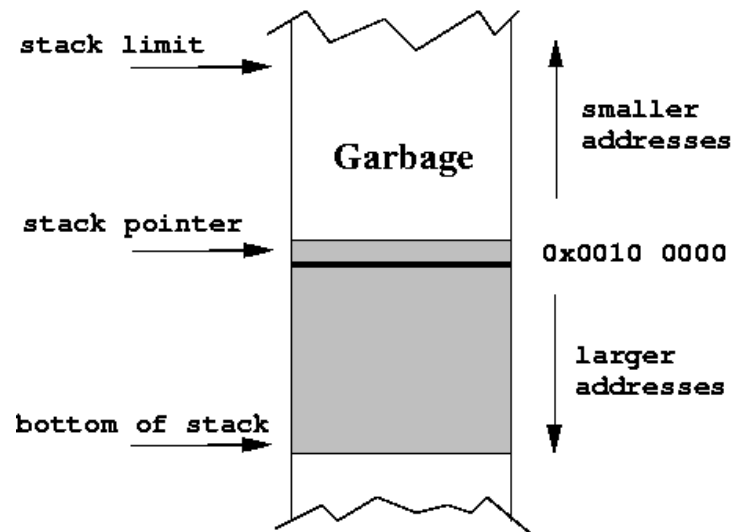
Working with the stack

- The concept of the stack is relatively simple to understand
- The stack is a LIFO structure similar to packing items in a box
 - The last item placed into the box is the first item removed from the box
 - However, the stack – like the box – has a fixed size and should NOT be overflowed



Working with the stack

- The stack is a fixed block of memory, divided into two parts:
 - Allocated memory used by the function that called the current function, and the function that called it, etc.
 - Free memory that can be allocated
- The borderline between the two areas is called the top of stack and is represented by the stack pointer (SP), which is a dedicated processor register
- Memory is allocated on the stack by moving the stack pointer



Working with the stack

A function should never refer to the memory in the area of the stack that contains free memory - if an interrupt occurs, the called interrupt function can:

- Allocate memory
- Modify memory
- Deallocate memory

on the stack...and functions are never the wiser that their memory has been corrupted!

Working with the stack

- The main advantage of the stack is that functions in different parts of the program can use the same memory space to store their data
- Unlike a heap, a stack will never become fragmented or suffer from memory leaks
- It is possible for a function to call itself—a *recursive function*—and each invocation can store its own data on the stack

Determining stack size

In order to determine the necessary size of the stack, we must understand what sorts of things get put on the stack:

- Auto variables, which are temporary local variables and parameters not stored in a register
- Temporary results of expressions
- Return value of a function (unless passed in a register)
- Processor state during interrupts
- Processor registers that are to be restored before the function returns

As you can quickly see, the size of the stack depends heavily on the number of function calls that you have in your code.

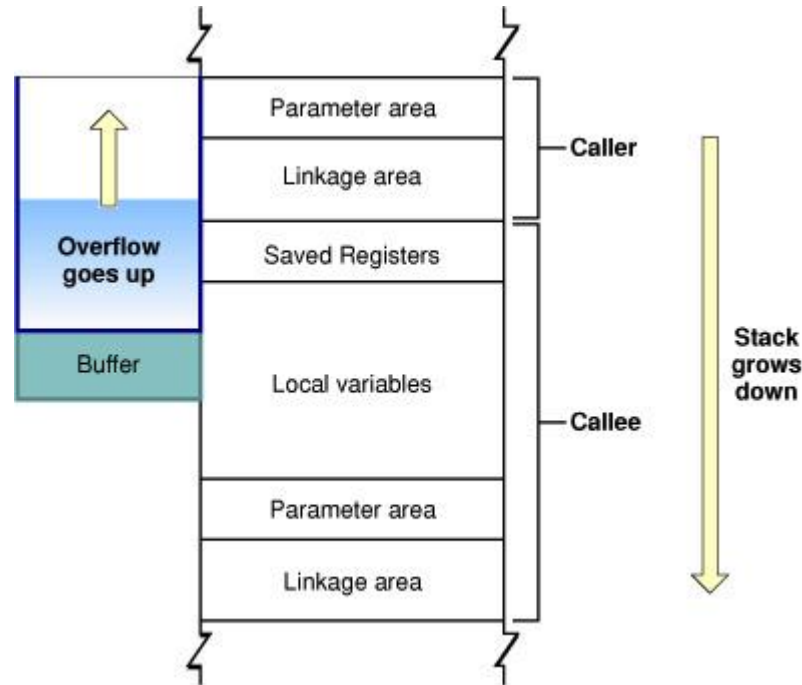
Potential problems working with the Stack

