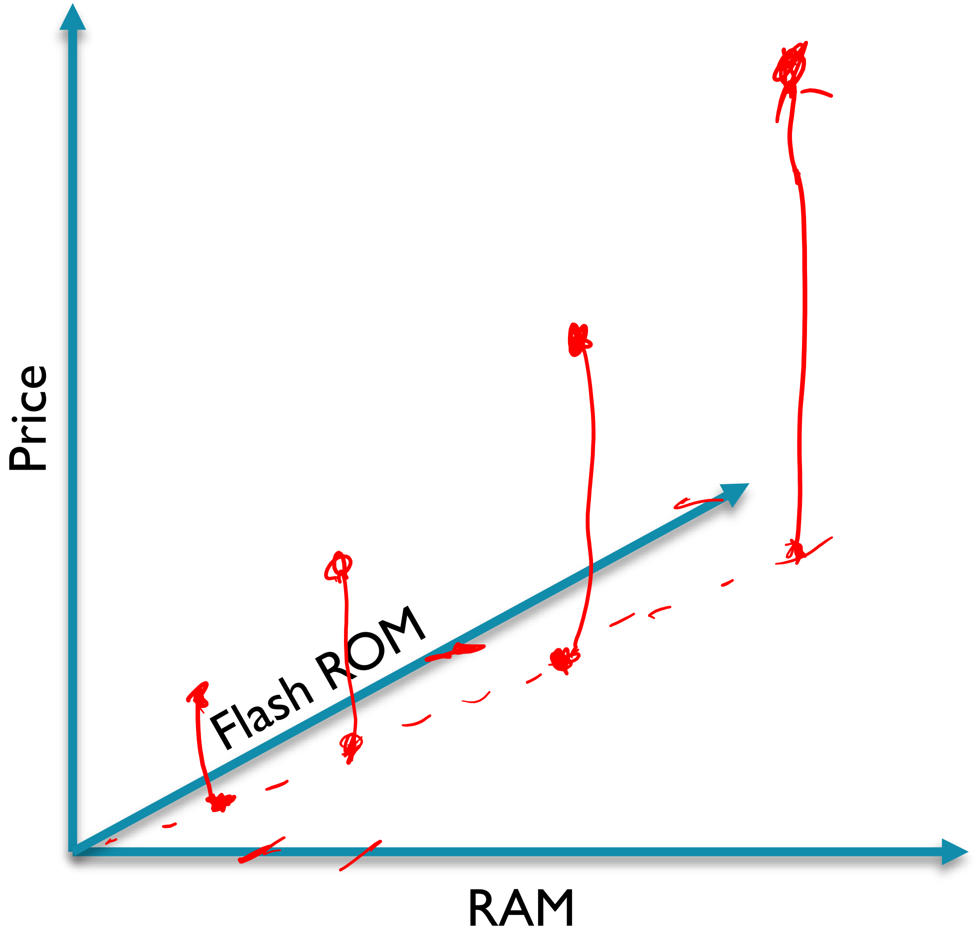
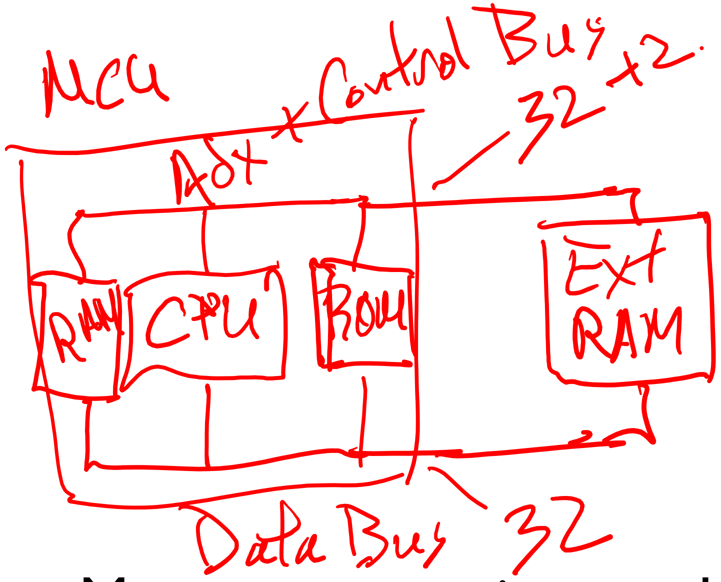
Memory Size Analysis and Optimization

MCUs and Memory

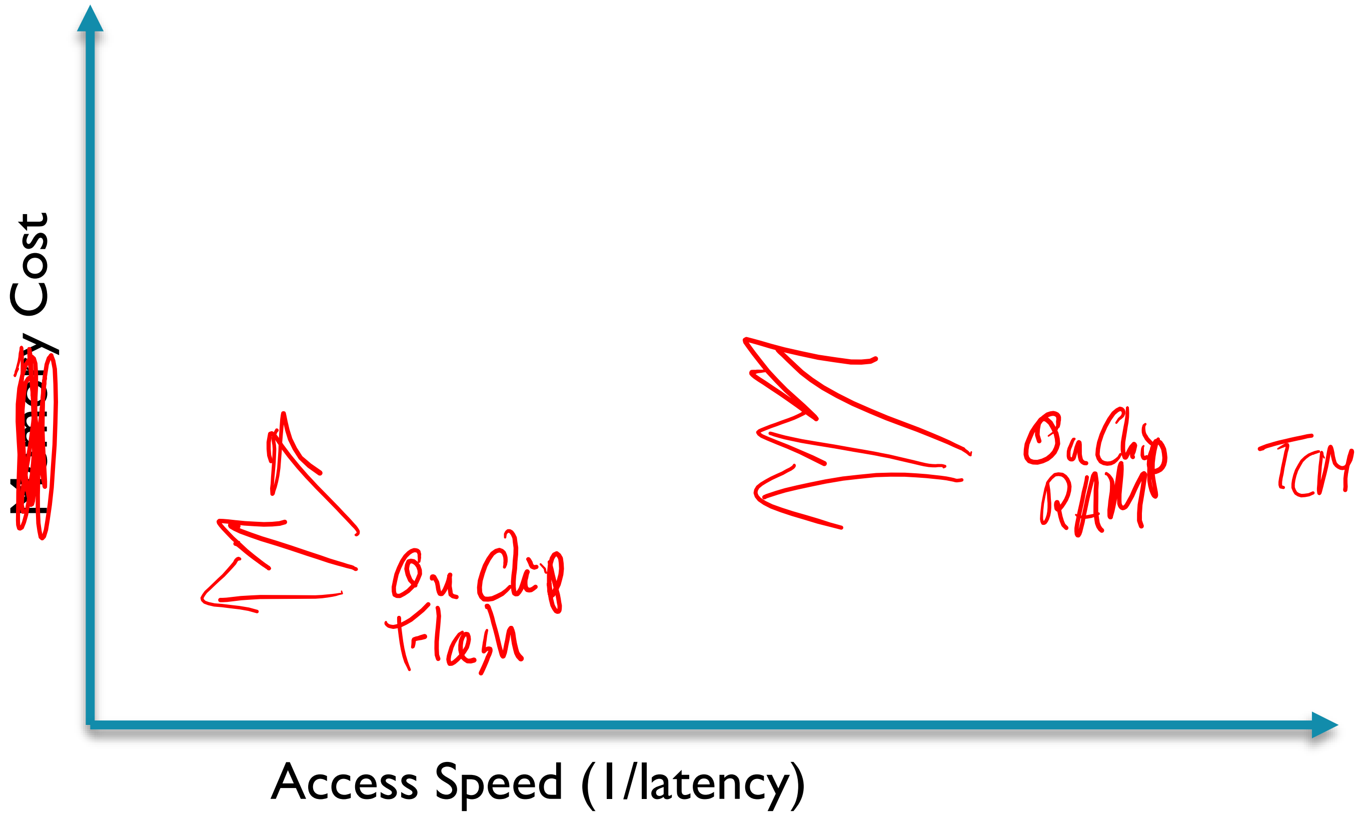
- MCUs typically have integrated memory
  - Flash ROM and RAM
  - Possibly EEPROM, FRAM
  - MCUs available with a variety of memory sizes
  - Price rises with increased memory
  - Can switch between MCUs in same family (with same peripherals)
- Pin-compatibility very important
  - Want a variety of MCUs with same package and footprint
  - Then don't need to redesign PCB and recertify it to change MCU



Memory Expansion Feature



- Memory expansion mode provides address and data buses to access external memory
  - Slower than on-chip memory
  - More pins required on MCU package
  - More complex PCB design



Motivation

- Why does memory size matter?
  - Software Gas Law: Over time, program expands to use all available resources (memory here).
  - Existing memory might be (nearly) full. Nearly full memory slows development.
  - MCUs with more memory (if available) cost more
  - Some systems have small sections of faster memory (cache, scratch-pad memory (TCM)).
    - Smaller memory requirements improve spatial locality, making caches and scratch-pads work better

- Which part of the program uses the most memory?
  - Start by shrinking that part

UNDERSTANDING MEMORY REQUIREMENTS

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,“Hello!”);
}
```

- Five possible types
  - Code
  - Read-only static data
  - Writable static data
    - Initialized
    - Zero-initialized
    - Uninitialized
  - Heap
  - Stack
- What goes where?
  - Code is obvious
  - And the others?

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,“Hello!”);
}
```

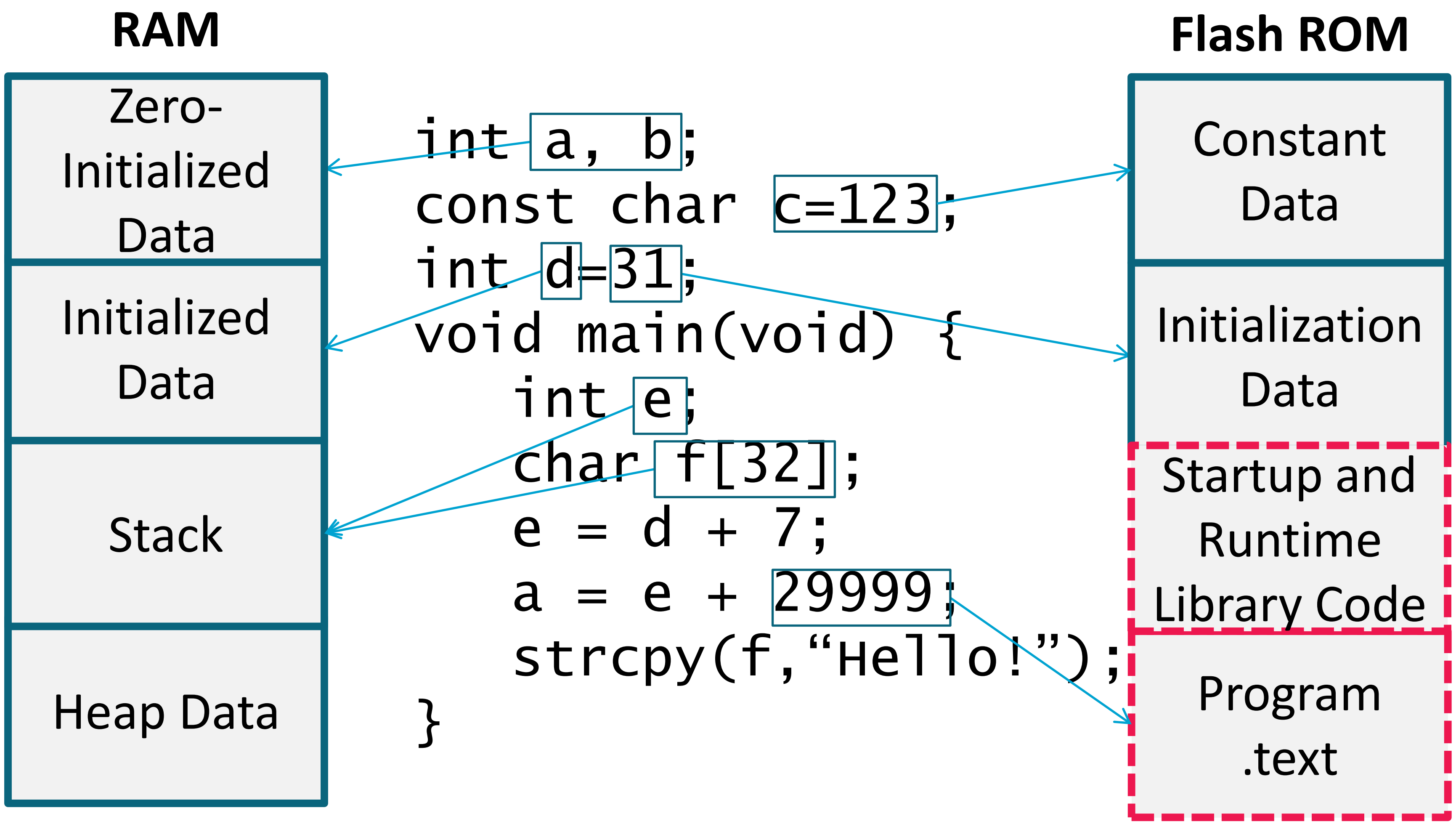
- Can the information change?
  - **No? Put it in read-only, nonvolatile memory**
    - Instructions
    - Constant strings
    - Constant operands
    - Initialization values
  - **Yes? Put it in read/write memory**
    - Variables
    - Intermediate computations
    - Return address
    - Other housekeeping data

What Memory Does a Program Need?

