Fachhochschule
Münster University of
Applied Sciences

Multitasking

in

Embedded Systems

Multitasking in ES

- What is Singletasking?
- What is Multitasking?
- Why Multitasking?
- Different approaches
- Realtime Operating Systems
- FreeRTOS
- Synchronization
- Example Project
- Debugging Multitasking Software

What is Singletasking?

- Do only one thing at a time
- Most efficient way to solve problems
- Applicable to every algorithm
- No management overhead
- No internal synchronization required
- Easy to code
- Busy-wait for I/O
- No Interrupts

Singletasking Example

```
while (1) {
    if ( Byte = receiveByte() ) { // blocks!
        Buffer[Index++] = Byte;
        if ( Index >= MessageSize) {
        M_t* M = (M_t*) Buffer;
            switch (M->Type) {
                case mt_CommandA: ...
                case mt_CommandB: …
                default: break;
            }
            Index = 0;
        }
    }
}
```

What is Multitasking?

- Aim for multiple targets
- Switch context often
- Management overhead
- Synchronization required
- Interrupts required
- Hard to code/ debug
- Implicit delays
- Increased memory usage
- Implementation difficult for many algorithms

Multitasking Example

```
main() {
  int Queue = xQueueCreate(...);
  ttc_task_create(Receive, Queue, …);
  ttc_task_create(Process, Queue, …);
}

void Process(int Q) {
  while (1) {
    M = (M_t*) ttc_queue_pop_front(Q);
    if (M) {
      switch (M->Type) {
        case mt_CmdA: ...
        case mt_CmdB: ...
        default: break;
      } } } }
```

```
void Receive(int Q) {
  char Buffer[10][100];
  int Index = 0;
  while (1) {
    char* Writer = &(Buffer[Index,0]);
    int Remaining = MessageSize;
    while (Remaining > 0) {
      if ( Byte = readByte() ) { // sleeps!
        *Writer++ = Byte; Remaining--;
    }
    ttc_queue_push_back(Q,
                        &(Buffer[Index,0]););
    Index++;
    if (Index > 99)  Index = 0;
  }
}
```

Fachhochschule Münster University of Applied Sciences

Why Multitasking?

- Functions spawnable multiple times
- Eases handling of slow IO
- Benefits from multiple CPU-cores
- Only 1 central Timer required
- Short Interrupt Service Routines
- Less global variables required

No life without Multitasking!

- Every Embedded System needs MT
- MT often implemented via Interrupts
  - Complex Service Routines
  - Data Exchange via Global Variables
  - Difficult to debug
- Typical approach: Super-Loop
  - Periodically starts set of functions
  - Similar to task scheduler

Different Approaches

- Multiprogramming
  - Ancient mechanism for Periheral Access
  - Realized via Terminal Stay Ready (TSR)
- Cooperative Multitasking
  - Central Scheduler manages Processes
  - Each process grants CPU to other processes
  - Single process can block whole system
- Preemptive Multitasking
  - Scheduler interrupts each process periodically
  - Requires central Interrupt-Timer
- Preemptible Multitasking
  - High priority Applications can interrupt others (OS/2, Linux, FreeRTOS)
  - Allows faster response times

Realtime Scheduling

- Definition of Realtime

  Real-time computing means a hardware + software system that must response within a strict time constraint.

- Realtime constraint
  - **Hard**
    Violation causes hazardous results
  - **Firm**
    Infrequent violations tolerable but degrade system's quality of service
  - **Soft**
    Usefullness of results degrade after their deadline

→ http://en.wikipedia.org/wiki/Real-time_computing

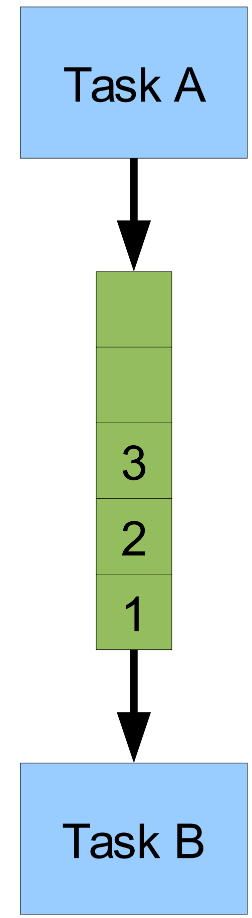
FreeRTOS

- Multitasking Scheduler
  - Premptible Multitasking
  - High triority tasks block low priority ones
- Inter Thread Communication
  - Semaphores
  - Queues
- Developed specially for Embedded Systems
- Open Source
- FreeWare with Commercial Support
- Ported to several µC Architectures

