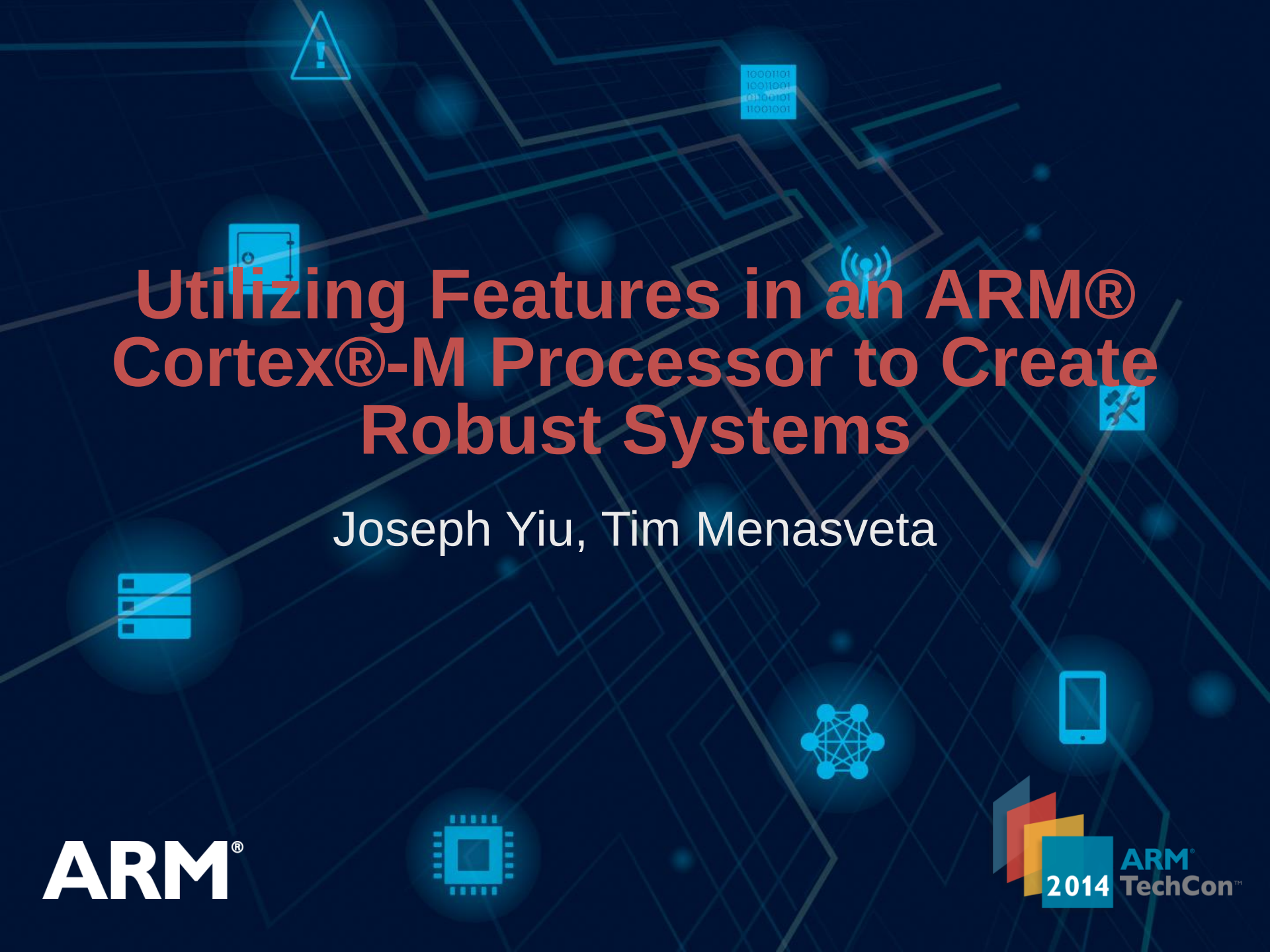

2014
ARM®
TechCon™
UBM
Tech



Utilizing Features in an ARM® Cortex®-M Processor to Create Robust Systems

Joseph Yiu, Tim Menasveta

ARM®

2014 ARM® TechCon™

Overview

- What we are trying to cover
  - NOT : Preventing hardware error
  - NOT : General coding practice
  - YES : Cortex-M processor specific topics
- What can go wrong?
- Stack Issues
- Exception Handling on Cortex-M Processors
- Software Techniques for Robustness
- Recommendation for Fault Handlers

What can go wrong?