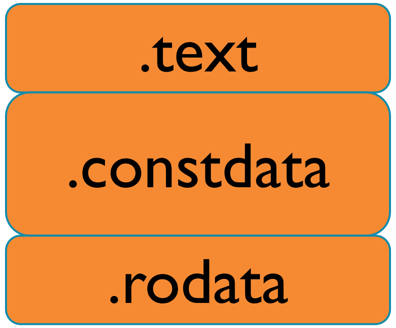
```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,"Hello!");
}
```

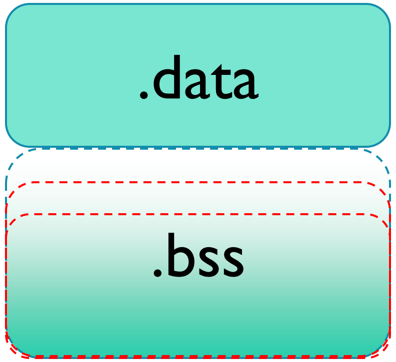
- How long must the data exist? Reuse memory if possible.
  - **Statically allocated**
    - Exists from **program start to end**
    - Each variable has its own fixed location
    - Space is not reused
  - **Automatically allocated**
    - Exists from **function start to end**
    - Space can be reused
  - **Dynamically allocated**
    - Exists from explicit allocation to explicit deallocation
    - Space can be reused

Program Memory Use



Executable File Sections



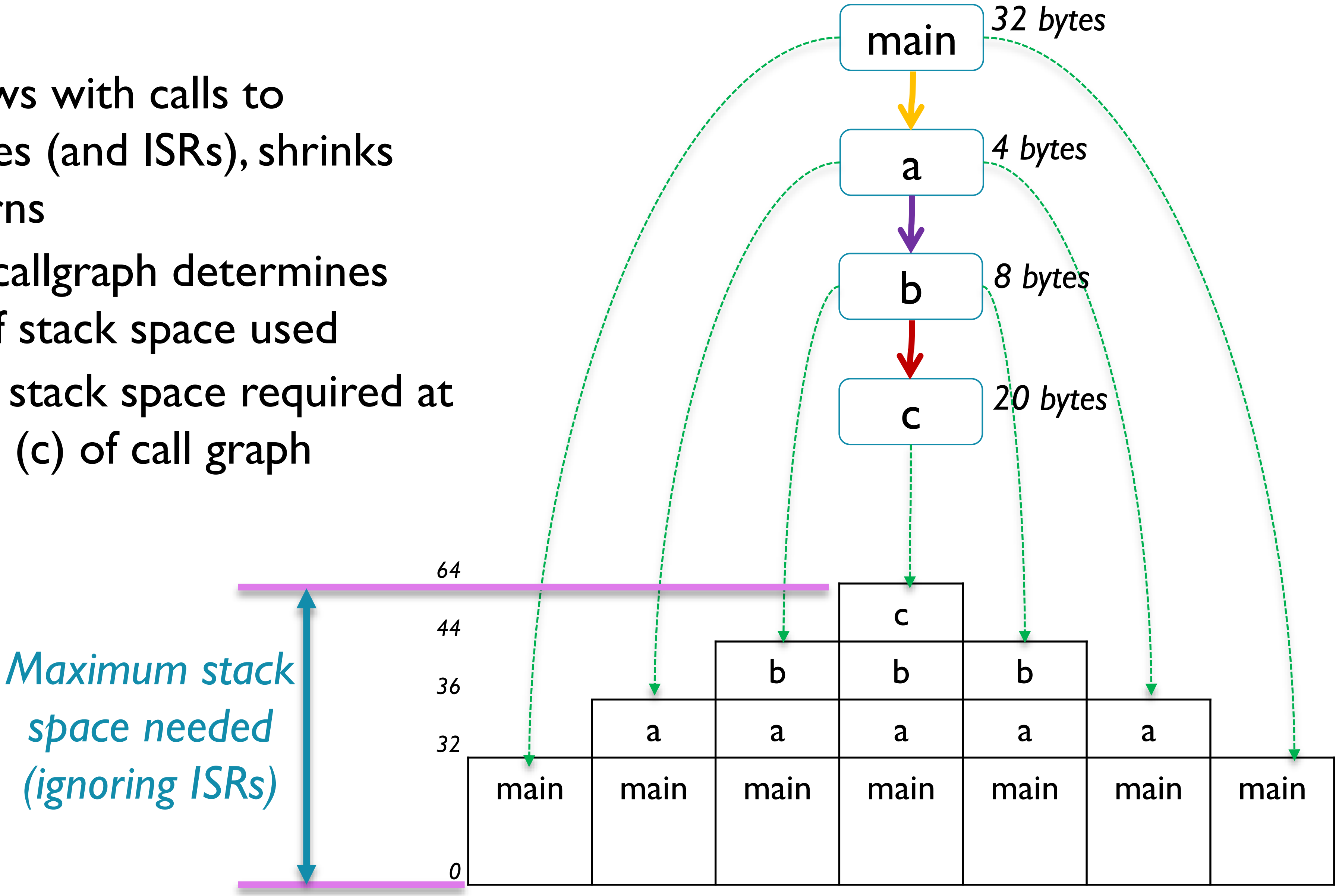


- ROM (RO)
  - **.text**
    - Program code (instructions)
    - Will not change, is read directly from ROM by program
    - Exact size known at build time
  - **.constdata**
    - Initialization data for variables
    - Is copied from ROM to RAM on system start-up
    - Exact size known at build time
  - **.rodata**
    - Read-only (const) data
    - Will not change, is read directly from ROM by program
    - Exact size known at build time

- RAM (RW, ZI)
  - **.data**
    - Holds variables which have been initialized
    - Is loaded from ROM to RAM on system start-up
    - Exact size known at build time
  - **.bss**
    - Uninitialized data, stack, heap
    - Is cleared to zero on system start-up before main() begins
    - **Exact size not known at build time** for non-trivial programs
      - Stack and heap growth depend on program behavior, input data
- So how much space to allocate for stack and heap?
  - Too little? Program may malfunction or crash
  - Too much? We waste precious RAM

Summary of Stack Memory Use

- Stack grows with calls to subroutines (and ISRs), shrinks with returns
- Depth in callgraph determines amount of stack space used
- Maximum stack space required at leaf nodes (c) of call graph



DATA ALIGNMENT AND PACKING

Data Alignment and Packing

- Naturally aligned data
  - N-byte object is located at address which is multiple of N
  - ARM architecture designed to be fast for accessing naturally aligned data, can access aligned word in memory with one operation

```
struct mystruct {
      char c;
      short s;
      … // next field
}
```

- What about a data structure (struct, union) with different field sizes?
  - Padding is used to align fields naturally

A char c	A+1 padding	A+2 short s	A+3
A+4 available	A+5 available	A+6 available	A+7 available

Data Alignment and Packing - ARMCC

- Use 1-byte alignment for all fields within structure/union using __packed qualifier
- ARMCC User Guide, Sections 4.30(++), 10.4, 15.10

```
struct mystruct {
      char c;
      __packed short s;
      … // next field
}
```

- Compiler will generate code which supports unaligned accesses
  - More code, slower

<table>
<tr><td>A
char c</td><td>A+1</td><td>A+2</td><td>A+3
available</td></tr>
<tr><td></td><td colspan="2">short s</td><td></td></tr>
<tr><td>A+4
available</td><td>A+5
available</td><td>A+6
available</td><td>A+7
available</td></tr>
</table>

Data Alignment and Packing – ARMCLANG (AC6)

A char c	A+1	A+2	A+3 available
	short s		
A+4 available	A+5 available	A+6 available	A+7 available

- Align all fields within structure/union to at most n bytes: `#pragma pack (n)`
- Pack all fields within structure/union by aligning to 1 byte: `__attribute__((packed))`
- Align specific field within structure/union to 1 byte: `__attribute__((packed))`
- ARMCLANG Arm Compiler for Embedded User Guide, Section 4.5
- Compiler will generate code which supports unaligned accesses
  - More code, slower

```
#pragma pack (1)
struct mystruct {
      char c;
      short s;
      … // next field
}
```

```
struct __attribute__((packed)) mystruct
{
      char c;
      short s;
      … // next field
}
```

```
struct mystruct {
      char c;
      short __attribute__((packed)) s;
      … // next field
}
```

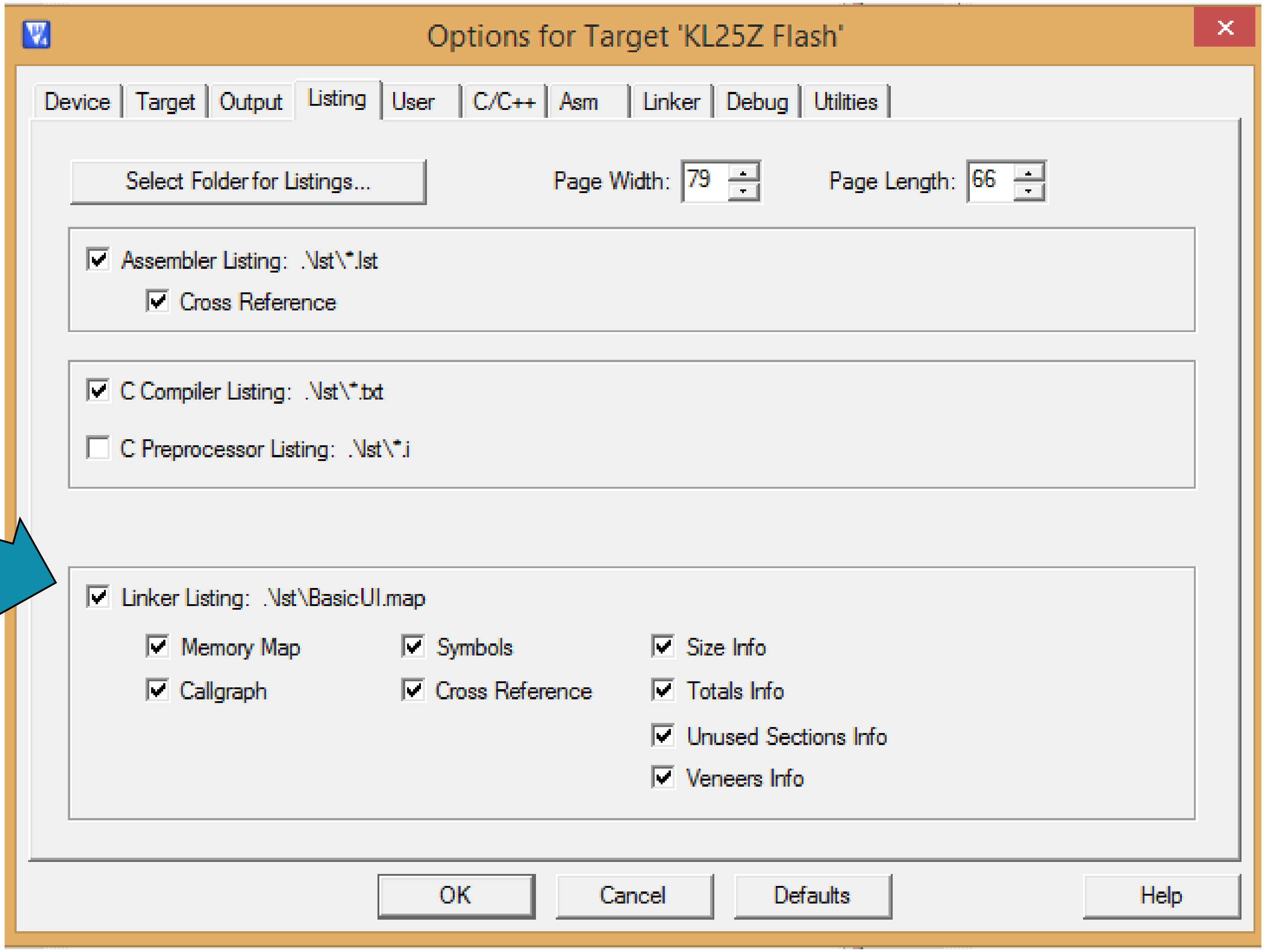
TOOLCHAIN SUPPORT FOR ANALYSIS OF MEMORY USE

Linker can Generate a "Map File"

- Useful memory information

- Select Project Options
- Select Listing tab
- Check Linker Listing and all checkboxes in section

- Two files created
  - .\lst*.map – text file with almost all this information
  - .\obj*.htm – HTML page with static call graph



Map File Contents

```
Global Symbols

Symbol Name                              Value     Ov Type        Size  Object(Section)
SystemInit                               0x000000f1   Thumb Code   164  system_mkl25z4.o(.text)
SystemCoreClockUpdate                    0x00000195   Thumb Code   310  system_mkl25z4.o(.text)
main                                     0x000002fd   Thumb Code   124  main.o(.text)
Delay                                    0x00000385   Thumb Code    24  delay.o(.text)
__aeabi_uidiv                            0x000003a1   Thumb Code     0  uidiv.o(.text)
__aeabi_uidivmod                         0x000003a1   Thumb Code    44  uidiv.o(.text)
__aeabi_i2d                              0x000003cd   Thumb Code    34  dflti.o(.text)
__I$use$fp                               0x000003f5   Thumb Code     0  iusefp.o(.text)
__scatterload_null                       0x000003f5   Thumb Code     2  handlers.o(i.__scatterload_null)
__mathlib_zero                           0x000003f8   Data           8  qnan.o(.constdata)
Init_RGB_LEDs                            0x00000411   Thumb Code   124  leds.o(.text)
Control_RGB_LEDs                         0x0000048d   Thumb Code    70  leds.o(.text)
i2c_init                                 0x000004e5   Thumb Code    74  i2c.o(.text)
i2c_start                                0x0000052f   Thumb Code    26  i2c.o(.text)
```

- Map file shows how memory is used
  - Symbol table, memory map, image component sizes
- We might care about function and data sizes

NC STATE UNIVERSITY

Per-Module Information

.text | .rodata .constdata in ROM | .data | .bss in RAM

Code	(inc. data)	RO Data	RW Data	ZI Data	Debug	Object Name
28	4	0	0	0	764	delay.o
604	12	0	0	0	1559	i2c.o
212	18	0	0	0	773	leds.o
136	12	0	0	0	479	main.o
528	36	0	16	0	1443	mma8451.o
44	24	192	0	1024	840	startup_mkl25z4.o
552	60	0	4	0	5505	system_mkl25z4.o

Code	(inc. data)	RO Data	RW Data	ZI Data	Debug	Library Member Name
544	70	152	0	0	112	atan.o
376	40	0	0	0	144	atan2.o
20	6	0	0	0	68	dunder.o
44	6	0	0	0	60	fpclassify.o
172	0	0	0	0	76	poly.o
0	0	8	0	0	0	qnan.o
72	6	0	0	0	76	sqrt.o

- Includes both **compiled** modules and **library** modules which were linked in
- Includes padding and data in code segment

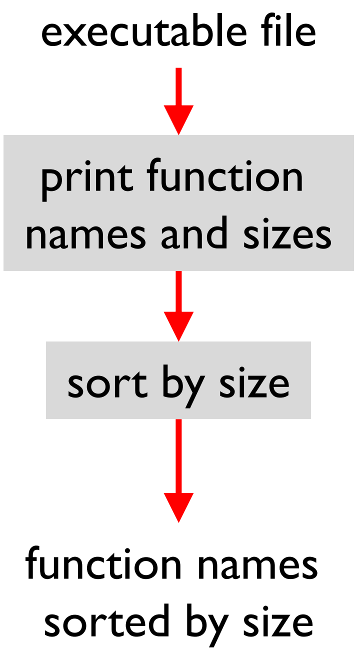
Summary Size Information

```
      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      5440        342        384         24       1024      12103   Grand Totals
      5440        342        384         24       1024      12103   ELF Image Totals
      5440        342        384         24          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 5824 (   5.69kB)
    Total RW  Size (RW Data + ZI Data)              1048 (   1.02kB)
    Total ROM Size (Code + RO Data + RW Data)       5848 (   5.71kB)

==============================================================================
```

- ELF image includes zero-initialized data and debug information, not included in ROM

Automation!