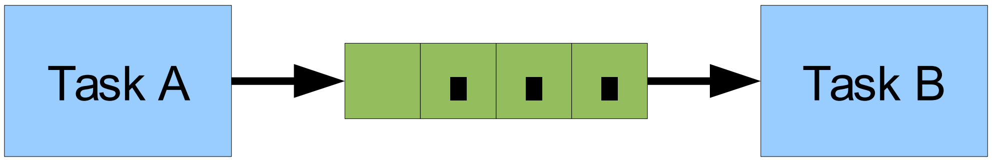
  → http://www.freertos.org/

TTC – Queues

- Base of inter task communication
- Send message
  - Task → Task
  - Interrupt Service Routine → Task
- Call by value
- Reading from empty queue
  - Function call waits until Queue is filled
  - No CPU time is wasted during Wait
- Operations
  - ttc_queue_byte_push_back(Q, Byte);
  - Byte = ttc_queue_byte_pull_front(Q);



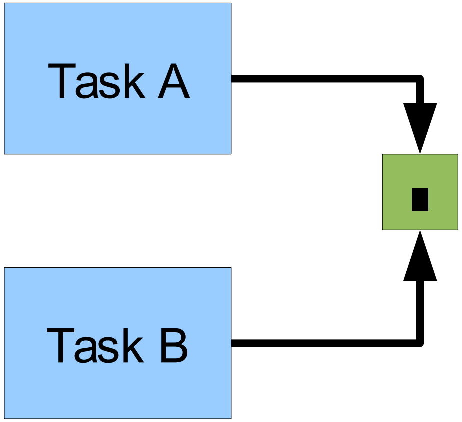
TTC - Semaphores



- Counting
  - Implemented as unsigned integer Counter
  - Task A can increase Semaphore by up to N token at once
  - Data in Queue is of no interest
  - ttc_semaphore_take(S, N)
  - ttc_semaphore_give(S, N)

TTC - Mutex

- Implemented as Counter 0 – 1
- Best suited for Resource locking
- Non recursive
  - Can be locked only once
- Recursive
  - thread can lock multiple times
  - must be unlocked as often as being locked
- Task B waits until Task A activates once
- Operations
  - ttc_mutex_lock(M)
  - ttc_mutex_unlock(M)



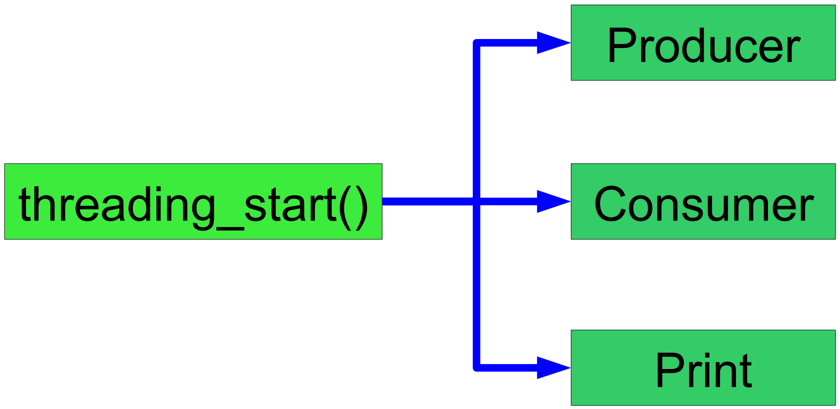
Multithreading with Queues

- Activate Extensions for Queues example on STM32-P107

```
> activate.600_example_threading_queues.sh
```