- Stack overflow / stack leak
  - A very common root cause of failure
- Inadequate checking of user/external inputs
- Hardware Errors
  - Memory corruptions
  - Bit flip in SRAM
  - Soft error (Data corruption) inside hardware
- Consequences
  - Fault exceptions
  - Lock-up

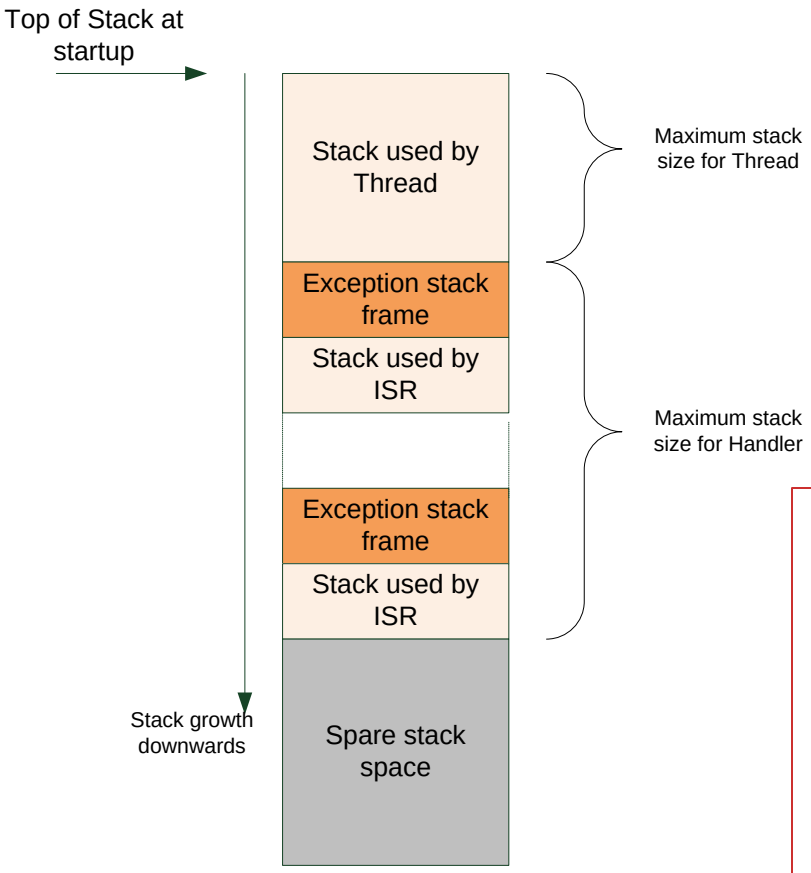
Stack Issues

- Determining the stack size requirements
  - Stack report from tools are useful…
  - Maximum stack size for the application + maximum stack size used by ISRs + Maximum size of stack frames for each level of ISR stacking (9 or 27)
  - Beware of nesting of exceptions
- Selecting the right stack arrangement on Cortex-M
  - 2 Stack Pointers (Main Stack Pointer – MSP, and Process Stack Pointer – PSP)
  - Need to determining relevant stack pointer MSP or PSP in fault handling
- Detecting stack overflow in runtime
  - Run-time detection requirements
  - MPU as trigger

Sizing the stack

- Stack usage report available in many development suites
  - Reports stack usage per function
  - Reports Maximum stack size in a call tree
  - Beware of limitations…
- Need to also consider stack used by exception handlers and their stack frames
  - Exception nesting needs to be considered