- Want to create a sorted list of function sizes
- Possible approaches
  - Process map file – depends on text format of map file, may change if linker changes
  - Process axf file – is in ELF format, will not change. More stable approach
- Tools needed
  - Program to analyze axf file and determine function sizes – use custom "getregions" program built on libelf code
  - Program to sort functions by size – use DOS "sort"
- Use a batch file to do this in one step
  - find_sizes.bat

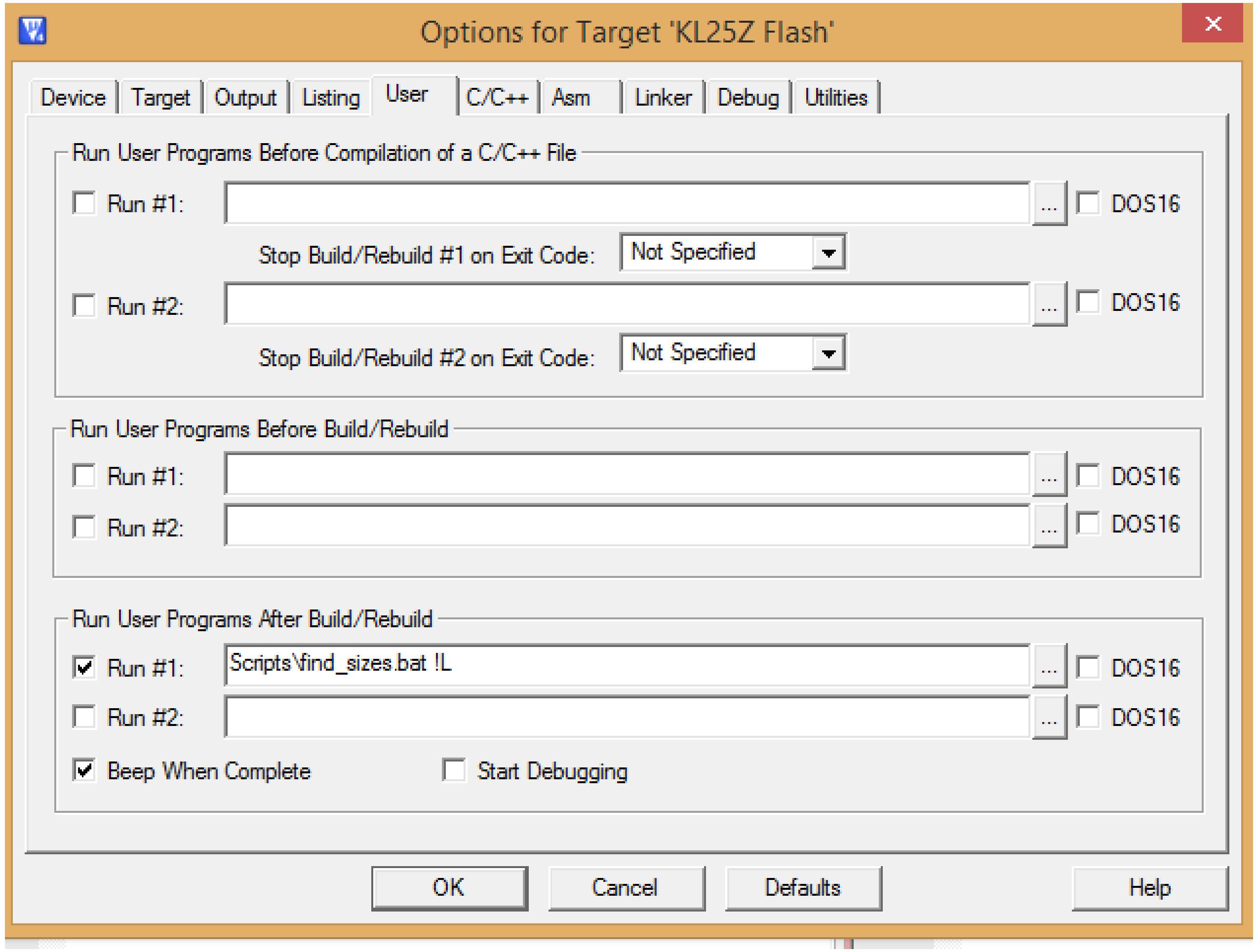


find_sizes.bat

```
@echo off
REM This script runs by  MDK, so is in <Project>
set PATH=%PATH%;..\..\Tools\GetRegions
@echo on
GetRegions.exe %1 -z -oObjects\function_sizes.txt
sort /R Objects\function_sizes.txt /O Objects\sorted_function_sizes.txt

```

More Automation

- Have MDK run the find_sizes.bat script every time the program is built



Results

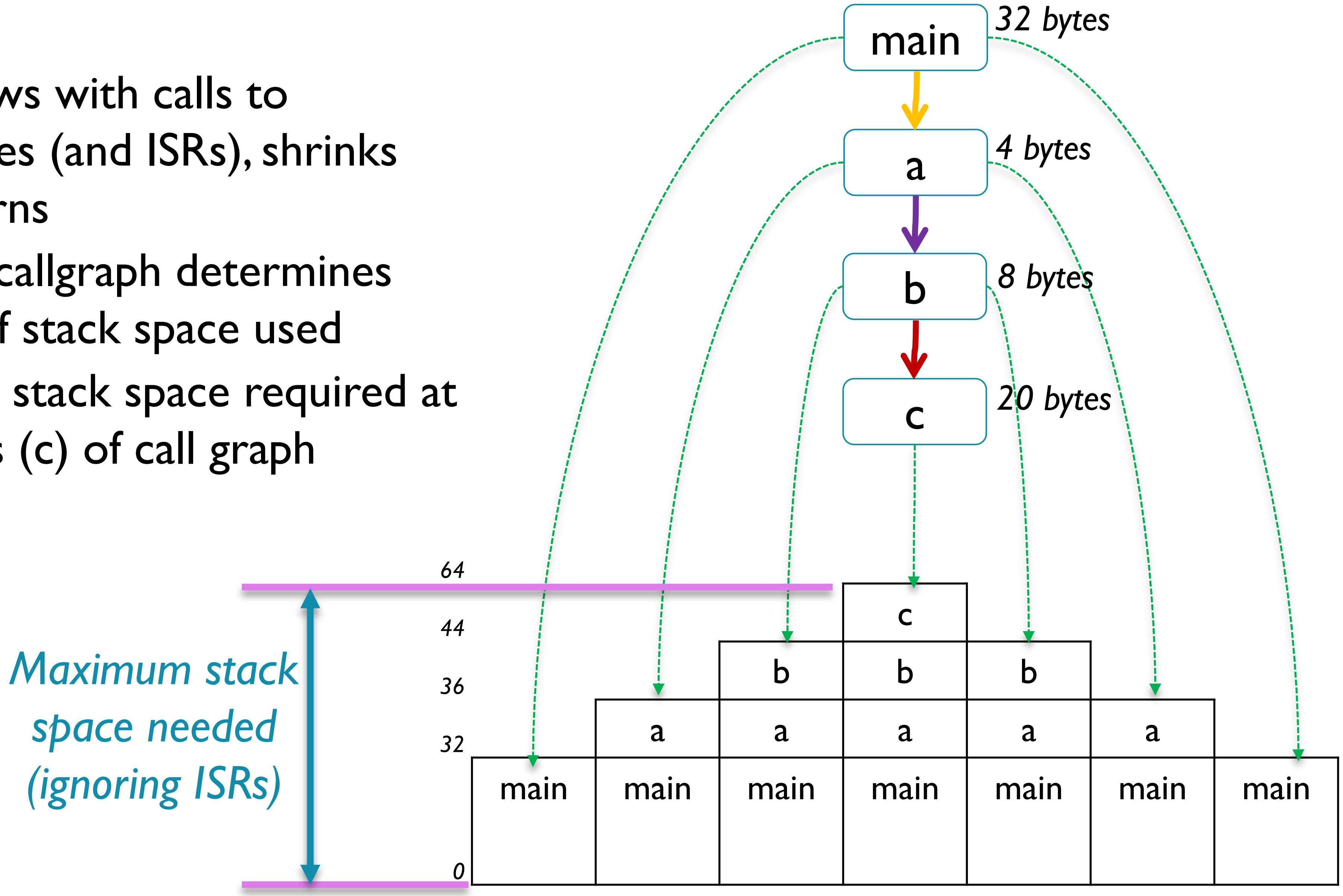
- Now we know what's using the most ROM
- We'll start at the top of the list and work our way down

```
00000474 atan
00000336 atan2
00000326 __aeabi_dadd
00000310 SystemCoreClockUpdate
00000260 convert_xyz_to_roll_pitch
00000234 __aeabi_ddiv
00000202 __aeabi_dmul
00000188 i2c_read_byte
00000172 __kernel_poly
00000166 _double_epilogue
00000164 SystemInit
00000162 _dsqrt
00000162 __aeabi_fadd
00000136 read_full_xyz
00000124 main
00000124 Init_RGB_LEDs
00000122 __aeabi_fmul
00000118 i2c_write_byte
00000116 _float_epilogue
00000106 i2c_read_setup
00000080 i2c_repeated_read
00000074 i2c_init
00000070 Control_RGB_LEDs
```

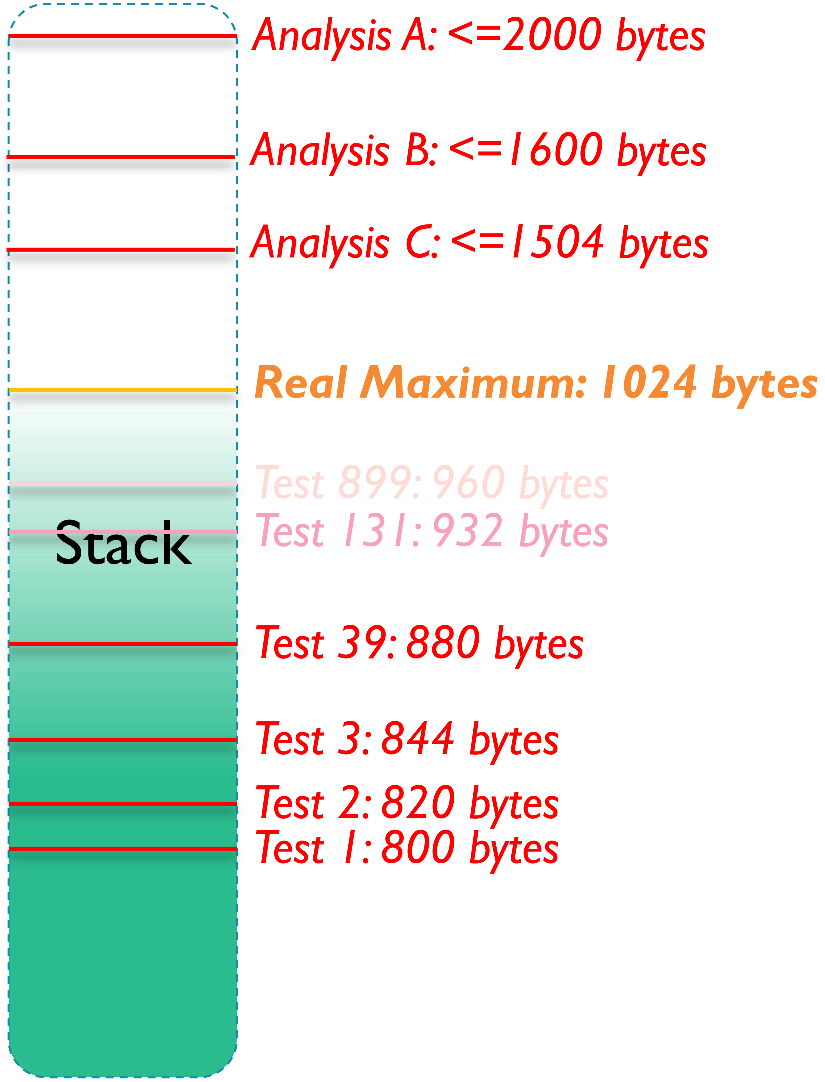
STACK SIZE ANALYSIS

Summary of Stack Memory Use

- Stack grows with calls to subroutines (and ISRs), shrinks with returns
- Depth in callgraph determines amount of stack space used
- Maximum stack space required at leaf nodes (c) of call graph



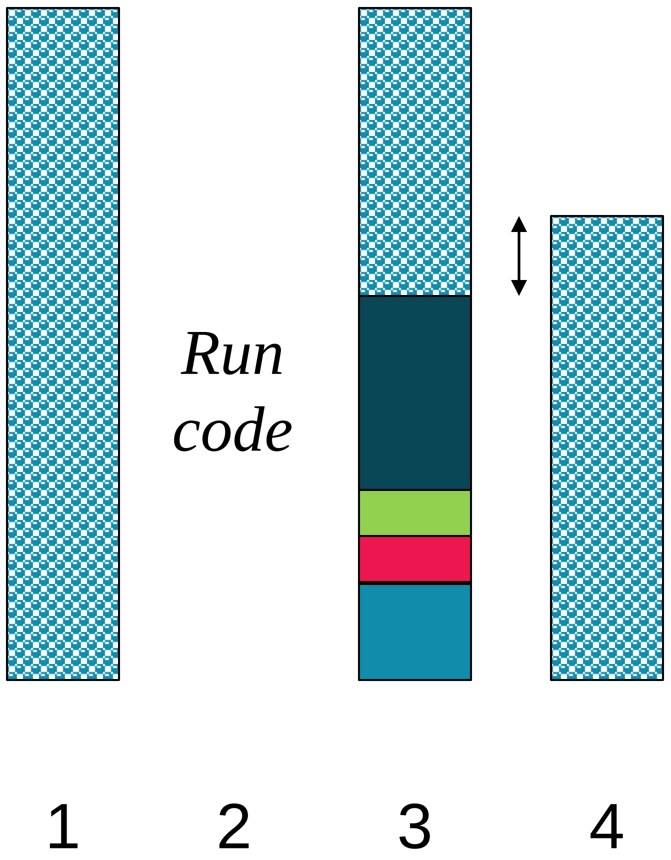
Stack Memory Size Estimation



- Two approaches to find maximum stack size
- 1. Run program, measure stack space used
  - Tests may not actually trigger worst-case stack usage, so ***unsafe***
  - Typically add a safety margin
- 2. Analyze program without running it
  - Based on analysis of function and handler call graphs
  - Some linkers provides this information
  - Typically overestimates maximum stack usage, so ***safe*** but maybe ***too conservative***

