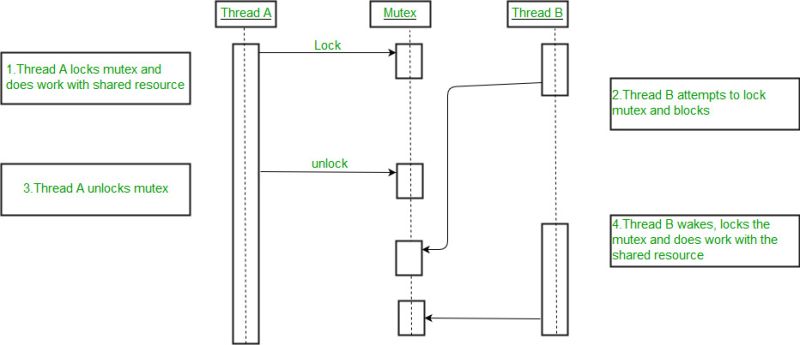
Mutex (short for mutual exclusion) is a synchronization mechanism that ensures exclusive access to shared resources. It is commonly used to coordinate access to critical sections of code, where multiple tasks or threads may compete for the same resources simultaneously.  
At its core, a mutex is a variable or data structure that can be in one of two states: locked or unlocked. When a task or thread wants to access a shared resource protected by the mutex, it must first attempt to acquire the mutex. If the mutex is unlocked, the task or thread successfully acquires it and can proceed to access the shared resource. However, if the mutex is locked, the task or thread is blocked, or it may choose to wait until the mutex becomes available.  
The following are key concepts related to mutexes in embedded system software:  
  
Mutual Exclusion: The primary purpose of a mutex is to provide mutual exclusion. It ensures that only one task or thread can access the protected resource at any given time. This property prevents race conditions, where multiple entities try to modify the same resource simultaneously, leading to unpredictable behavior.  
Ownership: A mutex can be owned by a single task or thread at any time. When a task acquires a mutex, it becomes the owner, and only the owner can release the mutex. Other tasks or threads can try to acquire the mutex, but they will be blocked until the current owner releases it.  
  
Blocking and Non-Blocking Operations: Mutexes can operate in either blocking or non-blocking mode. In the blocking mode, a task that attempts to acquire a locked mutex will be suspended until the mutex becomes available. In the non-blocking mode, a task will immediately return with an indication of whether it successfully acquired the mutex or not.  
  
Priority Inversion: Mutexes can be associated with priority inheritance or priority ceiling protocols to prevent priority inversion issues. Priority inversion occurs when a lower-priority task holds a mutex that a higher-priority task needs. This situation can lead to a situation where the higher-priority task is blocked by the lower-priority task, causing undesired delays. Priority inheritance and priority ceiling protocols aim to mitigate this problem.  
  
Deadlock: Care must be taken when using mutexes to avoid deadlock situations. Deadlock occurs when multiple tasks are waiting for each other to release the mutexes they hold, resulting in a state where no progress can be made. Techniques such as deadlock detection and avoidance algorithms can be employed to mitigate the risk of deadlock.  
Implementing a mutex typically involves using hardware support, operating system services, or software libraries specifically designed for concurrency control. Mutexes are commonly used in embedded systems to protect access to shared resources like memory, peripherals, communication channels, and other critical sections of code.  
[#embeddedsystems](https://www.linkedin.com/feed/hashtag/?keywords=embeddedsystems&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A7064524855303450624) [#software](https://www.linkedin.com/feed/hashtag/?keywords=software&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A7064524855303450624) [#data](https://www.linkedin.com/feed/hashtag/?keywords=data&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A7064524855303450624) [#communication](https://www.linkedin.com/feed/hashtag/?keywords=communication&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A7064524855303450624) [#automotiveindustry](https://www.linkedin.com/feed/hashtag/?keywords=automotiveindustry&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A7064524855303450624) [#automotivetechnology](https://www.linkedin.com/feed/hashtag/?keywords=automotivetechnology&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A7064524855303450624)

Activate to view larger image,



1. First, we check if the input number "n" is less than or equal to 0. If it is, we know that it cannot be a power of 2, so we return 0.  
  
2. Next, we perform a bitwise AND operation between "n" and "n - 1". This operation clears the least significant bit in "n" (i.e., the rightmost 1-bit) and leaves all other bits unchanged.  
  