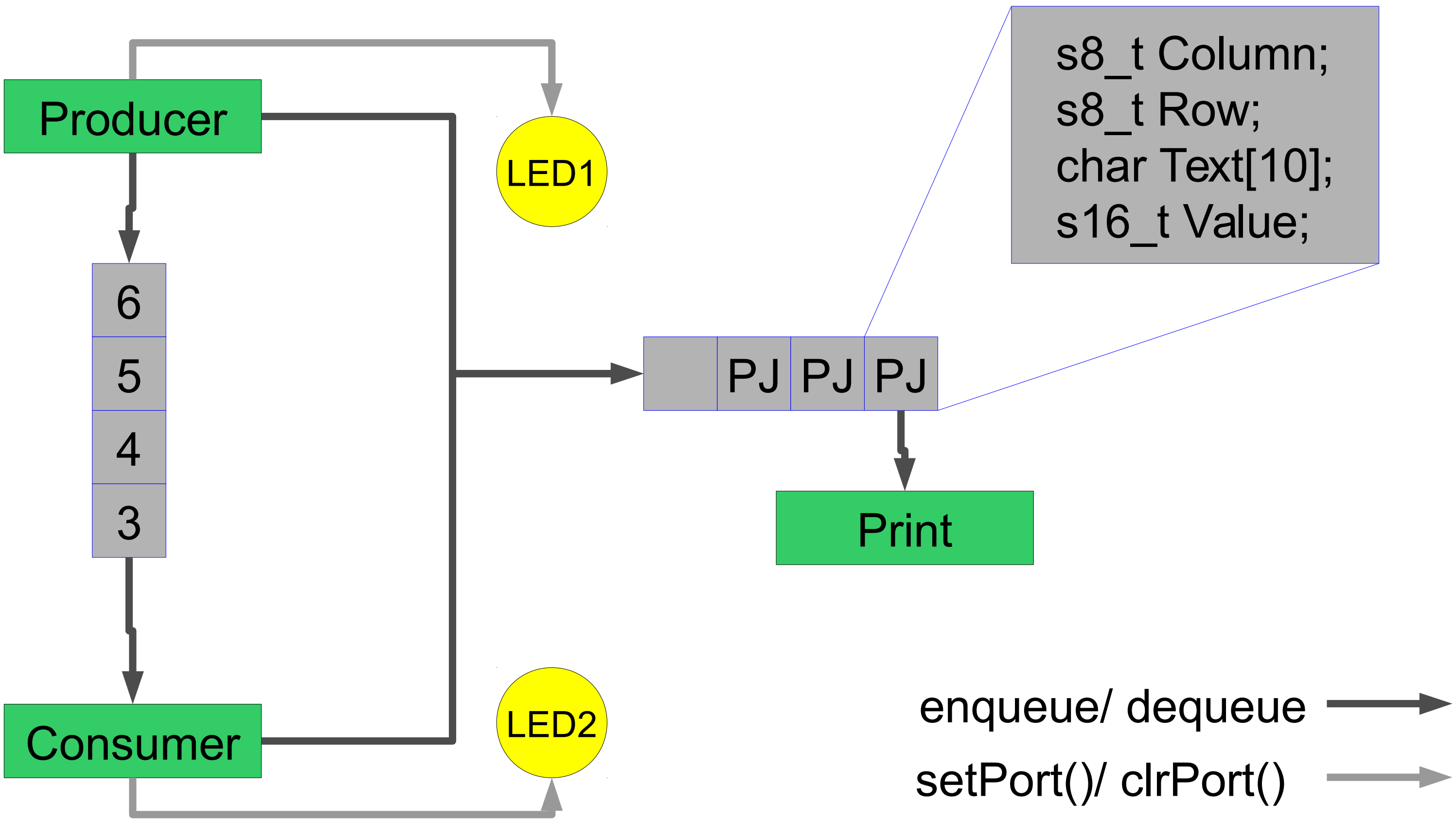
Multithreading with Queues

- Spawning Tasks



Multithreading with Queues

- Communicate via Queues



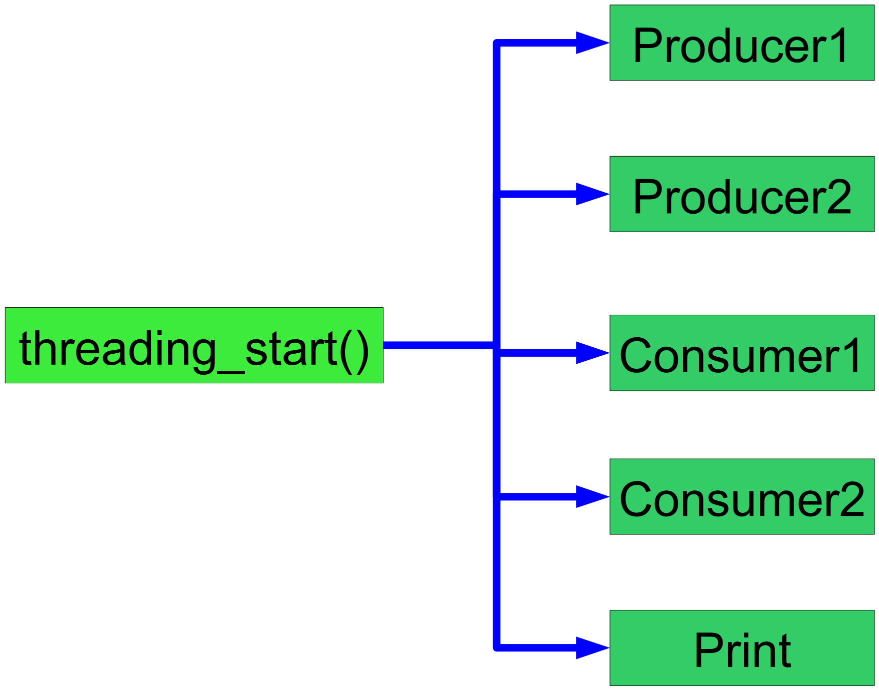
Multithreading with Semaphores

- Activate Extensions for Semaphores example on STM32-P107

```
> activate.600_example_threading_semaphores.sh
```