Experimental Stack Memory Size Estimation

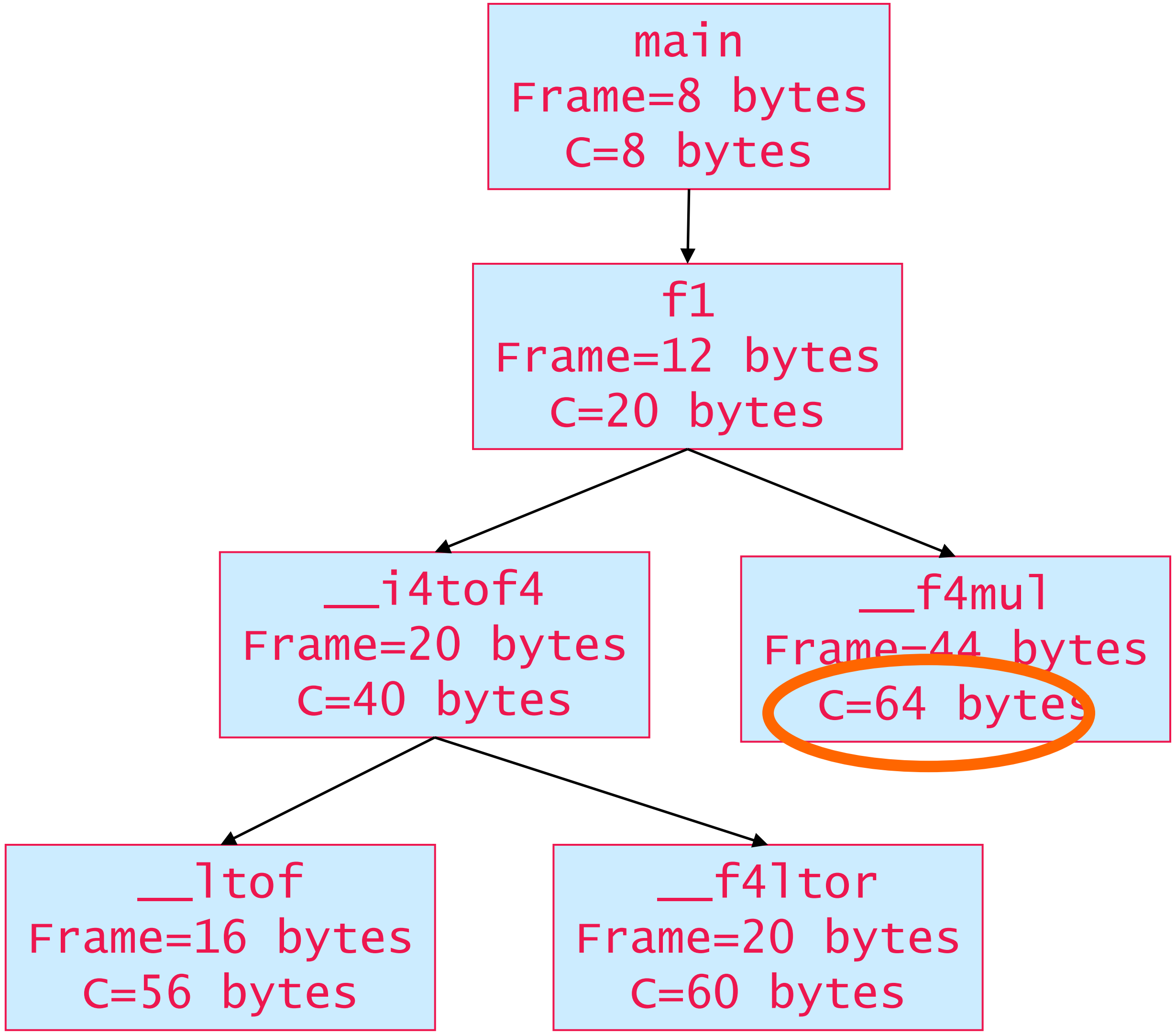
- Experimental measurement
  1. Preload stack memory space with known pattern
  2. Run program with many different test cases
     - Want to increase likelihood of hitting maximum stack depth
  3. Examine stack memory space to see how much was overwritten
  4. Use this measurement but add a safety margin (e.g. 20%)



- Can also have code perform this measurement at run-time
  - CMSIS-RTOS2 offers osThreadGetStackSpace(thread_id) if stack watermarking was enabled.

Analytical Stack Depth **Bounding**

- Find a number which is not less than the maximum stack memory possibly required
  - Smaller is better – less overestimation and wasted RAM
- Basic approach per thread
  - Determine stack frame size for each function and ISR/exception handler in program
  - Create call graph for main and each ISR/exception handler
  - For each call graph, find maximum stack depth of all leaf nodes (functions which cannot call any other functions)
- Then must put together thread, handler and RTOS information



Static Call Graph

- Text-based description of all possible function calling activity
  - Default output is html (./obj/*.htm)
  - See –callgraph option in ARM Linker User Guide
- Information per function F
  - Code size
  - Stack frame size
  - Maximum depth stack depth and call chain *starting with this function*
  - Callers: Functions which may call F
  - Callees: Functions which F may call
- Obstructions to call graph analysis
  - List of recursive functions
  - List of function pointers

Init_LCD (Thumb, 110 bytes, Stack size 16 bytes, lcd_4bit.o(.text))

[Stack]

- Max Depth = 64
- Call Chain = Init_LCD ⇒ lcd_write_cmd ⇒ wait_while_busy ⇒ lcd_read_status ⇒ Delay

[Calls]

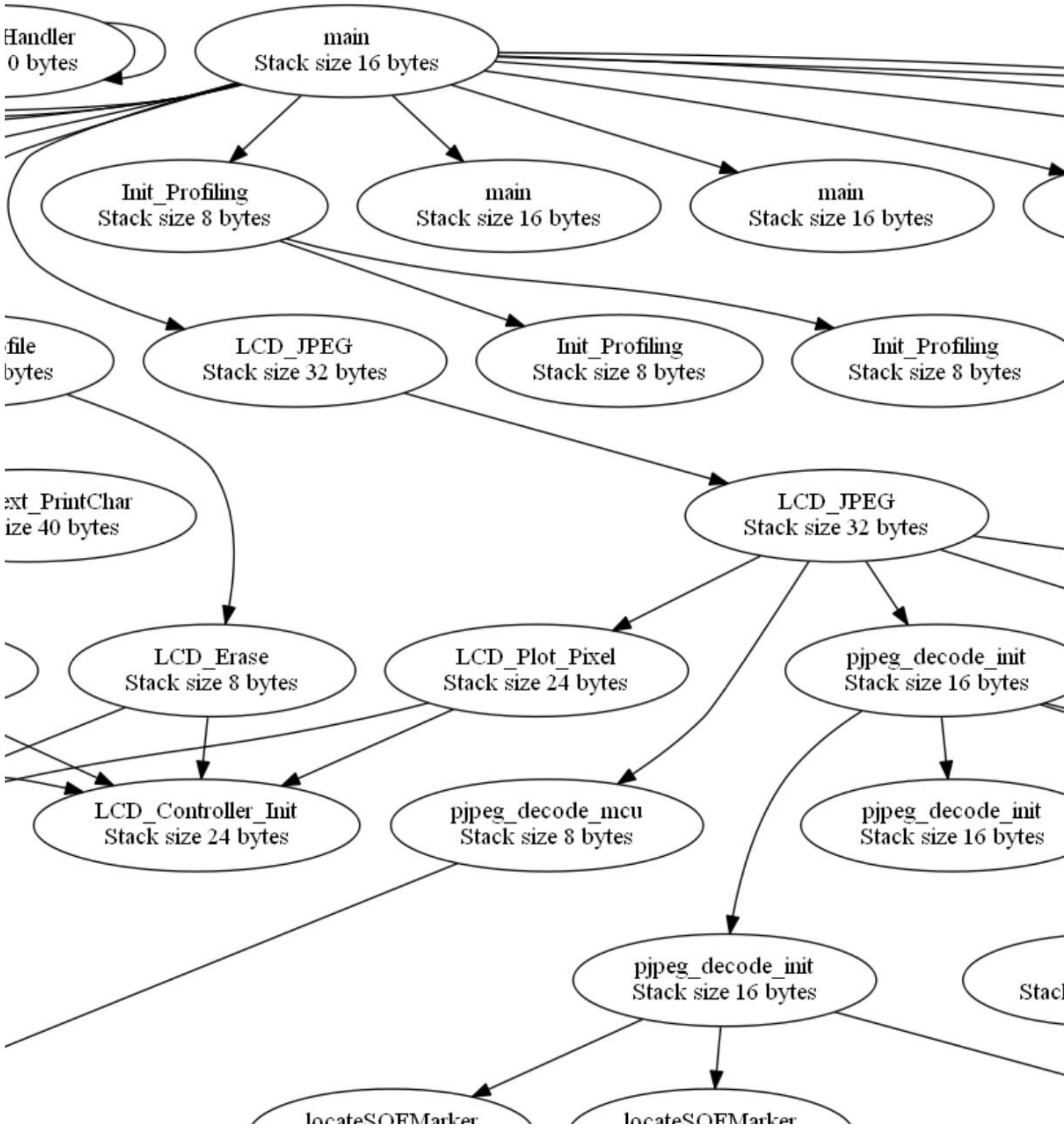
- >> Delay
- >> lcd_init_port
- >> lcd_write_cmd
- >> lcd_write_4bit

[Called By]

- >> main

Graphical Call Graph Generator Tool

- https://github.ncsu.edu/mjdargen/Keil-uVision-Call-Graph-Generator, by Michael Dargenio
- Python script
  - Converts .txt callgraph description from linker in Keil uVision to graph description file for GraphViz
  - GraphViz can generate png, svg outputs
- Options
  - Stack size per function
    - Include by uncommenting line outputfile.writelines(… %stack_size)
  - Library functions (_<function_name>)
    - To reduce graph size, ignores likely library functions
    - Can disable this if needed by removing tests (if ((caller.find(‘_’)!=0) …



Maximum Stack Depth Use – Main (non-threaded)

Static Call Graph for image .\obj\BasicUI.axf

#<CALLGRAPH># ARM Linker, 5.03 [Build 76]: Last Updated: Thu Mar 13 20:30:55 2014

Maximum Stack Usage = 132 bytes + Unknown(Cycles, Untraceable Function Pointers)

Call chain for Maximum Stack Depth:

main ⇒ calculate_roll ⇒ atan2f ⇒ __aeabi_fsub ⇒ __aeabi_fadd ⇒ _float_epilogue

- Maximum stack depth and corresponding function call chain
- **Unknown?**

Improving Stack Depth Bounding

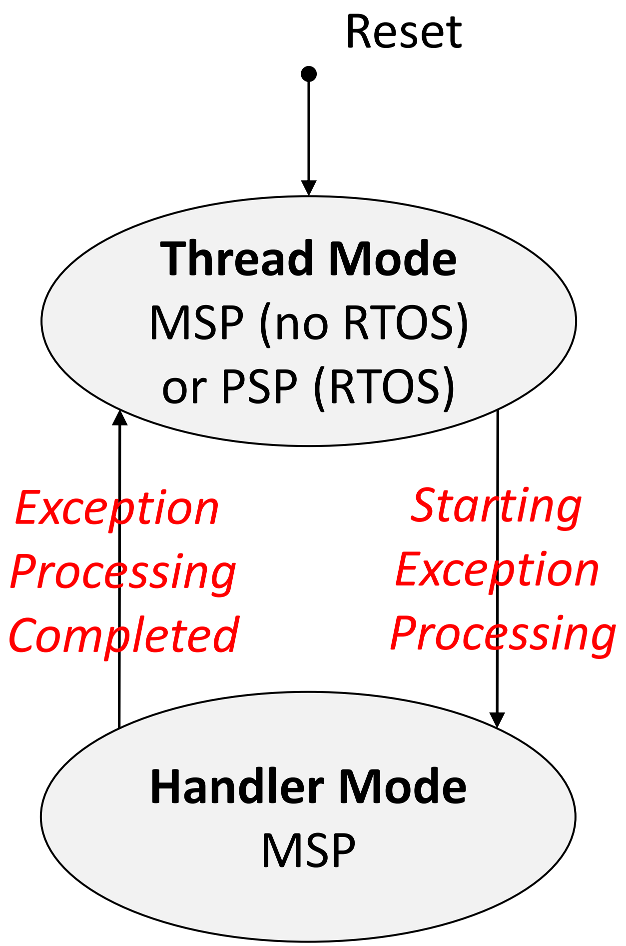
- Handle recursion and function pointers
  - Eliminate them from program
  - Modify program to help analyzer understand better
  - Manually adjust estimates based on ad-hoc analysis
  - Tell the analyzer limits to recursion, where function pointers go
- Eliminating infeasible paths
  - Analyze program control and data flow to prune call graph
  - Requires a quite bit of analysis
- Commercial stack-depth analysis tools provide these and other features

What about Interrupts?

- Follow same call graph analysis procedure for each ISR
- On entry, there will already be 32 bytes on stack due to hardware interrupt handling (pushing registers onto stack)
- Can any interrupt handlers be interrupted?
  - No: Worst case stack contribution is from only one handler (with largest stack frame)
  - Yes:
    - Which ISR(s) can be interrupted?
    - What are the interrupt priorities?
    - How often can these interrupts occur? Fast enough to interrupt a previously running ISR?
    - *Complexity: good reason to keep interrupts disabled in ISRs*

Which Stack? Main or Process?

- CPU has two stack pointers
  - Main Stack Pointer
  - Process Stack Pointer
- CPU can operate in two different modes
  - Handler mode for exception/interrupt handlers
  - Thread mode otherwise
- SPSEL flag (in CONTROL) selects SP for thread mode
  - On reset, SPSEL = 0
- SP refers to either MSP or PSP, depending on mode and SPSEL
  - Handler mode uses MSP
  - Thread mode
    - If SPSEL is 0, uses MSP
    - If SPSEL is 1, uses PSP. This means handlers use a different stack than threads.