No OS Stack Usage



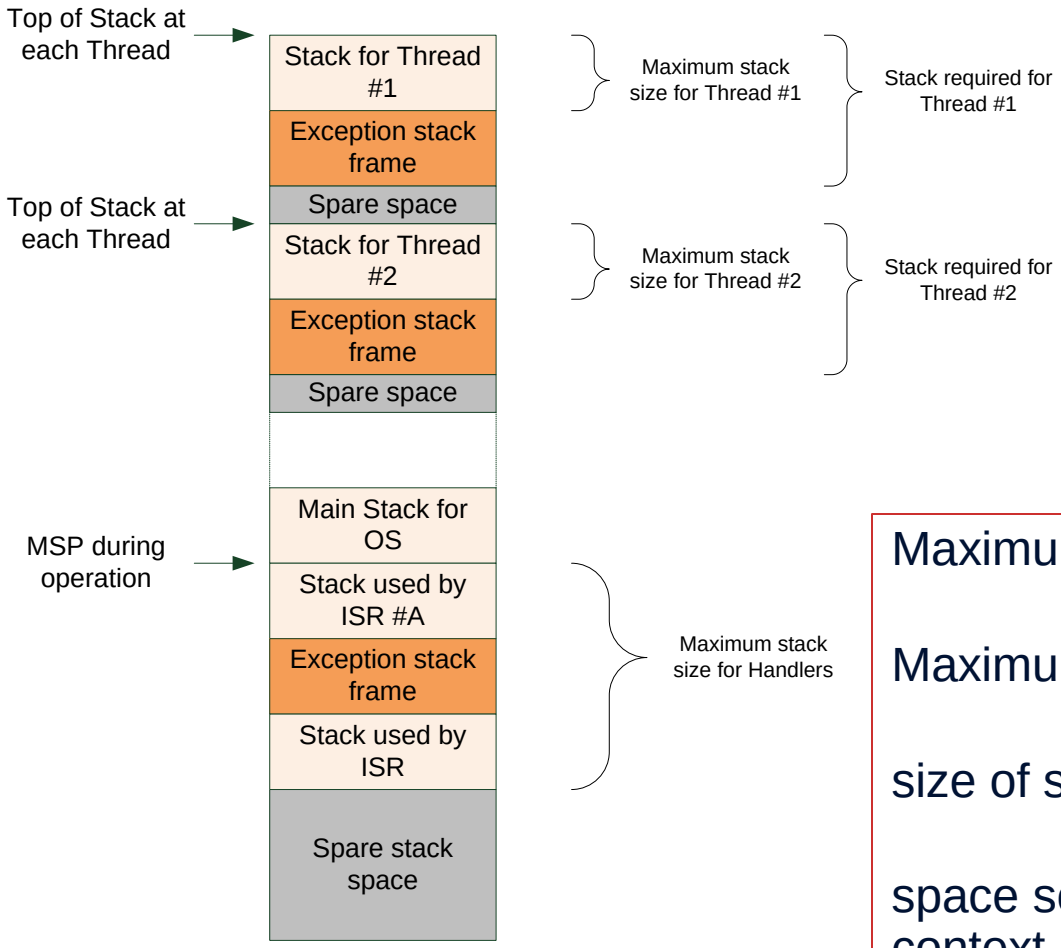
Maximum stack size =

Maximum stack size for the application +

Maximum stack size used by ISRs +

Maximum size of stack frames for each level of ISR stacking

OS / Multi-threaded Stack Usage



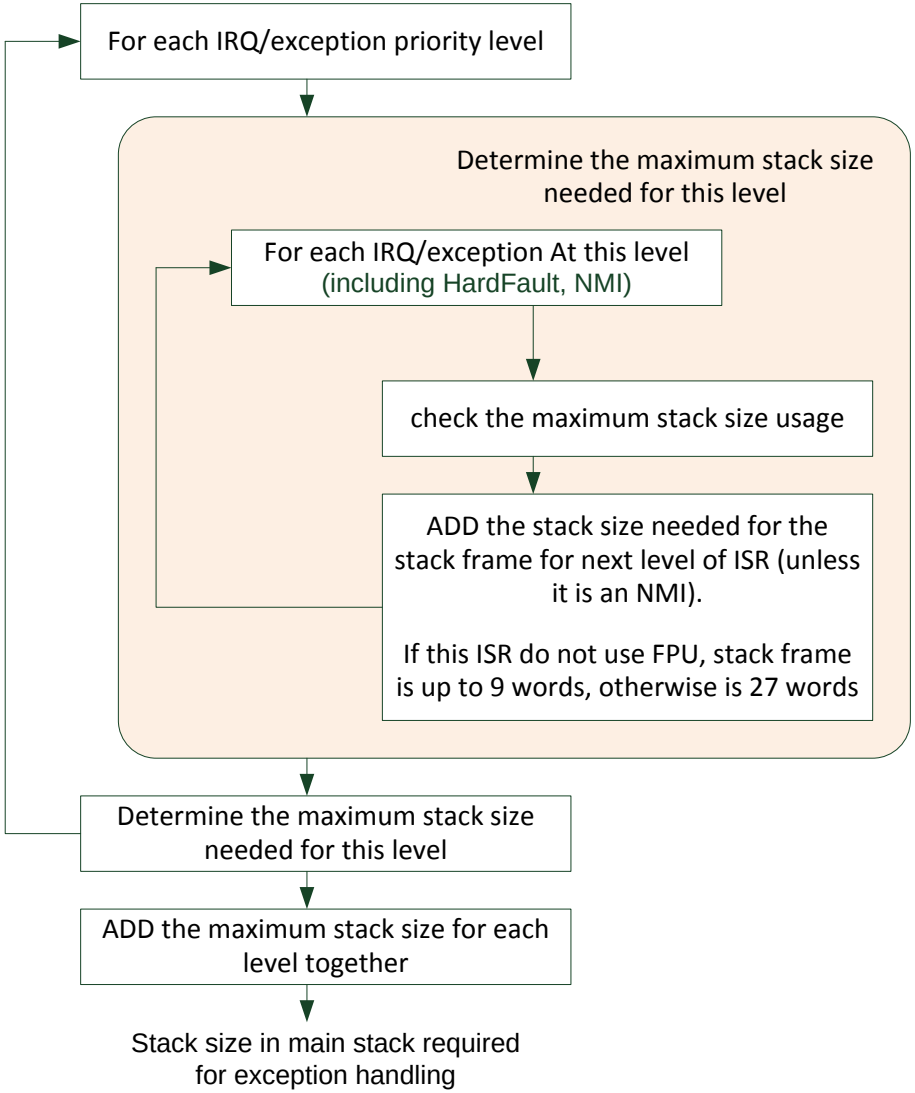
Maximum stack size =

Maximum stack size for application thread +

size of stack frame (9 or 27 words) +

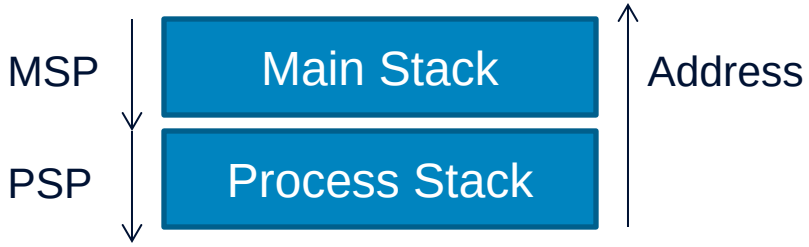
space software stacking operations in OS context switching code

Sizing the Stack



Dual Stack Pointers on Cortex-M

- MSP and PSP dual stack pointers maintain separate stacks for Operating Systems and Application Threads
- It can also be useful for applications without OS
  - PSP for thread
  - MSP for handlers



- Fault can trigger exception handler from either OS or thread mode – check valid pointer when exception occurred!

```
TST LR, #0x4; Test EXC_RETURN number in LR bit 2
ITE EQ; if zero (equal) then
MRSEQ R0, MSP; MSP was used, put MSP in R0
MRSNE R0, PSP; else, PSP was used, put PSP in R0
… ; carry-out validity check on stack pointer
```