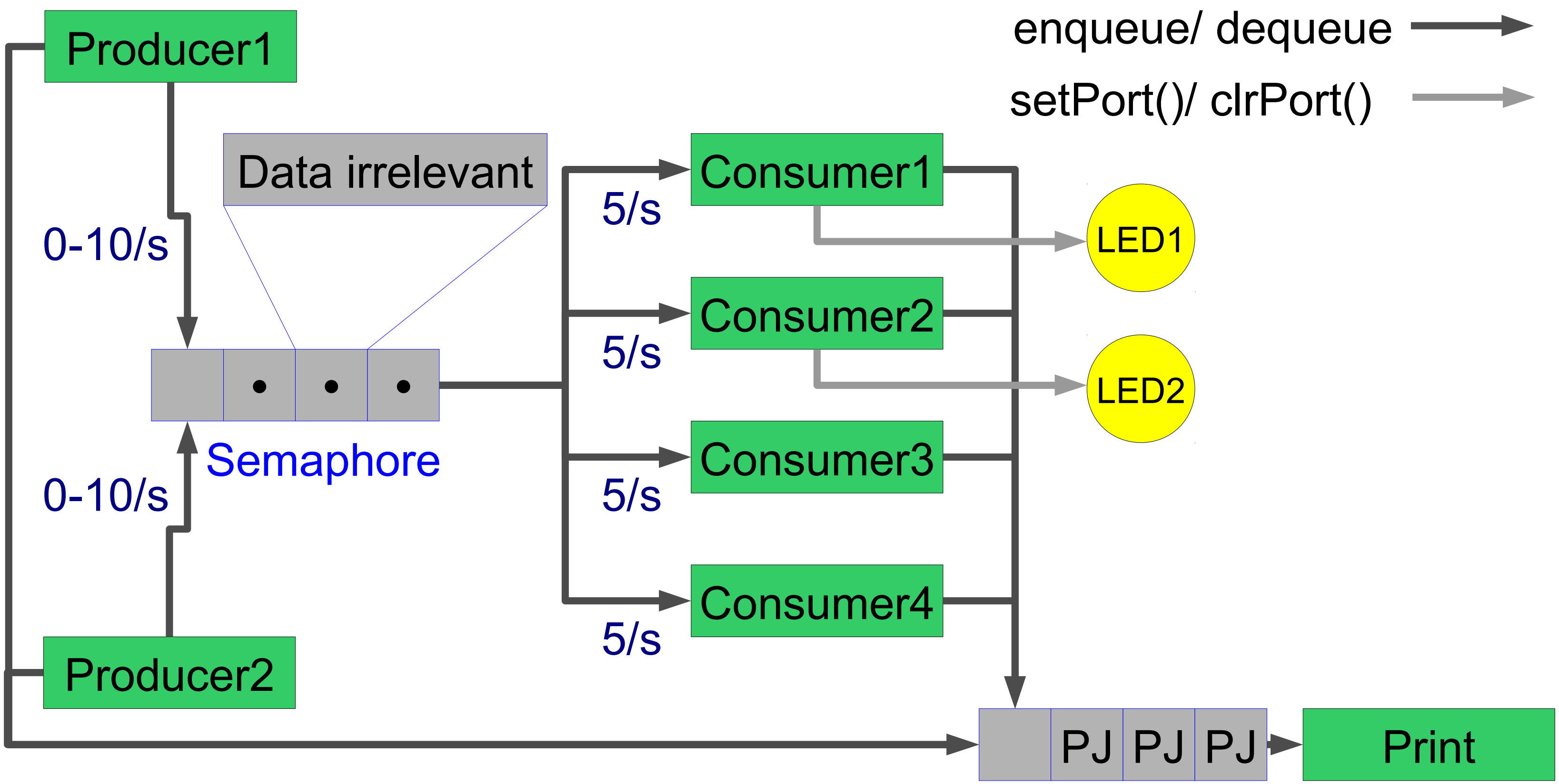
Multithreading with Semaphores

- Spawning Tasks



Multithreading with Semaphores

- Communicate via Semaphore



Debugging Multitasking Software

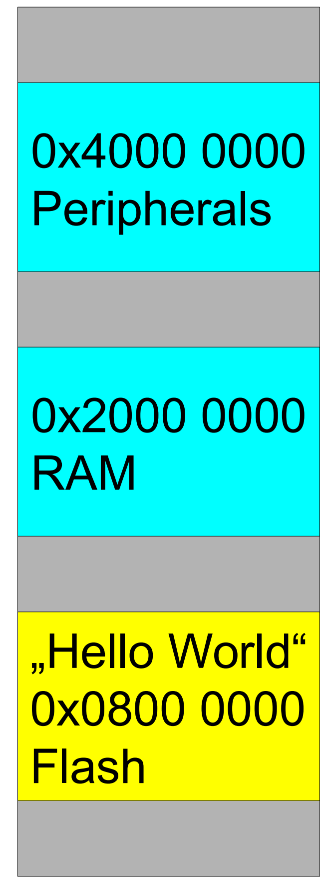
- Different types of Memories
- Multitasking & Stacks
- Watchdogs
- Binary file formats