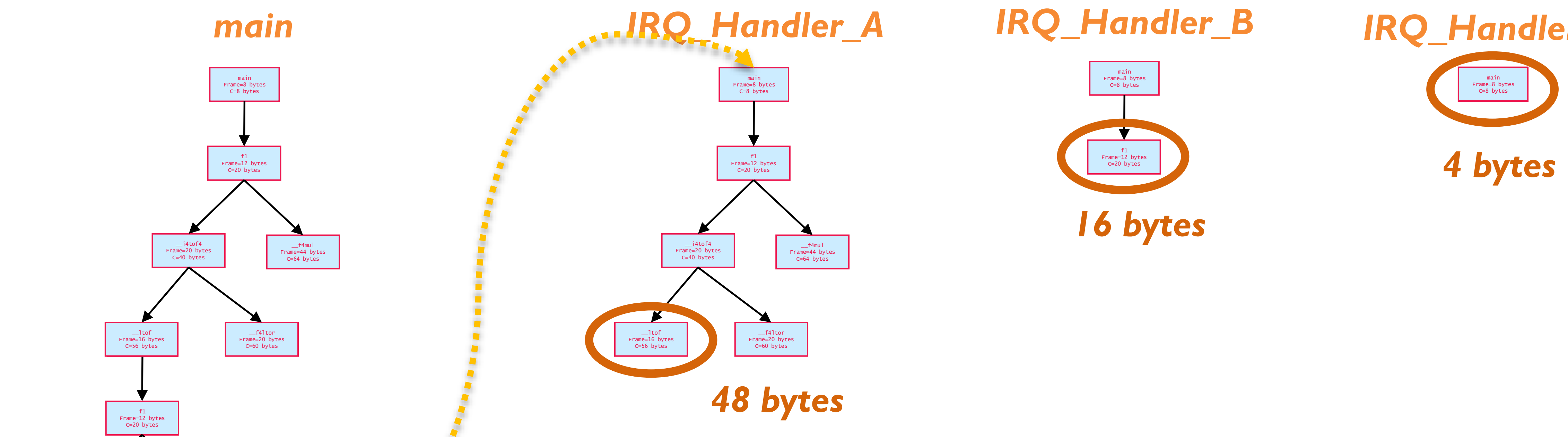
Handler Stack Usage without RTOS

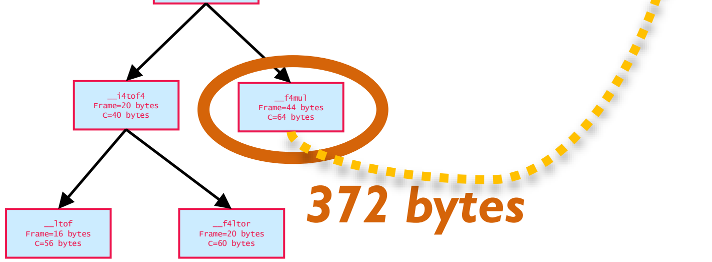
	Address Offset	Contents
	-8	Free space/Handler stack frame?
	-4	
MSP upon entering exception handler →	0	Saved R0
	+4	Saved R1
	+8	Saved R2
	+12	Saved R3
	+16	Saved R12
	+20	Saved LR
	+24	Saved PC
	+28	Saved xPSR
MSP before entering exception handler →	+32	Foo’s stack frame

- Only main stack pointer (MSP) is used

- CPU hardware automatically pushes 32 bytes onto main stack before executing handler (“register stacking”)
- Handler stack frame is added to main stack

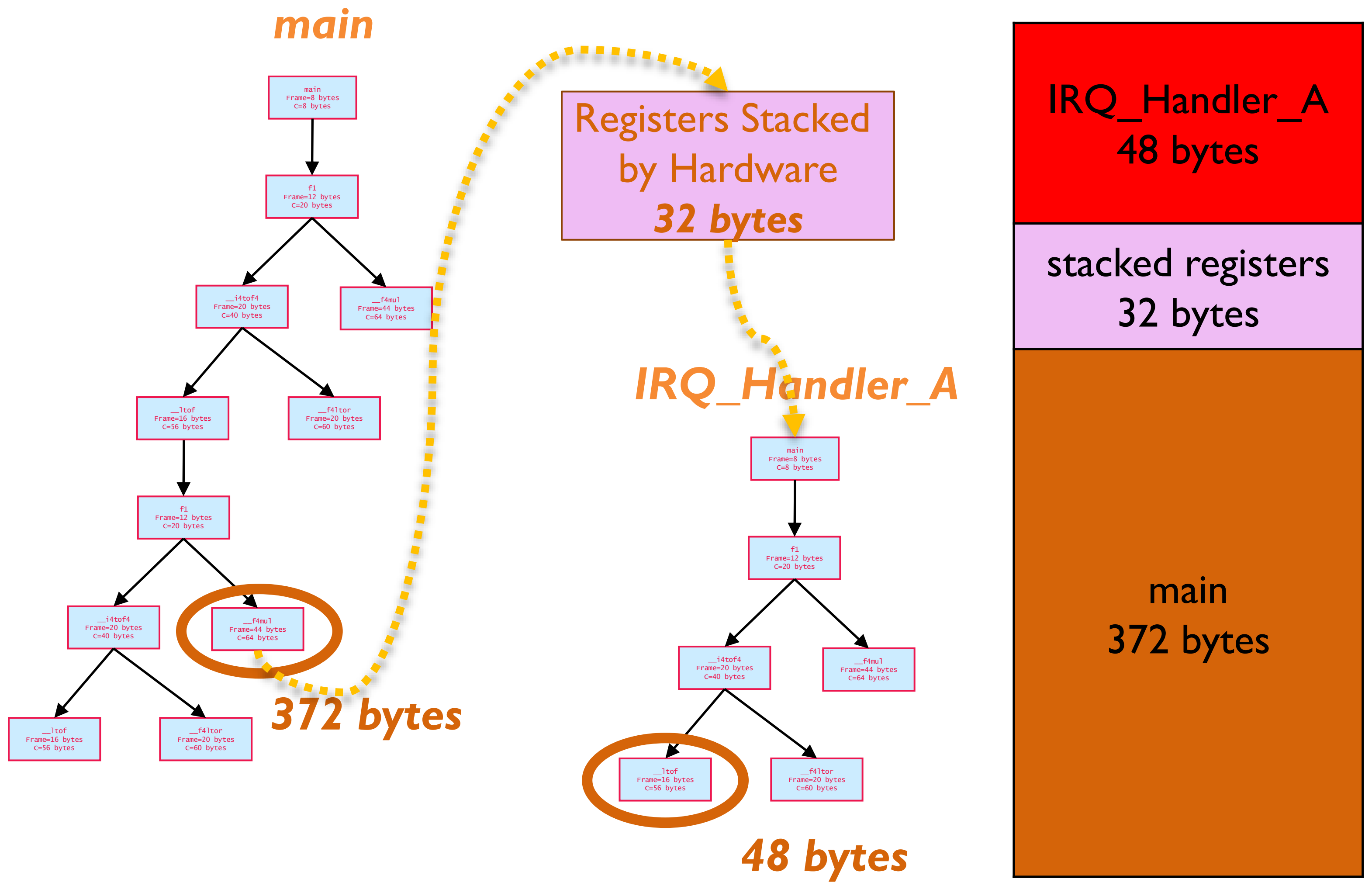
Stack Depth (MSP) for Single-Threaded System





- Overview of worst-case
  - Main is using its largest possible stack
  - IRQ with max. possible stack use occurs
  - Must include stacked registers (pushed by hardware)
- Details on next slide

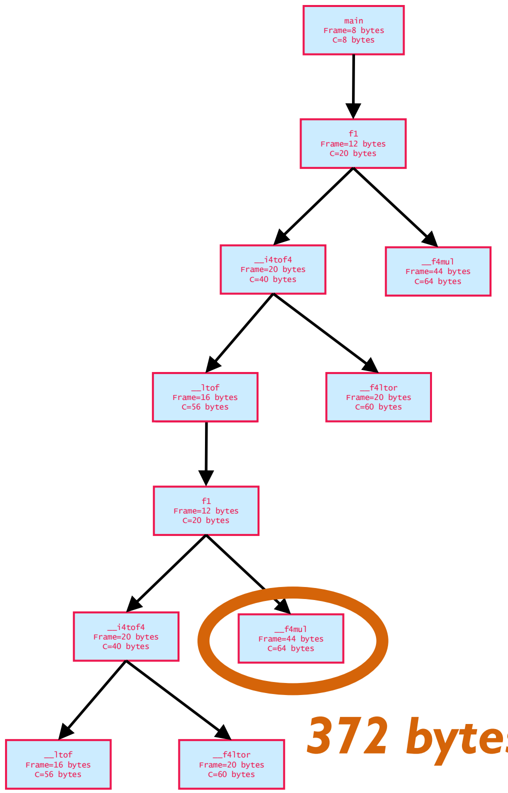
Maximum Stack Depth for Single-Threaded System (MSP Only)



- Bound on maximum program stack depth
  - main: 372 bytes
  - + Registers stacked by hardware in response to IRQ: 32 bytes
  - + largest IRQ Handler (A): 48 bytes
- Total: 452 bytes

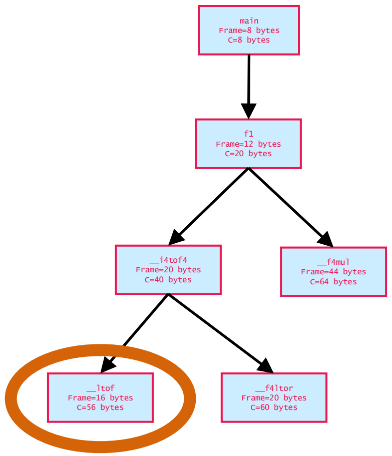
What if Interrupts can be Interrupted in Single-Threaded System?

main



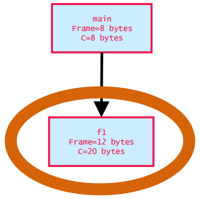
372 bytes

IRQ_Handler_A



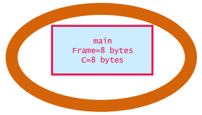
48 bytes

IRQ_Handler_B



16 bytes

IRQ_Handler_C

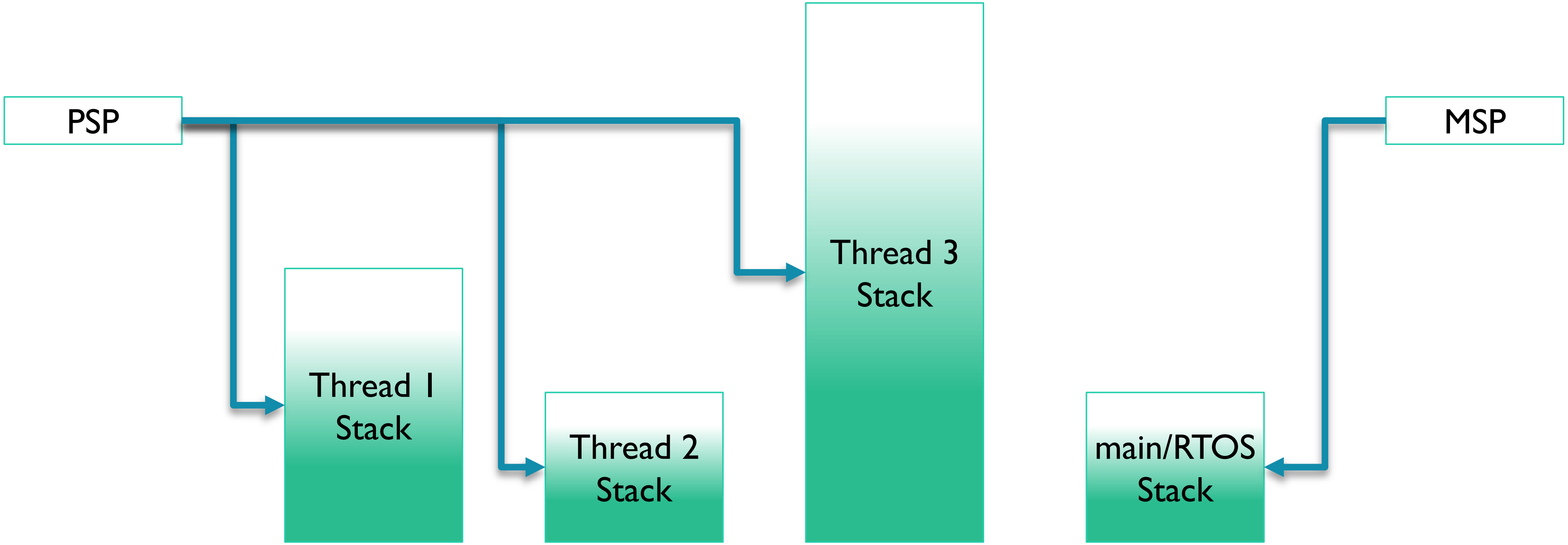


4 bytes

IRQ_Handler_A	48 bytes
stacked registers	32 bytes
IRQ_Handler_B	16 bytes
stacked registers	32 bytes
IRQ_Handler_C	4 bytes
stacked registers	32 bytes
main	372 bytes

- Need to consider…
  - Priority of interrupts: Assume A > B > C
  - Worst-case interrupt sequencing
    - Opposite to priority: C, B, A
  - Registers stacked only once due to tail-chaining
- Example: If no additional interrupt arrivals possible, max. stack depth = 472 bytes
- Even more complex if an IRQ can interrupt its own handler
  - Handler must be reentrant
  - Must consider arrivals of interrupts during service time (use response-time analysis)
  - Need timing constraints for worst case analysis

Stacks in Multi-Threaded Systems (with RTOS)



- Process SP (PSP) used for threads
- Main SP (MSP) used for RTOS

Handler Stack Usage with RTOS

Address Offset	Contents
0	Saved R0
+4	Saved R1
+8	Saved R2
+12	Saved R3
+16	Saved R12
+20	Saved LR
+24	Saved PC
+28	Saved xPSR
+32	Foo's stack frame

PSP upon entering exception handler → (offset 0)

PSP before entering exception handler → (offset +32)