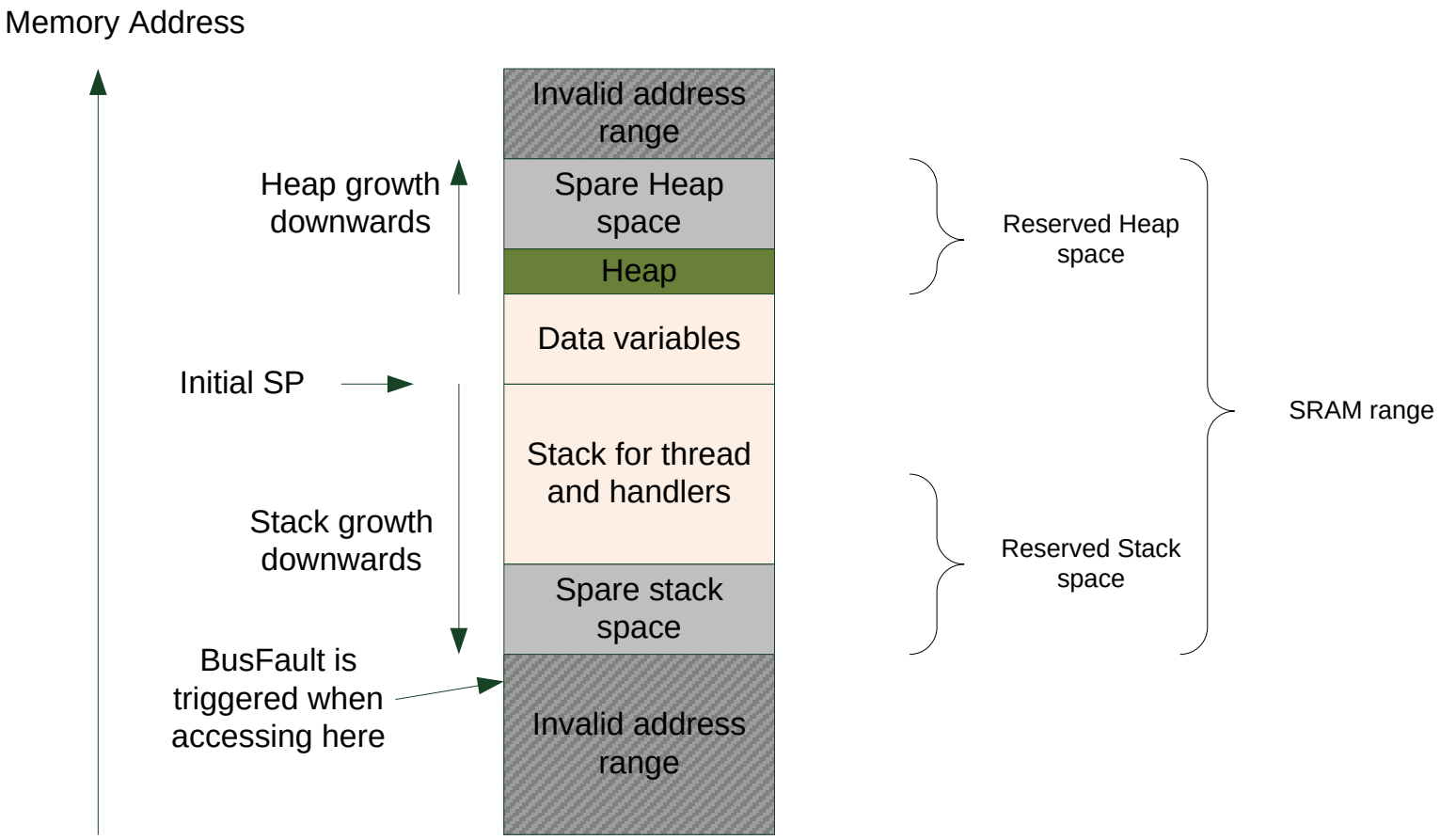
Stack Overflow / Leak Detection

- Key word fill (e.g. 0xDEADBEEF)
  - For development only – detect by memory inspection, and can miss worst cases
- Periodic checks of MSP by Timer interrupt
  - Delayed detection
- Added Instrumentation code
  - Not suitable for deployment
- Immediate Detection is better
  - Place stack at bottom of memory page
  - Define non-accessible regions in Memory Protection Unit (MPU)