Different types of Memories

- Storage
  - RAM
  - Heap
  - Stack
- Datatypes
  - Constants
  - Global Variables
  - Local Variables

Constants

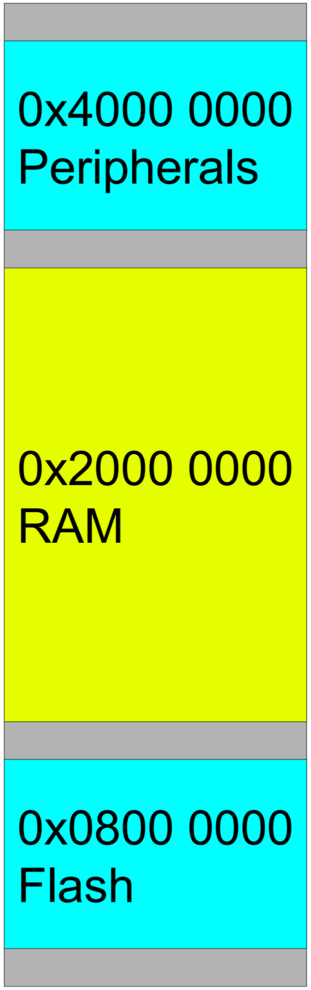
- Stored in Flash along code
- Automatically initialized
  (initial value contained in binary)
- Reference to value points to
  address > 0x0800 0000