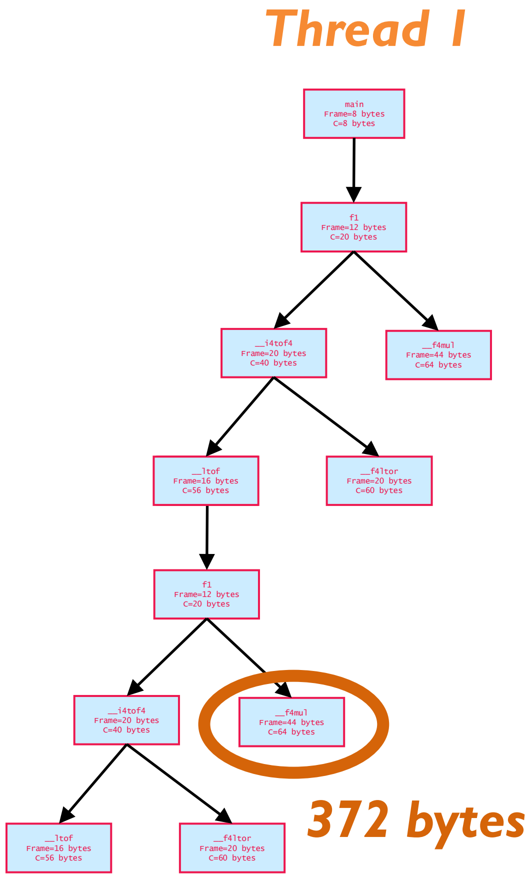
- Which SP?
  - PSP used for threads, and first level of interrupts
  - MSP used for RTOS and nested interrupts (second and deeper levels)
- Thread running Foo uses PSP
  - CPU automatically pushes 32 bytes onto process stack before starting executing handler

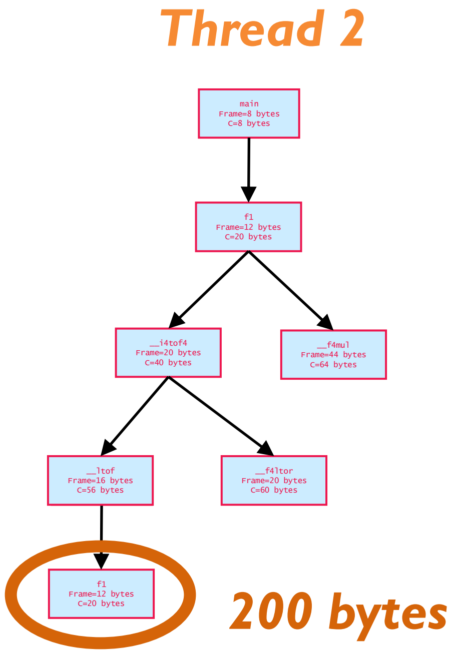
Address Offset	Contents
0	
+4	
+8	
+12	
+16	
+20	
+24	
+28	Free space becomes Handler's stack frame
+32	bar's stack frame

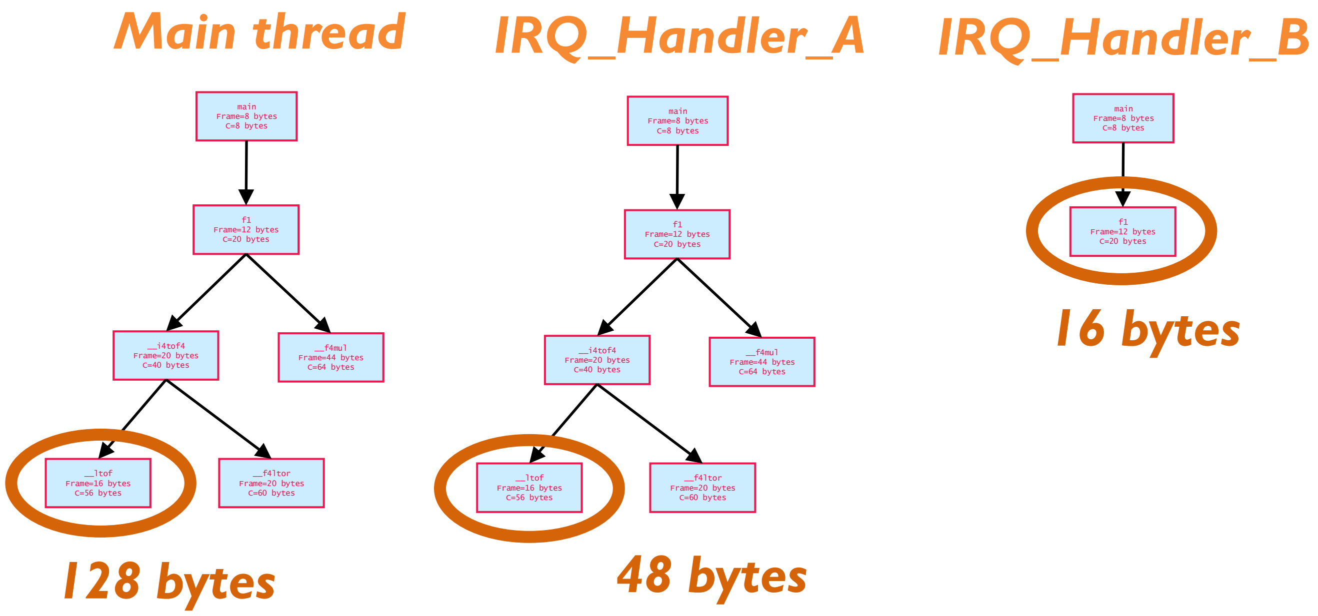
MSP before and upon entering exception handler → (offset +32)

- Handler uses MSP
  - Handler stack frame is added to main stack

Stack Depths for Multi-Threaded System



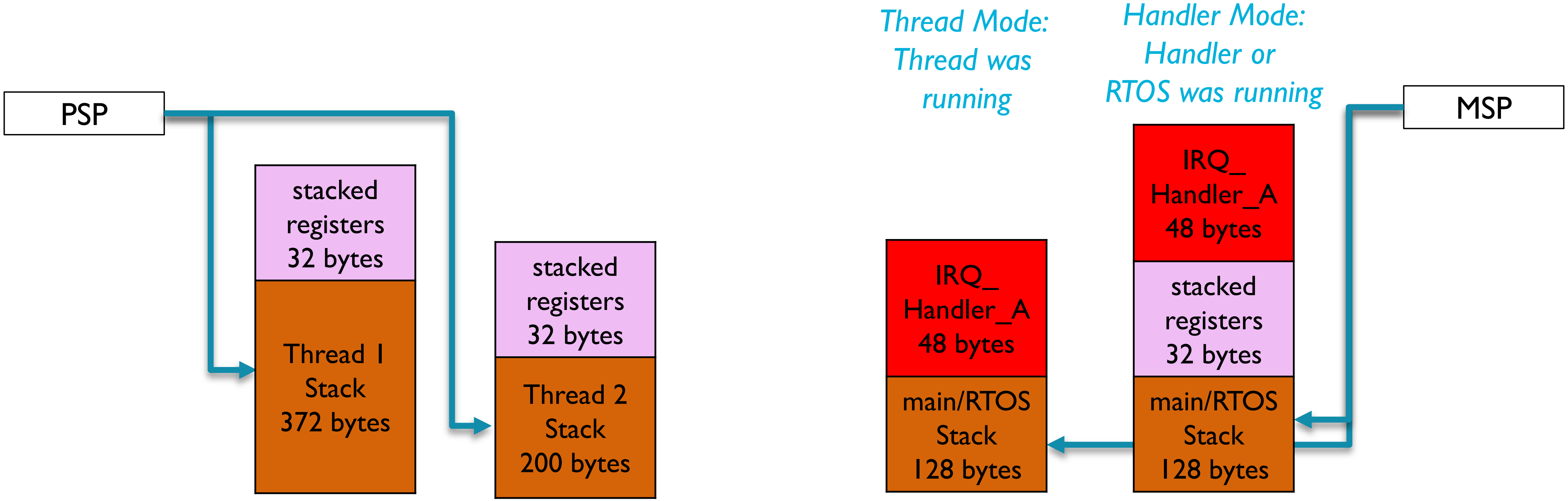




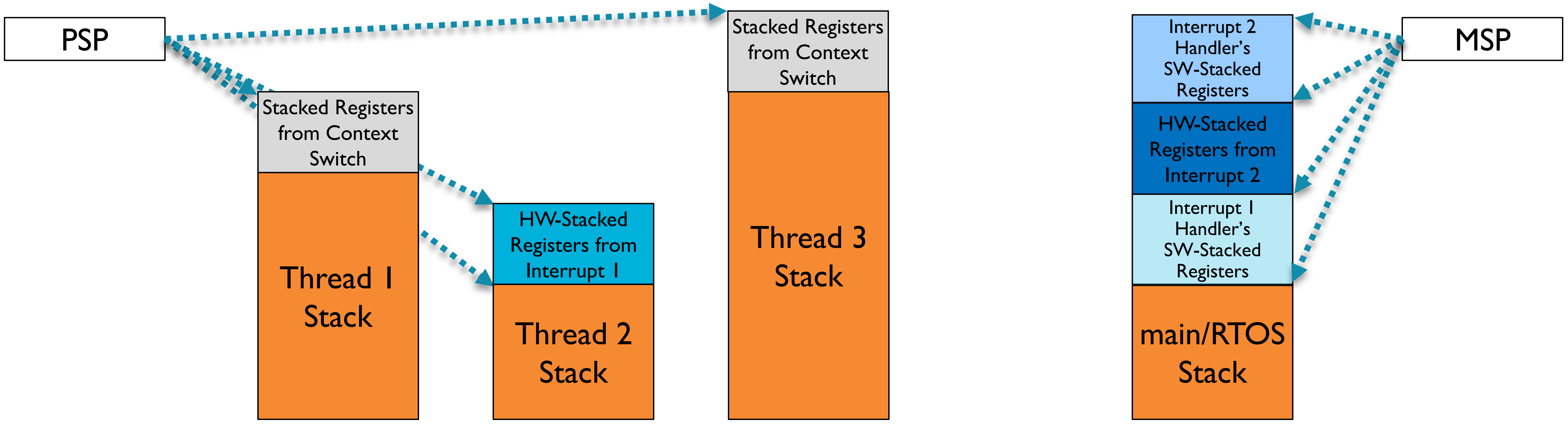
- Need maximum stack depth for main and each thread
- Determine bounds on each thread's maximum stack depth
  - Maximum stack depth
  - + Registers stacked for IRQ handler: 32 bytes
- Determine bound on maximum main stack depth
  - Main thread: 128 bytes
  - + Largest IRQ Handler (A): 48 bytes
  - + Registers stacked for IRQ handler from main thread: 32 bytes

Maximum Stack Depths without Nested Handlers

What was the CPU mode (what was it running) when the interrupt / exception was requested?



Nested Interrupt/Exception Handlers



- CPU is running Thread 2 in thread mode, using PSP as SP
- Interrupt 1 is requested
- CPU stacks some of Thread 2's registers via PSP onto Thread 2 stack
- CPU switches to handler mode, using MSP as SP
- CPU runs Interrupt 1 handler, using MSP for stack frame
- Interrupt 2 is requested, preempts Interrupt 1
- CPU stacks some of Interrupt Handler 1's registers via MSP onto main stack
- CPU runs Interrupt 2 handler, using MSP for stack frame

REDUCING READ-ONLY MEMORY REQUIREMENTS

Reducing ROM Requirements

- Conceptual Goals
  - Eliminate code for unneeded features
  - Implement needed code more efficiently (densely)

- Methods
  - Use language support
  - Configure compiler and toolchain better
  - Write better code with a better design
    - Rearchitect software to reduce duplicated or similar code
    - Often, seeing the details of the problem make solutions obvious

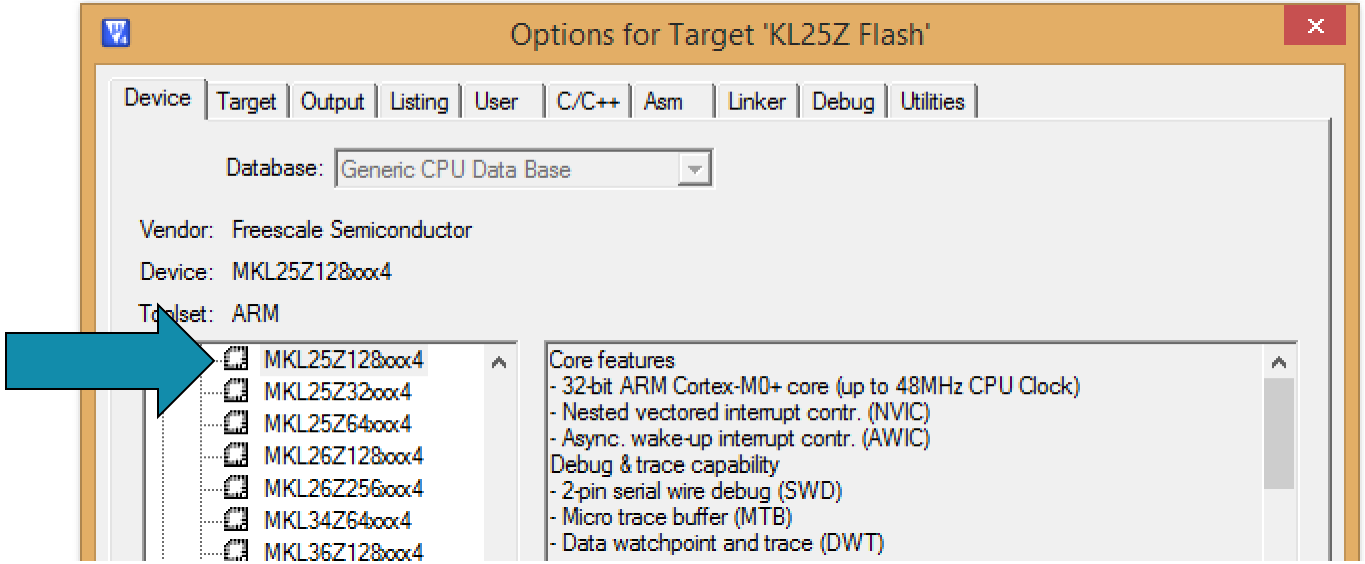
Use C Language Support

- Enable linker to delete unused code from module
  - Modify function declaration with **static** to indicate that no function outside of the declaring file will call that function

my_file.c

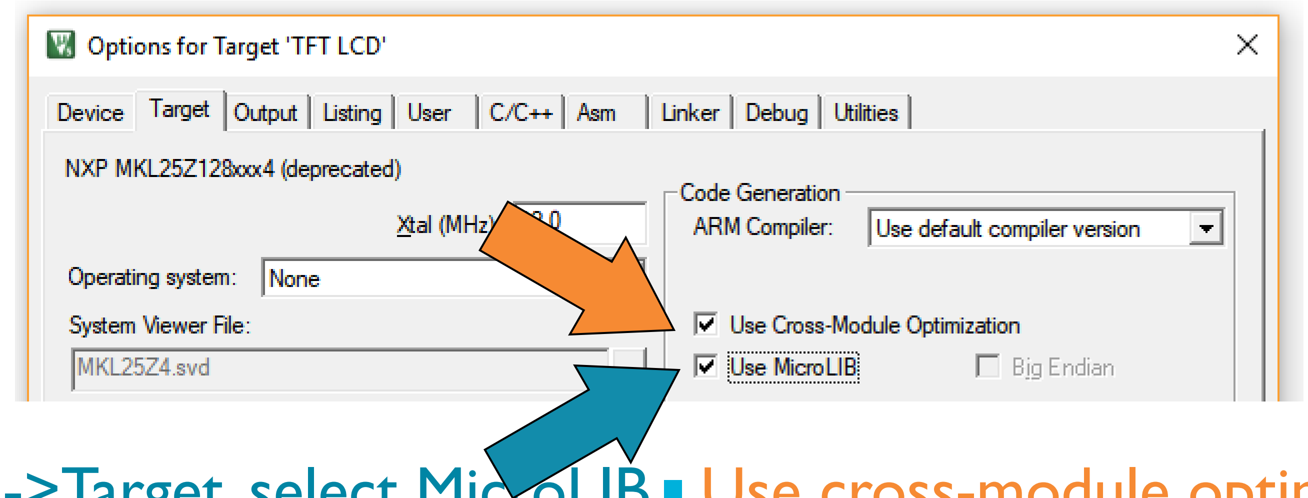
```
static void read_xyz(void)
{
	// sign extend byte to 16 bits.
	// need to cast to signed since function
	// returns uint8_t which is unsigned
	acc_X = (int8_t) i2c_read_byte(MMA_ADDR, REG_XHI);
	Delay(100);
	acc_Y = (int8_t) i2c_read_byte(MMA_ADDR, REG_YHI);
	Delay(100);
	acc_Z = (int8_t) i2c_read_byte(MMA_ADDR, REG_ZHI);
}
```

Configure Compiler and Toolchain



- Select correct processor type, or else
  - Compiler may think you have an older processor core…
  - … which lacks certain instructions (but which actually exist in this core) …
  - … which leads the compiler to link in library code to implement that functionality …
  - … even though the core supports it with efficient, native instructions!