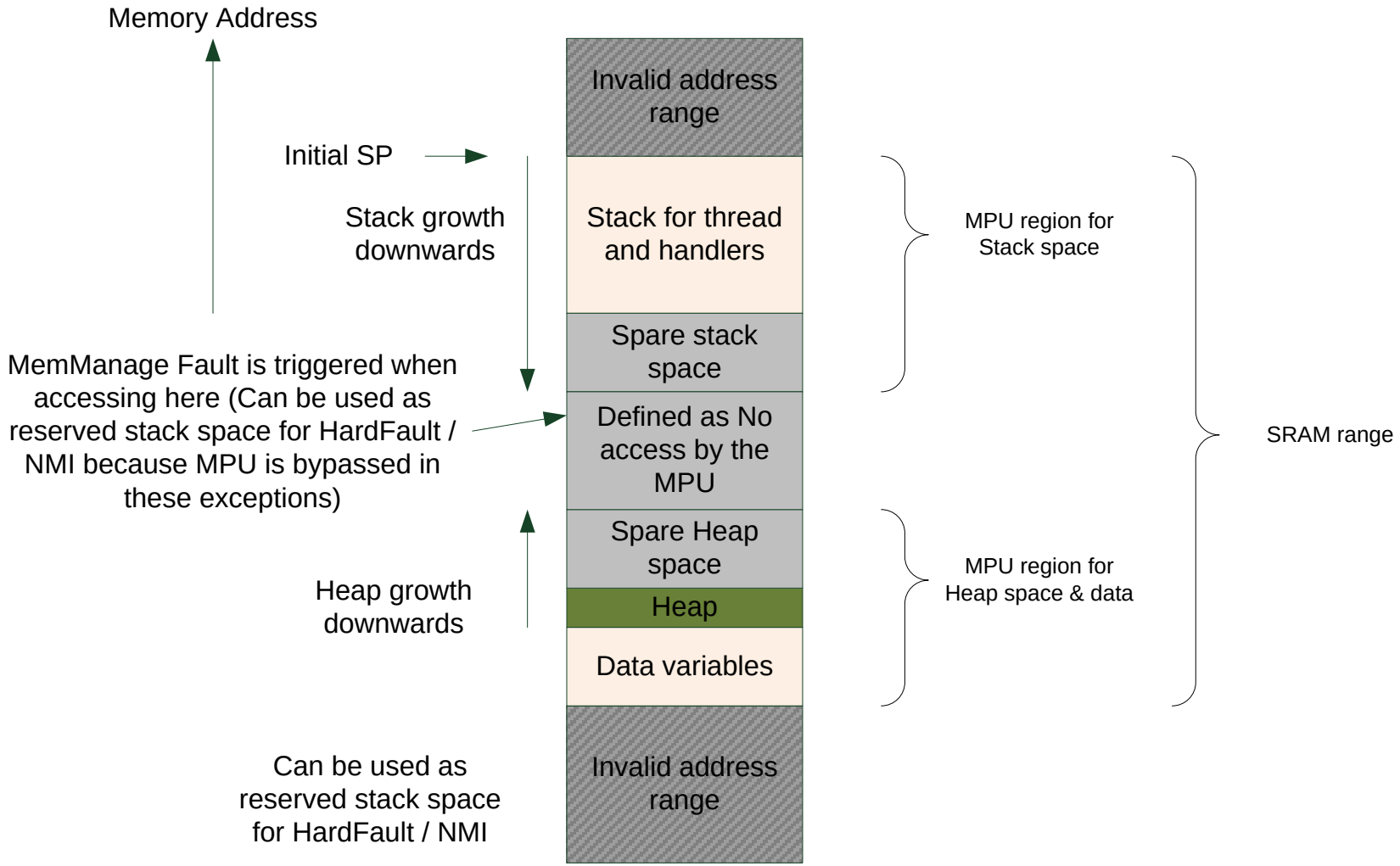
Exceptions on Cortex-M

Architecture		ARMv6-M		ARMv7-M	ARMv7E-M
Processor Core	Priority	Cortex-M0	Cortex-M0+	Cortex-M3	Cortex-M4
Reset	-3 (Highest)	☑	☑	☑	☑
NMI	-2	☑	☑	☑	☑
Hard Fault	-1	☑	☑	☑	☑
MemManage Fault	Programmable			☑	☑
Usage Fault	Programmable			☑	☑
Bus Fault	Programmable			☑	☑
SVC	Programmable	☑	☑	☑	☑
Debug Monitor	Programmable	☑	☑	☑	☑
PendSV	Programmable	☑	☑	☑	☑
SYSTICK	Programmable	☑	☑	☑	☑
Interrupt #0	Programmable	☑	☑	☑	☑
Interrupt #1	Programmable	☑	☑	☑	☑
Interrupt #2	Programmable	☑	☑	☑	☑
...	...	☑	☑	☑	☑
Interrupt #N	Programmable	☑	☑	☑	☑