- Example:
  char* HelloWorld = „Hello world“;

RAM

- Area with read- / write access
- Multiple Memories possible
  - On chip (start at 0x2000 0000)
  - Off Chip via memory controller
    - Static RAM
    - Dynamic RAM



Stack

- Located at one end of RAM
- Grows towards other end by each
  - Function call
  - Local variable
- Shrinks by each
  - local variable end of scope
  - Return from function call

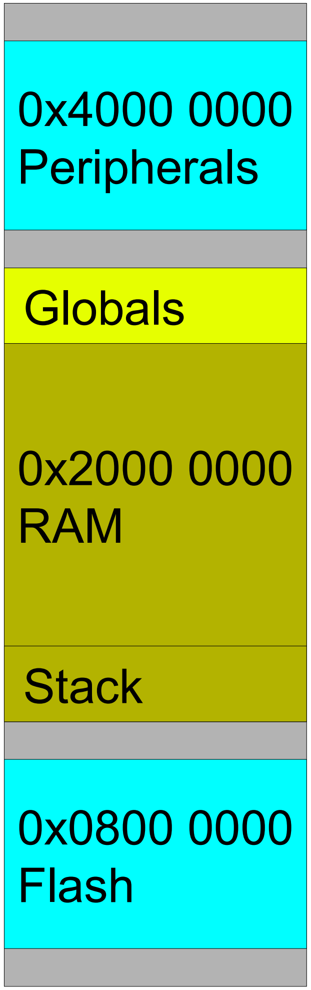
Global Variables

- Stored in RAM
- Initialized by code at program start
- Assembled into block by linker
- Reference to value points to address > 0x2000 0000

- Example:

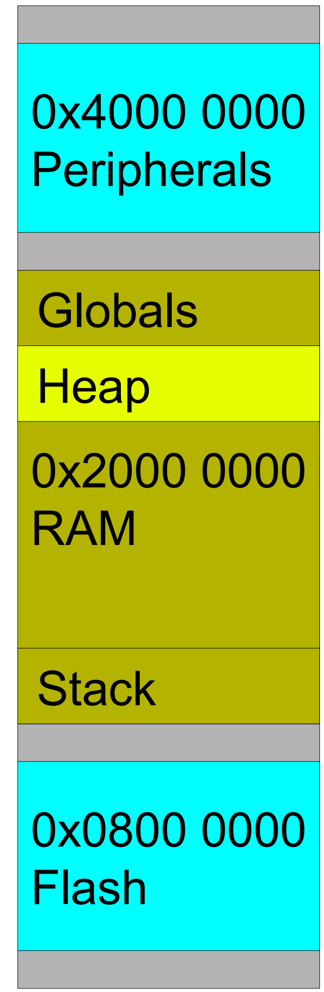
```
int V = 42;

void main() {
  V++;
}
```



Memory Heap

- Block of memory in RAM
- Allocated via malloc()
- Freed via free()
- Different managers
  - heap1, heap2, heap3
  - see corresponding extensions