Configure Compiler and Toolchain



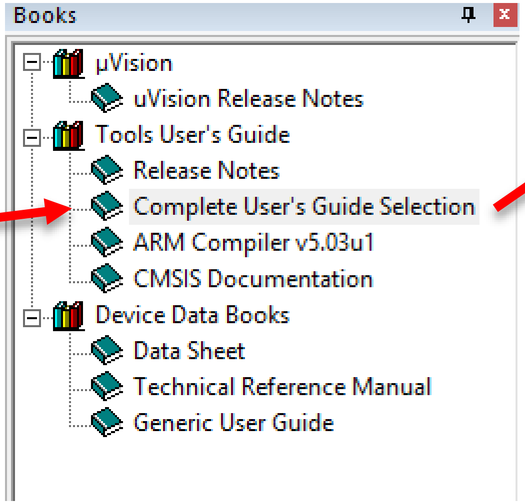
- In Project Options->Target, select MicroLIB
  - Highly optimized for code size, unlike default C library
  - Warnings
    - Not ISO C-compliant, some C features missing or slower
    - No reentrant function variants
    - No mutex locks provided
    - Floating point code handles denormalized values differently from IEEE 754 standard
    - No OS interaction functions provided
    - No wide characters or multibyte strings
    - No file I/O support, just stdin, stdout and stderror.
    - I/O streams are unbuffered
- Use cross-module optimization
  - Uses multiple passes to build program, using feedback to improve performance

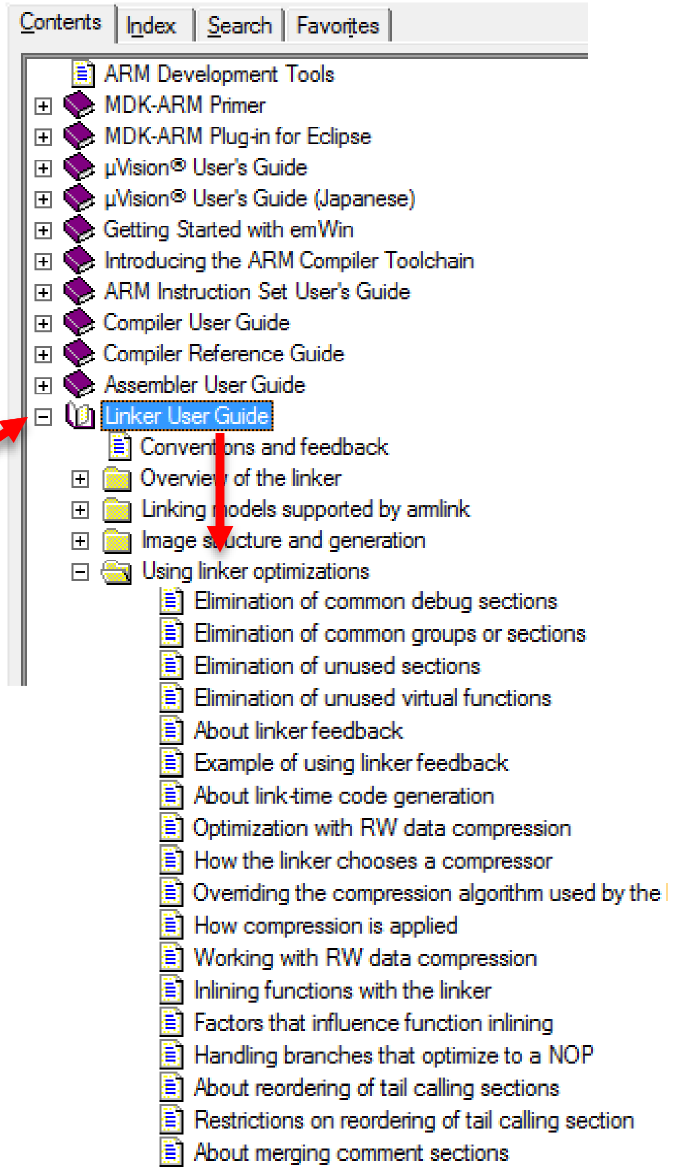
Linker Basics

- Basics of sections
  - **Input Section:** For each source module a compiler processes, it generates up to three input sections
    - RO – read-only
    - RW – read/write
    - ZI – zero-initialized
  - **Output Section:** Linker creates an output section by contiguously joining multiple input sections of the same type (RO, RW, or ZI)
  - **Region:** Contiguous sequence of one, two or three output sections
  - **Program Segment:** Contains one region
- Main linker optimization is elimination of unused/unreachable sections
  - Linker can only remove completely unused input sections
  - So help linker identify and delete these sections

Use Linker Options and Optimizations

- Many options available, well-documented
- See Linker User Guide
  - MDK ARM-> Help->Open Books Window
  - Tools User's Guide->Complete User's Guide Selection
  - Linker User Guide-> Using Linker Optimizations





Linker Options

- Linker feedback option – used in cross-module optimization
  - Use `--feedback file` on compiler and linker command lines
  - Linker creates text file naming the unused and inlined functions
  - Re-compile. Compiler reads the file and rebuilds objects, moving functions into own sections
  - Re-link. Linker can remove functions in own sections.
- Function inlining
  - Controlled with `--inline, --no_inline`
  - Replaces calls to a small function with copies of the called function
  - Will save time
  - May or may not save memory – depends on function body size and work needed to set up arguments and activation record
- Help the linker by adding *compiler* option `--split_sections`
  - Makes compiler generate one section per function in source file
  - Linker can then remove unused sections
- Allow linker to compress initialization (RW) data sections
  - ARMlink enables compression by default, applies it if it will reduce total size
  - ARMlink supports run-length compression (runs of repeated bytes) and Limpel-Ziv 77 compression (repeated phrases in buffer)

Rearchitect Software to Remove Similar or Identical Code

```
if(DifferentTalker)
   sprintf(buffer, "$APRMC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,”
    “W,%05.1f,%04.1f,%06ld,%05.1f,W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
else if(DifferentSenType)
   sprintf(buffer, "$GPGLC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,”
   ”W,%05.1f,%04.1f,%06ld,%05.1f,W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
else if(IllegalInField)
   sprintf(buffer, "$GPRMC,%02d%02d%02d,A,%02d%06.3fa,N,%03d%06.3f,”
   ”W,%05.1f,%04.1f,%06ld,%05.1f,W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
else if(IllegalAsField)
   sprintf(buffer, "$GPRMC;%02d%02d%02d;A;%02d%06.3f;N;%03d%06.3f;”
   ”W;%05.1f;%04.1f;%06ld;%05.1f;W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
```

- How can we improve this?

Use a Table

- Everything is the same except for the format strings
- So put them in a table.

```
char Formats[13][] = {
"$APRMC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,W,%05.1f,%04.1f,%06ld,%05.1f,W*“,
"$GPGLC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,W,%05.1f,%04.1f,%06ld,%05.1f,W*“,
"$GPRMC,%02d%02d%02d,A,%02d%06.3fa,N,%03d%06.3f,W,%05.1f,%04.1f,%06ld,%05.1f,W*“,
"$GPRMC;%02d%02d%02d;A;%02d%06.3f;N;%03d%06.3f;W;%05.1f;%04.1f;%06ld;%05.1f;W*",
   … (deleted) …
};

if(DifferentTalker)
  format_num = 0; //error in talker id - not gps
else if(DifferentSenType)
  format_num = 1; //error in sentence type - not gll
else if(IllegalInField)
  format_num = 2; //letter in field
else if(IllegalAsField)
  format_num = 3; //illegal separator
sprintf(buffer, Formats[format_num],
    hr, min, sec, lat_deg, lat_min, lon_deg, lon_min, speed, track, date, var);
  …
```

But Don't Always Use Tables

- Example: Floating point sine function has multiple possible implementations
  - Standard C math library
  - Lookup table (LUT)
  - Polynomial approximation
- Different memory requirements for each - Which is smallest?
  - **Standard C math library**
    - Fixed size - examine map file to determine
  - **Lookup table**
    - RO Size = (resolution needed) * (range of input values) * (element size)
    - May be able to reduce table size with symmetry, periodicity, interpolation
    - Needs mathlib code for floating point add, multiply (and divide?) to perform these operations
  - **Polynomial approximation**
    - Needs mathlib code for floating point multiply and addition
    - Coefficients use a negligible amount of memory

When to Compute Constant Data?

- At compile/build time?
  - Optimizes program execution speed
  - Minimizes program initialization time
  - Requires ROM section to hold all that data
- At system start-up?
  - Increases program initialization time
  - Requires RAM to hold table
  - Requires ROM to hold code to compute the data values
- When that particular value is needed?
  - Increases program execution time
  - Useful in some but not all cases
  - Could cache the values in RAM as a trade-off

Example: Printf (scanf is similar)

- Standard printf format string: %[flags][width][.precision][length]specifier

Flags
- Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+ Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space) If no sign is going to be written, a blank space is inserted before the value.
Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0 Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width
(number) Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
***** The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Precision
.number For integer specifiers (d, i, o, u, x, X) − precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers − this is the number of digits to be printed after the decimal point. For g and G specifiers − This is the maximum number of significant digits to be printed. For s − this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type − it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.* The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Length
h The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

Specifier
c Character
d Signed decimal integer
I Signed decimal integer
e Scientific notation (mantissa/exponent) using e character
E Scientific notation (mantissa/exponent) using E character
f Decimal floating point
g Uses the shorter of %e or %f
G Uses the shorter of %E or %f
o Signed octal
s String of characters
u Unsigned decimal integer
x Unsigned hexadecimal integer
X Unsigned hexadecimal integer (capital letters)
p Pointer address
n Nothing printed
% Character

Based on https://www.tutorialspoint.com/c_standard_library/c_function_printf.htm

Small Printf

- From Georges Menie, https://www.menie.org/georges/embedded/printf.c

Flags

-
Left-justify within the given field width; Right justification is the default (see width sub-specifier).

+
Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.

(space)
If no sign is going to be written, a blank space is inserted before the value.

#
Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.

0
Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width

(number)
Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.

*****
The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Precision

.number
For integer specifiers (d, i, o, u, x, X) − precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers − this is the number of digits to be printed after the decimal point. For g and G specifiers − This is the maximum number of significant digits to be printed. For s − this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type − it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.

.*
The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Length

h
The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).

l
The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.

L
The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

Specifier

c Character

d Signed decimal integer

I Signed decimal integer

e Scientific notation (mantissa/exponent) using e character

E Scientific notation (mantissa/exponent) using E character

f Decimal floating point

g Uses the shorter of %e or %f

G Uses the shorter of %E or %f

o Signed octal

s String of characters

u Unsigned decimal integer

x Unsigned hexadecimal integer

X Unsigned hexadecimal integer (capital letters)

p Pointer address

n Nothing printed

% Character

Based on https://www.tutorialspoint.com/c_standard_library/c_function_printf.htm

REDUCING READ/WRITE MEMORY REQUIREMENTS

Reducing RAM Requirements

- Conceptual Goals
  - Estimate RAM requirements more accurately
  - Reduce RAM requirements
    - Eliminate unneeded data storage
    - Improve storage density
- Methods
  - Analyze
    - Use better stack depth analysis
  - Reduce

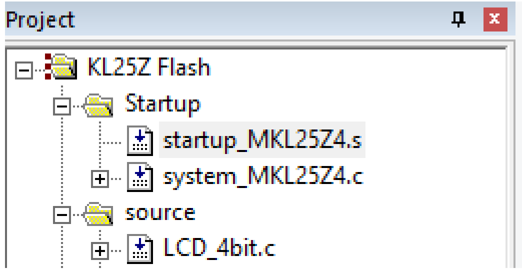
- **Allocate less space for stacks**
- Use language support
- Configure compiler and toolchain better
- Improve storage density
  - Pack data
  - Compress data
  - Better data structures
- **Use more memory-efficient algorithms**
- **Reduce activation record size**
  - Pass arguments by register, not stack
- **Demote costly variables**
  - Too much scope (static, global)
  - Volatile
  - Read-only (const)

Minimize Space Allocated to Stack(s)