Placing Stack at Bottom of SRAM

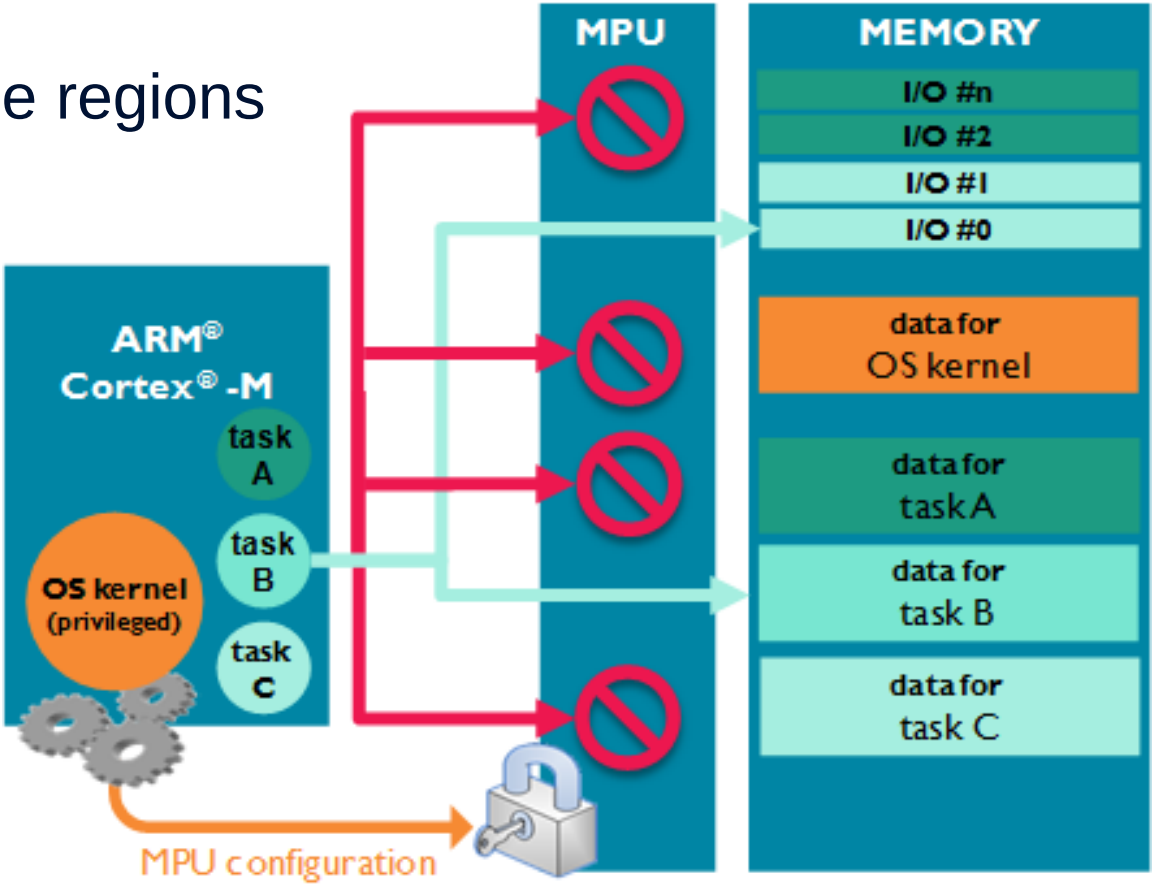


No Access Regions by MPU



Memory Protection Unit (MPU)