3. If "n" is a power of 2, it will only have one 1-bit set in its binary representation (e.g., 2 = 0b10, 4 = 0b100, 8 = 0b1000, etc.). This means that when we subtract 1 from "n", we get a number with all 1-bits to the right of the rightmost 1-bit in "n" (e.g., if "n" is 8, "n - 1" is 7 = 0b111).  
  
4. Therefore, if "n" is a power of 2, the bitwise AND operation between "n" and "n - 1" will result in 0, since the only differing bit between the two numbers is the rightmost 1-bit, which is cleared by the bitwise AND operation.  
  
5. Finally, we return the result of the logical NOT operator applied to the bitwise AND operation. This converts the result to a boolean value: 1 if "n" is a power of 2, and 0 otherwise.  
  
That's it! This function uses simple bitwise operations to determine whether a given number is a power of 2. It's a useful trick to have in your toolbox when working with binary numbers in C.  
  
There might be better ways to do the same, so please feel free to suggest improvements.

Int isPowerOfTwo(int n)

{

If(n <= 0)

{

Return 0;

}

Return ! (n & (n-1))

}

1️⃣ Transistor to Analog/Digital Circuits: <https://lnkd.in/gMZBxiwn>  
  
2️⃣ Digital Circuits to Processor (+compiler) Design: <https://lnkd.in/emtqTtfS>  
  
3️⃣ OS Design and Internals: <https://lnkd.in/gGzcBttK>  
  
4️⃣ Communication System Design (all three): <https://lnkd.in/g_QmBarw>  
  
5️⃣ Embedded Systems-based Design: <https://lnkd.in/gsi3i4Hj>  
  
6️⃣ FPGA-Based Design: <https://lnkd.in/gHMkw-64>

<https://youtube.com/playlist?list=PLyYrySVqmyVPzvVlPW-TTzHhNWg1J_0LU>

I plan on deploying it on an FPGA. Lattice Hx8K is what I have with me. A few years back I developed a single-cycle CPU and deployed it on Intel's DE1SoC: <https://github.com/dstreetdog/mips-cpu>

[Connor Skelland](https://www.linkedin.com/in/connor-skelland-a52439122/) - Very much!! I see that it has 33k LUTs. The one I have only has 8k! The more the better. You might want to check this playlist - <https://www.youtube.com/playlist?list=PLEBQazB0HUyT1WmMONxRZn9NmQ_9CIKhb>

[Yogesh Bang](https://www.linkedin.com/feed/) - Very cool. We had done some experiments on Qemu for all popular processors: <https://github.com/google/esh>

Useful topics:

There are several online resources to choose from and sometimes it is confusing to know where to start. As requested by many of you, I have curated a list of video courses and online websites that will help you  
  
I have also added a bonus point at the end :) :  
  
If you are already good in some, you can skip it:  
  
1. Electronics Fundamentals  
  
Digital and Analog Electronic courses:  
  
<https://lnkd.in/enmbxaiK>  
  
Basic Electronics: <https://shorturl.at/dlMQ5>  
  
Digital Signal Processing: <https://shorturl.at/etNZ9>  
  
2. Computer Architecture  
  
Course by Princeton Uni: <https://shorturl.at/fntDN>  
  
Building Embedded Sys: <https://shorturl.at/inoGS>  
  
3. C Programming  
  
Basics:  <https://shorturl.at/bdl38>  
  
C Coding Standard: <https://shorturl.at/aguO0>  
  
4. RTOS  
  
Basics: <https://shorturl.at/nwFGN>  
  
5. Microcontroller  
  
Basics: <https://shorturl.at/enEOV>  
  
6. Git version control  
  
<https://lnkd.in/eDphv4pX>  
  
7. CMake  
  
Fundamentals: <https://shorturl.at/hiBMO>  
  
8. Embedded Systems ( Safety, Security )  
  
Professor Phil Koopman ( CMU ):  
  
<https://shorturl.at/kxG14>  
  
9. Linux :  
  
Fundamentals: <https://shorturl.at/muINY>  
  
Embedded Linux Wiki: <https://lnkd.in/eyAzbQXt>  
  
10. Embedded Systems podcast:  
  
<https://embedded.fm/>  
  
11. Modern Embedded Systems  
  
Miro Samek videos: <https://shorturl.at/pzEWZ>  
  
12. Project Ideas ( Bonus ) :  
  
<https://blog.adafruit.com/>  
<https://lnkd.in/eeYRWYWV>  
  
If you have any other recommendations that can help someone in their Embedded Software journey, then let us know in the comments.  
  
I will share more advanced courses in the upcoming posts.  
  
If you found this post useful, you can share it with your friends and connections and help them start their Embedded Careers.