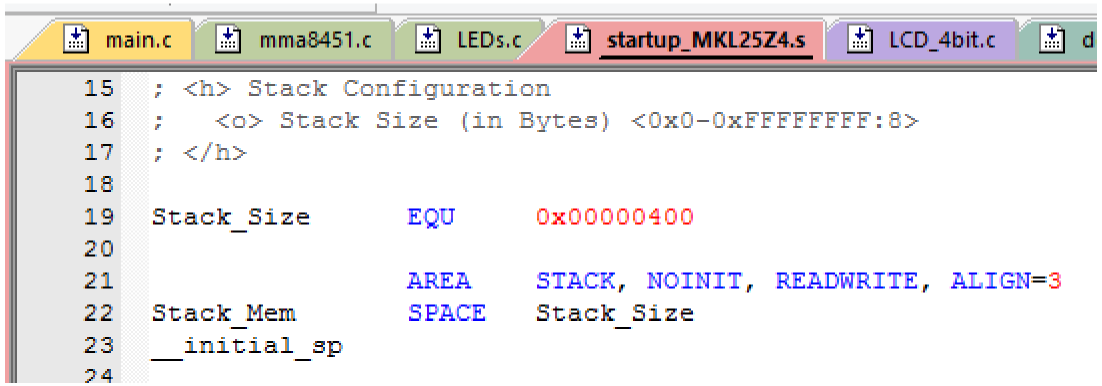
- Analyze program's stack use by threads and handlers (including ISRs)
- Set sizes
  - Bare metal – no scheduler
    - Main thread stack size
  - RTOS
    - Default stack size
    - Thread-specific sizes

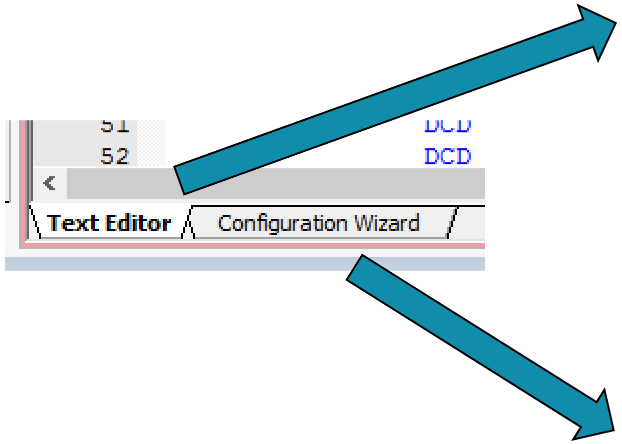
Setting Stack Memory Size (Bare Metal)

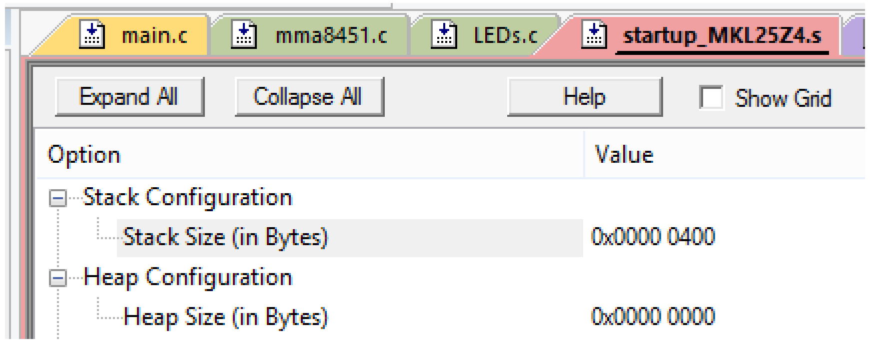
- Defined in Startup/startup_MKL25Z4.s



- Two editing options
  - Select using bottom tab on IDE source code editor window

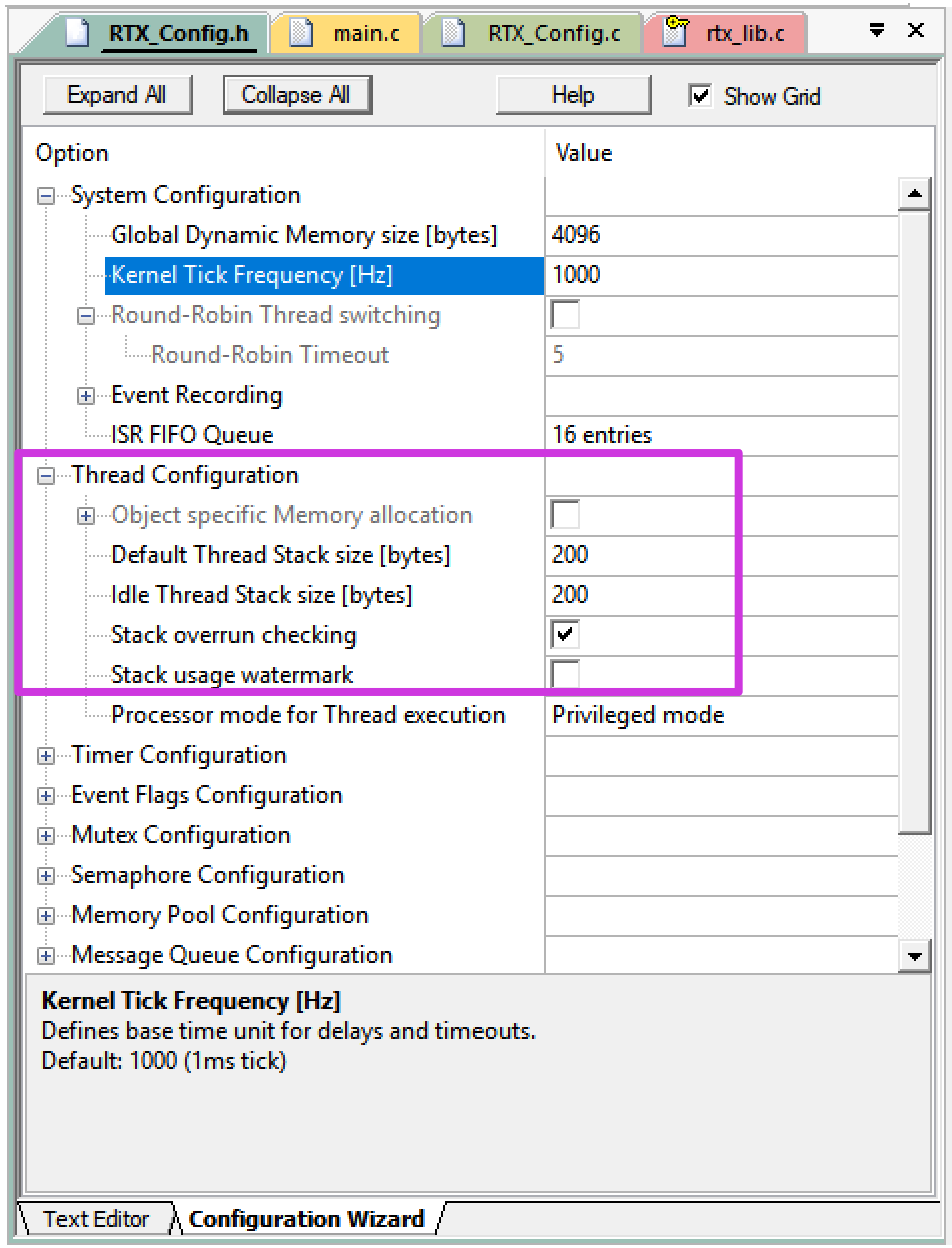






RTX5 Thread Configuration

- Default thread stack sizes
  - Idle thread
  - Other threads
- Support for monitoring stack size
  - Overflow (overrun) detection
  - Watermark initialization



CMSIS-RTOS2: Setting a Custom Stack Size for a Thread

```
/// Attributes structure for thread.
typedef struct {
  const char                   *name;    ///< name of the thread
  uint32_t                  attr_bits;   ///< attribute bits
  void                       *cb_mem;    ///< memory for control block
  uint32_t                    cb_size;   ///< size of provided memory for control block
  void                    *stack_mem;    ///< memory for stack
  uint32_t                 stack_size;   ///< size of stack
  osPriority_t               priority;   ///< initial thread priority (default: osPriorityNormal)
  TZ_ModuleId_t             tz_module;   ///< TrustZone module identifier
  uint32_t                   reserved;   ///< reserved (must be 0)
} osThreadAttr_t;
```

```
osThreadId_t osThreadNew ( osThreadFunc_t          func,
                           void *                  argument,
                           const osThreadAttr_t *  attr
                         )
```

- Declare a structure "attr" of type osThreadAttr_t
- Initialize it
  - attr.stack_mem to point to custom stack space
  - attr.stack_size to size of custom stack space
  - Clear unused attributes to NULL
- Call osThreadNew(my_thread, &argument, &attr);

Use Toolchain Support

- Maximize optimization for size
- Select smallest feasible memory model, if applicable
  - Shortens addresses
  - Simplifies their calculation and use
- If possible, use library versions optimized for less RAM
- If speed is also important, take advantage of the "80/20" rule
  - Most programs spend about 80% of their time in 20% of the code
  - Can specify different optimizations per module
- Apply different optimizations for the different parts of the code
  - Optimize time-critical modules for speed
  - Optimize remaining modules for size

Use Different Algorithms

- Example: sorting n data items
- Merge sort
  - Typical implementation does not sort in place, needs an extra O(n) RAM space
  - Possible to reduce space overhead O(n/2), but makes coding more complicated, slows sorting operation
  - Possible to reduce space requirement to O(n), but requires recursion (so need more stack space) and makes coding more complicated
- Quick sort
  - Data stays in place
  - Divide and conquer approach uses recursion, requiring O(n) stack space and O(n) data storage,
  - Can reduce to O(log n) stack space with special tricks (Sedgewick's method)

Use Different Algorithms

- Bubble sort
  - Is an in-place sort, so no extra memory needed (actually, need to store one element to perform the swap)
  - Is slow - $O(n^2)$
- Shell sort
  - In-place sort
  - Not recursive, so stack is not used
  - Speed is typically $O(n \log n)$, worst case is $O(n^2)$
  - Sometimes used to implement C's qsort library function
- Heap sort
  - Uses a binary tree
    - If explicitly implemented as a tree, extra memory ($O(n)$) is needed
    - If binary tree is implemented in an array, no extra memory is needed but coding is more complex
  - Speed is $O(n \log n)$

Reduce Activation Record Size

- Use smallest practical automatic variables
- Remember ARM calling convention
  - First four arguments can be passed in registers r0-r3
  - Remaining arguments will go on the stack
- Use smallest practical arguments
  - Pass pointers to large data rather than data itself
  - Group related items together in a structure and pass a pointer
  - Note that arguments passed on stack are word-aligned, so a one-byte argument will take four bytes
- Limit number of arguments to a function
- Make function result fit within two registers
- Do NOT convert automatic variables to statics or globals
  - Automatics reuse memory space, releasing their memory when the scope (e.g. function) ends
  - Each global or static variable uses its memory for the full lifetime of the program

Example Source Code

```
sprintf(buffer,
"$APRMC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,W,%05.1f,%04.1f,%06ld,%05
.1f,W*", hr, min, sec, lat_deg, lat_min,
lon_deg, lon_min, speed, track, date, var);
```

- What does the object code look like?

Object Code for Call to Sprintf

```
BL      __aeabi_f2d
MOV     r7,r0
MOV     r0,r4
STR     r1,[sp,#0x2c]
BL      __aeabi_f2d
STR     r0,[sp,#0x28]
MOV     r6,r1
MOV     r0,r4
BL      __aeabi_f2d
MOV     r4,r0
MOV     r5,r1
LDR     r0,[sp,#0xc]
BL      __aeabi_f2d
STR     r0,[sp,#0x10]
STR     r1,[sp,#0x18]
LDR     r0,[sp,#8]
BL      __aeabi_f2d
MOV     r3,r1
MOV     r2,r0
LDR     r0,[sp,#0x40]
LDR     r1,[sp,#0x2c]
STR     r1,[sp,#0x3c]
STR     r6,[sp,#0x2c]
STR     r0,[sp,#0x30]
STR     r7,[sp,#0x38]
LDR     r1,[sp,#0x18]
STR     r5,[sp,#0x24]
LDR     r0,[sp,#0x10]
STR     r1,[sp,#0x1c]
STR     r0,[sp,#0x18]
LDR     r0,[sp,#0x44]
STR     r4,[sp,#0x20]
STR     r3,[sp,#0xc]
STR     r2,[sp,#8]
STR     r0,[sp,#0x10]
LDR     r1,|L1.204|
ADD     r0,sp,#0x48
LDR     r3,[sp,#0x34]
LDR     r2,[sp,#0x14]
BL      __2sprintf
```

Why So Much Code?

- 13 arguments!
- Can only pass first four in registers
- Remaining arguments go on stack
- Compiler must move each argument from memory (automatic variables in activation record) to memory (argument space in activation record)

Use Stack-Friendly Functions

- Printf, scanf provide many features but use much stack space (or even heap, depending on implementation)
- Use other functions (itoa, ftoa, atoi, atof, etc.)
- Look for other lightweight functions when possible, or write your own

Demote Costly Variables

- Too much scope
  - Static variables (including globals) always use RAM, eliminating reuse
  - Reduce scope of variables by declaring them within function (or even block {})
  - Enables RAM reuse by placing variables on stack
- Volatile
  - Compiler must allocate space in memory for a volatile variable, can't optimize it into only a register
- Read-only (const)
  - Some data is only read, never written
  - Store this data in ROM rather than RAM
  - Use **const** to modify variable type, indicating variable goes in ROM

```
const int lookup_table[256] = {234, 234, 345, 252, … };
```

OPTIMIZATION FOR MULTITASKING SYSTEMS

Key Optimizations for Multitasking Systems

- ROM
  - Configure RTOS to include only necessary features
  - Share read-only resources (code, RO data) between threads if possible
- RAM
  - Improve the accuracy of stack depth estimates
    - Tighter bounds enable safe reduction in stack space allocated
    - Acquire tool to analyze stack depth from object code
    - Set up test harness to perform automated stack depth measurement
  - Use a non-preemptive scheduler
    - Only needs enough stack space for the largest stack of all the tasks
  - Retain some preemption, but reduce number of stacks required
    - Combine independent run-to-completion tasks with a mini-scheduler (state machine)
      - Note: combined tasks must not block!
    - Use preemption threshold scheduling
  - Configure RTOS
    - Do not over-provision (e.g. # tasks, # mutexes, queue sizes, etc.)
    - Use safe dynamic memory allocation to reuse resources

MEMORY SIZE OPTIMIZATION EXAMPLE: FILE SYSTEM

MEMORY SIZE OPTIMIZATION EXAMPLE: SHIELD CODE

APPENDIX

Maximum Stack Use - Threaded

- Use information for thread root functions
- Also need to consider other kernel threads (idle thread, event timer thread, etc.)

Init_LCD (Thumb, 110 bytes, Stack size 16 bytes, lcd_4bit.o(.text))

[Stack]

- Max Depth = 64
- Call Chain = Init_LCD ⇒ lcd_write_cmd ⇒ wait_while_busy ⇒ lcd_read_status ⇒ Delay

[Calls]

- >> Delay
- >> lcd_init_port
- >> lcd_write_cmd
- >> lcd_write_4bit

[Called By]

- >> main