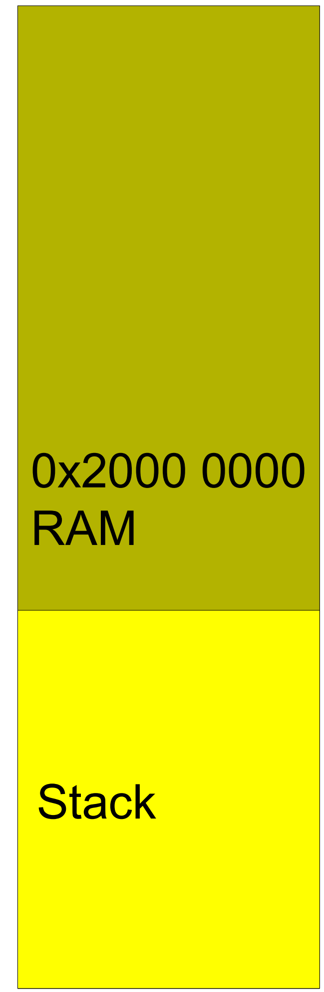


Multitasking & Stacks

- Stacks in Singletask Applications
- Stacks in Multitasking Applications
- Stacks in FreeRTOS
- Stackoverflow

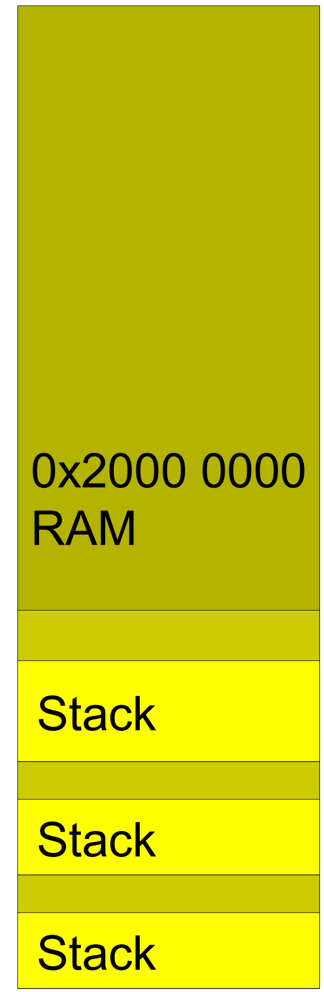
Stacks in Singletask Applications

- One single stack
- Grows from one end of RAM
- Efficient memory usage (no holes)
- Always big egnough
- Only one stack size to calculate
- Mostly invisible to developer



Stacks in Multitasking Applications

- One stack per thread
- Grows at different locations
- Memory holes between stacks
- sizes are tradeoff
- Each size hard to calculate
- Developer has to re-check size
  - New local variables
  - More function calls



Stacks in FreeRTOS

- Stacks handled by FreeRTOS
- Stack size defined at stack creation
  - Argument to xTaskCreate()
  - Cannot be changed during task execution
  - Tasks normally do not end
- Stacks allocated on memory heap

Stackoverflow

- Two detection schemes in FreeRTOS
  - Activated in FreeRTOSConfig.h
  - Stack size checked whenever a task is put to sleep
- activate.300_scheduler_free_rtos_stack_check_1.sh
  - Checks stack pointer to be within allowed boundaries
  - Cannot find overflow occured during task run time
  - Fast check (only two pointer comparisons)
- activate.300_scheduler_free_rtos_stack_check_2.sh
  - First checks stack pointer via method 1
  - Each stack filled with known value
  - Last 16 bytes of stack checked for changed values
  - Will find out if stack has grown into last 16 bytes
  - Slower than method 1 but more reliable

Stack size calculation

- Three step approach
  1) Double stack sizes until no overflow occurs
  2) Run application
  3) Use high water marks to minimize sizes
- Problems
  – How to force tasks to use maximum memory?
    → Global variable: bool TestMode
  – Recalculate after code changes

High Water Marks

- Configured in
  - #define INCLUDE_uxTaskGetStackHighWaterMark 1
- Activate via
  - activate.300_scheduler_free_rtos_high_watermarks.sh

→ FreeRTOS - HighWaterMark

Hardfault Handler

- CPU reads/ writes illegal memory address
  → bus fault error generates interrupt
  → calls cm3_hard_fault_handler_c()
- cm3_hard_fault_handler_c()
  - Gathers information about fault reason
  - Calls Assert_Halt_EC(ec_HardFault)
- Assert_Halt_EC()
  - Stops execution in endless loop
  - Contains Breakpoint

Debugging a Hardfault

Memfault Handler

Watchdogs

Binary file formats

- objdump