. C language  
  
• Some important concepts to note are Pointers, Data structures in C, functions.  
  
• The C language can be tested either using online C simulator or a problem statement will be provided and you will be asked to submit it after the interview.  
  
2. Embedded systems basics  
  
• Communication protocols - UART, I2C, SPI, Ethernet, WiFi  
  
• Microcontrollers and Microprocessor differences  
  
• Memory management and different types of memory  
  
• Interrupts (For ex: Context Switching)  
  
• Computer architecture ( For ex: comparison between Harvard and Von Neumann architecture)  
  
• RTOS ( For ex: Scheduler, Threads, Tasks etc.)  
  
3. Linux Commands  
  
• Depending on the company product, you can be tested in depth in Linux OS (For ex: Memory paging, Memory management in Linux, Kernel etc )  
  
• Basic Linux commands could be tested (For ex: copying and moving files, search commands etc)  
  
4. Networking  
  
If the interview is in networking team, you will be required to know  
  
• The basics of of computer networking  
  
• Different layers of OSI Model (Open Systems Interconnection Model).  
  
5. Technical projects  
  
I have saved the most important part of the interview process to the last.  
  
• Relevant technical projects in C add a lot of value to your profile.  
  
• Showcasing your hardware and software knowledge in the projects will be an added value.

FPGAs and SoCs - [theeeview.com](http://theeeview.com/)