The way the stack works makes it impossible to store data that is supposed to live after the function returns. The following function demonstrates a common programming mistake. It returns a pointer to the variable `x`, a variable that ceases to exist when the function returns:

```
int *MyFunction()  
{  
    int x;  
    /* Do something here. */  
    return &x; /* Incorrect */  
}
```

Potential problems working with the Stack

- Another problem is the risk of running out of stack. This will happen when one function calls another, which in turn calls a third, etc., and the sum of the stack usage of each function is larger than the size of the stack.
- The risk is higher if large data objects are stored on the stack, or when recursive functions—functions that call themselves either directly or indirectly—are used.



Potential problems working with the Stack

- If the given stack size is too large, RAM will be wasted
- If the given stack size is too small, two things can happen, depending on where in memory you located your stack:
 - Variable storage will be overwritten, leading to undefined behavior
 - The stack will fall outside of the memory area, leading to an abnormal termination of your application
- Because the second alternative is easier to detect, you should consider placing your stack so that it grows toward the end of the memory.

Static Stack Checkers

There are several static C checkers:

- Express Logic's StackX
- AbsInt's StackAnalyzer
- John Regher's Stack Analysis (for AVR/430 only)
- AdaCore's GNATStack



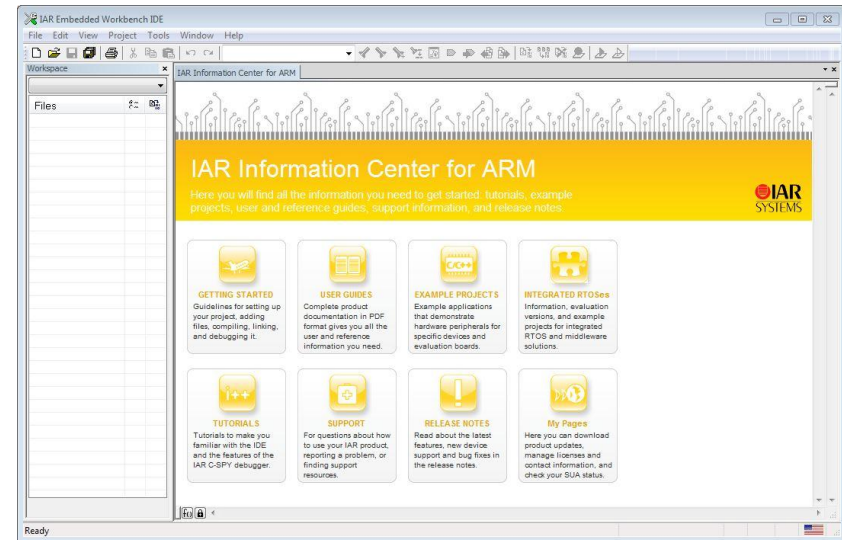
Embedded Workbench's Stack Plug-in

The Embedded Workbench contains a handy stack plug-in that monitors the size of the CSTACK and can print a message to the debug log if you exceed a certain stack threshold.

Caveat: The plug-in is not RTOS-aware, so it cannot be used with an RTOS as it will always report that you have overflowed the stack.

Demo

- Open up the Embedded Workbench for ARM
- Choose “Example Projects” from the Information Center and select a project
- Check the Project-Options-Debugger-Plugins to make sure it is setup to use the stack plugin
- Recompile the code to make sure it compiles with no errors or warnings, then download and debug!



Q&A