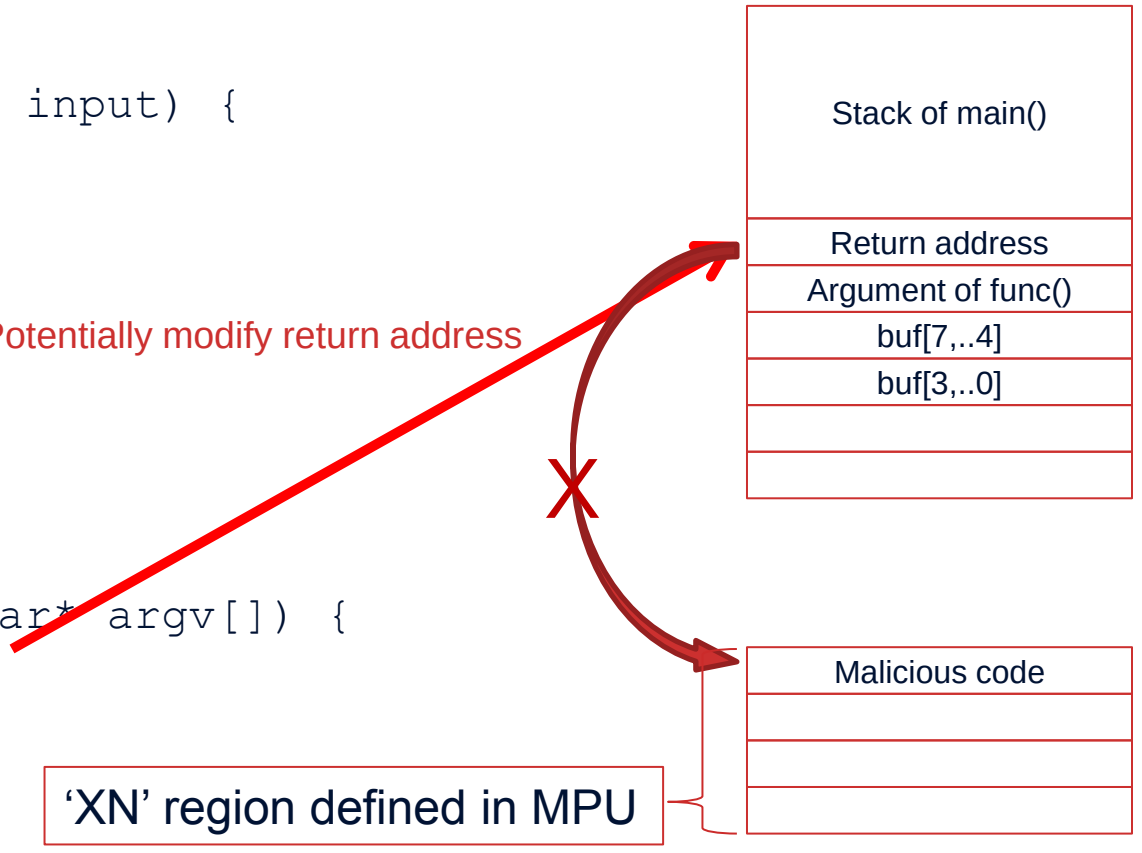
- Prevents application task from corrupting OS or other task data
  - Improves system reliability
- Up to eight configurable regions
  - Address
  - Size
  - Memory attributes
  - Access permissions
- Optional in Cortex-M0+, Cortex-M3, Cortex-M4 and Cortex-M7



XN attribute in MPU

- Prevent Stack overflow attacks

```
void func(const char* input) {

        char buf[8];

        do_task();

}



int main(int argc, char* argv[]) {
        func(argv[1]);

        return 0;

}
```



Software Techniques

- Short, Simple Handlers
  - Split if necessary and used PendSV
- INT mask register
  - Block out exceptions during critical code execution
- Never-execute MPU Attribute
  - Prevent malicious injected code execution
- Stack leak detection in SW
  - Low priority Timer periodically keeps track of stack usage size
- Validation of external inputs
  - Prevents failures due to unintentional or malicious input of unbounded arguments
- SW runtime diagnosis
  - Run-time self-test software, e.g. Yogitech's fRSTL

Recommendation for fault handlers

- Stack overflow is the most common cause of fault
  - MPU enables real-time detection of stack overflow
  - Size your stack usage carefully
  - Keep handlers short and simple
    - Avoid C lib functions such as printf() and malloc()
    - Split fault handlers to critical short parts to avoid stack errors in fault handlers
- Check MSP valid
  - Stack overflow might have caused the exception – check valid stack pointer before handling
- Automatically reset of system
  - Not recommended during development/debugging

Joseph Yiu

Senior Embedded Technology Manager

Joseph.Yiu@arm.com

Tim Menasveta

Product Manager

Tim.Menasveta@arm.com

ARM®

2014 ARM® TechCon™


2014
ARM®
TechCon™
UBM
Tech