The one that I was very proud of was a CPU with 6 instructions deployed on an FPGA: <https://github.com/dstreetdog/mips-cpu>

~~~ 𝐀 𝐁𝐫𝐢𝐞𝐟 𝐢𝐧𝐭𝐫𝐨𝐝𝐮𝐜𝐭𝐢𝐨𝐧 𝐨𝐟 𝐒𝐞𝐦𝐚𝐩𝐡𝐨𝐫𝐞𝐬 𝐢𝐧 μ𝐂/𝐎𝐒-𝐈𝐈 ~~~  
  
For Tasks to inter communicate, they use a common resource which is usually referred to as “critical region” in memory. Sharing memory is useful and required by tasks but it is to be made sure by kernel that the shared resources are protected and not corrupted/changed by multiple tasks accessing it at once.  
  
From many ways, one of the ways to obtain exclusive access to shared resources is by disabling interrupts. μC/OS-II uses this technique by providing two macros 𝐎𝐒\_𝐄𝐍𝐓𝐄𝐑\_𝐂𝐑𝐈𝐓𝐈𝐂𝐀𝐋() and 𝐎𝐒\_𝐄𝐗𝐈𝐓\_𝐂𝐑𝐈𝐓𝐈𝐂𝐀𝐋() where you can keep the shared data/resources within those functions. These functions disable and enable interrupts respectively.  
  
It is to be noted that the interrupts shouldn’t be disabled for a longer amount of time as it might lead to interrupt latency. It is preferred to used this method only where there is a need to change/perform actions of few variables.  
  
One other way to obtain the exclusive access to by using Semaphores. By using semaphores, we can control the access to shared resources, the kernel can signal the occurrence of an event, allow two tasks to synchronize their activities. Semaphore is like a key for the tasks and the critical region is a lock. If a semaphore is in use, the requesting task will be suspended until the semaphore is released.  
There are two types of Semaphores,  
·       Binary Semaphores  
·       Counting Semaphores.  
  
As the name suggests, the binary semaphore can have only two values, 0 and 1. While, the counting semaphore has values ranging from 0 to 2^(No of bits its being implemented in, ex 0 – 255 for 8 bits).  
The operations performed by the Semaphores are:  
·       INITIALIZE(CREATE)  
·       WAIT(PEND)  
·       SIGNAL(POST)  
The waiting list of the tasks are initially empty and the initial semaphore should always be provided with a value.  
  
The process:  
The task that wants to perform semaphore action is in WAIT.  
If the semaphore is available/ value>0  
           The value is decremented and the task continues execution.  
If the value==0  
           Task performing wait on the semaphore is placed in waiting list.  
If the semaphore is not available within the required time, the task is made to run in ready state and an error code is returned.  
A task releases the semaphore by performing SIGNAL operation.  
If no task is waiting for the semaphore  
           Semaphore value is incremented  
If task is waiting for the semaphore  
           No increment is performed on the value and the task is ready to run.  
In μ𝐂/𝐎𝐒-𝐈𝐈 the tasks that receives the semaphore is highest priority task waiting for the semaphore. 𝐎𝐒𝐒𝐞𝐦𝐏𝐞𝐧𝐝() and 𝐎𝐒𝐒𝐞𝐦𝐏𝐨𝐬𝐭() are used by μC/OS-II for the usage of semaphores.

A segmentation fault in C occurs when a program tries to access a memory location that it is not allowed to access. This can happen due to various reasons, including:  
  
Subscribe my channel for more.  
<https://lnkd.in/dGdNRGkU>  
  
1. Null pointers: Trying to access or dereference a null pointer can result in a segmentation fault. Ensure that all pointers are properly initialized before using them.  
  
2. Out-of-bounds array access: Accessing an array beyond its bounds can lead to a segmentation fault. Make sure to validate array indices and ensure they are within the correct range.  
  
3. Memory leaks: Failing to deallocate memory properly can lead to memory leaks. If memory is continuously allocated without being freed, it can eventually cause a segmentation fault when the available memory is exhausted.  
  
4. Stack overflow: Excessive recursion or large local variables can cause the stack to overflow, resulting in a segmentation fault. Ensure that recursive functions have proper termination conditions, and be mindful of the stack space allocated to local variables.  
  
5. Invalid memory access: Accessing memory that has already been freed or memory that has not been allocated can cause a segmentation fault. It is essential to manage memory allocation and deallocation carefully.  
  
To debug and fix a segmentation fault in C, you can follow these steps:  
  
1. Compile with debugging symbols: Compile your C code with debugging symbols enabled, which will provide additional information during debugging. For example, using the `-g` flag with the compiler.  
  
2. Use a debugger: Utilize a debugger such as gdb (GNU Debugger) to step through your code and identify the line or function causing the segmentation fault. Set breakpoints, inspect variables, and examine the stack trace to understand the program's state when the fault occurs.  
  
3. Review the code: Analyze the code where the segmentation fault occurs. Look for any memory-related issues, uninitialized pointers, improper memory deallocation, or array access problems. Use printf statements or debugger output to trace the program's execution and narrow down the problematic section.  
  
4. Check for common mistakes: Double-check for common mistakes like off-by-one errors, incorrect loop conditions, and invalid pointer assignments.  
  
5. Enable compiler warnings: Configure your compiler to emit warnings for potential issues in your code. Address these warnings as they can often point to problematic areas that could lead to segmentation faults.  
  
6. Use tools for memory management: Utilize memory management tools like valgrind, which can detect memory leaks, buffer overflows, and other memory-related issues in your program.  
  
7. Divide and conquer: If the codebase is large, try isolating the problem by commenting out sections of code or creating a minimal, reproducible example.

I'm excited to share with you an update on my recent project of building a Real-Time Operating System (RTOS) from scratch. SysTick\_Handler is used to switch tasks Preemptive with Round Robin at the same Priority Task. This endeavor involved developing a set of key functions to provide a robust and efficient operating system for real-time applications. Let me briefly introduce some of these functions:  
  
1:MYRTOS\_Init:  
The initialization function that sets up the necessary system resources, data structures, and configurations for the RTOS.  
  
2:MYRTOS\_CreateTask:  
This function allows job creation within the RTOS. Prepares the task execution environment. MyRTOS\_Create\_TaskStack is called to create the stack.  
  
MyRTOS\_Update\_Schadule\_tables :  
The task that is in Ready state is moved to Ready\_QUEUE to be executed according to the highest priority.  
  
Decide\_whatNext:  
update Ready queue to keep round round robin Algorithm happen.  
  
PendSV\_Handler:  
In the cortex-m3 ,in switch context, some of the records are automatically regested, such as R0 to R3, R12, LR, PC, and PSR, and the rest is stored manually, and so on in the restore process.  
  
3:MYRTOS\_ActivateTask:  
With this function, tasks created using the previous function can be activated to start executing their designated code. It ensures proper task scheduling and allocation of system resources.  
  
4:MYRTOS\_TerminateTask:  
When a task completes its execution or reaches its termination point, this function is used to gracefully terminate the task and free any resources associated with it.  
  
5:MYRTOS\_STARTOS:  
This function serves as the entry point for the RTOS, triggering the system to start its operation and task scheduling.  
  
6:MYRTOS\_TaskWait:  
This function enables tasks to enter a waiting state for a specified number of system ticks. It allows efficient utilization of system resources and synchronization between tasks.  
  
7:MYRTOS\_AcquireMutex:  
Mutexes are essential for controlling access to shared resources. This function provides a way for tasks to acquire a mutex, ensuring exclusive access to the associated resource.  
  
8:MYRTOS\_ReleaseMutex:  
When a task finishes using a shared resource protected by a mutex, this function is used to release the mutex, allowing other tasks to access the resource.  
  
Building an RTOS from scratch has been a challenging but rewarding experience. It has deepened my understanding of real-time systems, task management, and resource allocation.

When working with the I2C (Inter-Integrated Circuit) protocol, several common issues or errors can occur. Here are some typical problems and approaches for debugging I2C issues:  
  
1. Addressing Issues:  
  - Verify that the device's I2C address is correct. Ensure that the device address is properly specified and aligned with the address used in the software.  
  - Check if multiple devices on the bus have conflicting addresses. Ensure that each device on the bus has a unique address.  
  
2. Bus Signal Integrity:  
  - Check for proper pull-up resistor values on the SDA and SCL lines. Inadequate or excessive pull-up resistance can cause signal integrity issues.  
  - Inspect the physical wiring of the I2C bus for any shorts, open circuits, or incorrect connections.  
  - Confirm that the I2C bus capacitance is within the allowed limits, as excessive capacitance can degrade signal integrity.  
  
3. Clock and Timing Issues:  
  - Verify that the clock frequency used by the master device matches the expected frequency of the slave devices.  
  - Check if the timing requirements, such as the setup and hold times for data, are met.  
  - Ensure that the I2C bus is not experiencing excessive noise or interference, which can affect timing.  
  
4. Communication Errors:  
  - Monitor for NACK (non-acknowledgment) responses from the slave devices. A NACK indicates a failure to acknowledge data and can signify issues such as incorrect addressing or communication timeouts.  
  - Confirm that the master device is properly sending start and stop conditions to initiate and terminate communication.  
  
5. Power and Ground Issues:  
  - Ensure that all devices on the I2C bus share a common ground.  
  - Verify that the power supply voltages are within the specified range for all devices.  
  - Check for any power supply fluctuations or voltage drops that may affect the stability of the I2C communication.  
  
To debug I2C issues, you can employ the following methods:  
- Use an oscilloscope or logic analyzer to monitor the I2C bus signals (SDA and SCL) and verify the signal waveforms, rise and fall times, and clock frequencies.  
- Print debug messages or status information in the software code to track the execution flow and identify any errors or unexpected behavior.  
- Temporarily isolate devices on the bus to identify potential conflicts or issues caused by specific devices.  
- Review the datasheets and documentation of the devices involved to ensure correct configuration and operation.  
  
By systematically addressing these common issues and employing appropriate debugging techniques, you can effectively identify and resolve I2C communication problems.

DMA :

Did you know that in an application CPU spends a significant amount of time just moving data from one memory to another?

Picture this: Imagine we've designed a mathematical genius capable of lightning-fast problem-solving, yet it spends most of its time engaged in load and store operations. Isn't that a wastage of its incredible capabilities?

Designers identified this problem and to improve this situation they came up with the idea of Direct Memory Access (DMA) controller.

DMA can be thought of as a specialized machine excellent at handling load and store tasks. Direct Memory Access, or DMA, is a peripheral that allows peripheral devices (such as network cards, hard drives, GPUs, and sound cards) to transfer data to and from the main memory of a computer without involving the central processing unit (CPU). DMA solves this problem by providing a dedicated pathway for data to flow directly between the source and destination, bypassing the CPU's intervention. It creates a pipeline for data, ensuring faster and more efficient transfers.

How Does DMA Work?

DMA operates like a specialized traffic controller for data moving in and out of memory. Here's a simplified breakdown of how it works:

Setup Phase: The CPU configures the DMA controller, specifying the source (where data is coming from) and the destination (where data is going to). It also sets the transfer size and other parameters.

2 Initiation: Once the setup is complete, the

peripheral device initiates a DMA request. This signals the DMA controller to take control of the memory bus temporarily.

3 Data Transfer: The DMA controller coordinates the

data transfer between the source and destination.

Completion: After the transfer is complete, the DMA controller signals the CPU. The CPU can then

process the transferred data as needed.

By offloading data transfers from the CPU, DMA minimizes the impact on CPU performance, allowing it to focus on more critical tasks. DMA transfers data in bulk, significantly speeding up the process compared to traditional methods that involve the CPU.

In essence, DMA serves as a silent but critical facilitator of the high-speed data transfers that underpin our digital experiences. Whether you're streaming a video, playing a game, or transferring files, DMA is there, working behind the scenes to create an efficient pipeline for dataflow, contributing to a smoother and more responsive computing environment.