Inter-Process Communication (IPC) with FreeRTOS

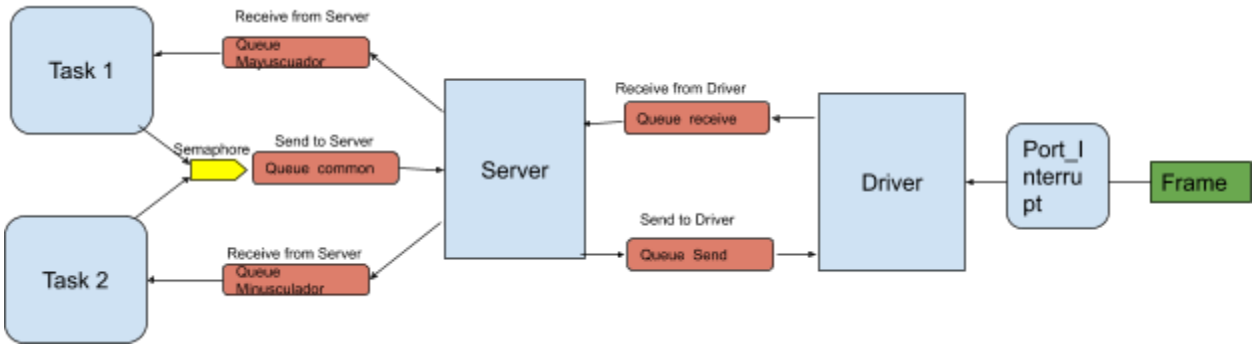
Purpose

This projects implements FreeRTOS to build IPC protocol on ARM-cortex 4.The RTOS runs 4 tasks;a Driver, Server, Task1 and Task2. The Driver receives the frame from the USB serial communication from ttyUSB device file inside Linux OS, and verifies if it is in a valid format, then passes the data part only to the Server which decides according to an operation byte at the beginning of the data received which task is in charge .

The data transmission between Driver and Server is established via two queues, one for sending and the other one is for receiving. The data is saved temporarily inside local buffers of each, they store temporarily the data before and after processing.

There is one queue for each task to receive data only from the Server and one common queue to send to the server between Task 1 and Task2.The shared queue is protected with semaphore to avoid data corruption. Each Task reports the current stack and heap size available before and after executing its instructions set .

The program also makes use of FSM (Finite State Machine) programming, two structures are created of different predefined types inside the service.h header file. The first one ,**program_memory,** stores the operation used, data before processing, data after processing and the message size. The second structure ,**Report**, is used to hold the stack and heap size of each task during run time, it is used to answer requests when the operation number is either 2 which means report total remaining stack size or 3 which means report available total remaining heap size. Both heap and stack sizes are updated at the at the beginning and the end of each task.



Installing

To run the program you need to have installed GCC compiler, go to the project folder then type the command “make”. To download it to your board, just type “make download”.You also need to install python to run the scripts.

How to use

There is a .py file for each operation a frame can have. After running the program, open another terminal and type ***python send_op0.py*** this will send the frame with “0” operation which results in executing task1.
Do the same thing by sending ***op1.py*** and **op2.py** to experiment with the rest.

The Frame

The frame is basically a series of hexadecimals characters that represent the following message “***This is rtos course TP1 in test, and it is working !”***. In addition to the message some more information is added like SOF (Start Of Frame), EOF(End Of Frame) , the data size and the operation to be performed.
Thus, the final frame should look like this :
{150This is rtos TP1 in test, and it is working}
Assuming the operation is 1 and the size is 50. The hexadecimal representation looks like this
“7B31353054686973206973205544F5320636F7572736520545031206E20746573742C616E642069742069732077 6F726B696E67217D”

“{” : Start of Frame
“1” :The operation byte
“50” Data size
“Message”

“}”: End Of Frame
The Driver validates the frame by checking ony SOF and EOF, removes SOF and EOF then it sends it to the Server. The server looks at the operation byte, sends the data section only to the task, receives it back, adds operation and size bytes back in and forwards it to the Driver.

Libraries

The project contains the following header files :
FreeRTOS.h
FreeRTOSConfig.h
Sapi.h
Task.h
Queue.h
Server.h
Driver.h
Operation.h
Server.h,driver.h and operation.h are personal libraries that were created for this project and include functions to extract data, convert from Hex to ASCII, convert to upper and lower case and more.

Memory management

The memory management algorithm used is **heap_2.c** since it allows allocation and freeing of memory and since I am not concerned about relalocation because the same memory size is the same, there should be any sort of internal nor external defragmentation. Some local buffers are stored inside heap memory and cleared out upong task completion.

Demo

Attached inside the main folder is a demo of how the program runs and the results that should come out**.**

Interrupt

Once a frame is received at the device file, an interruption if fired to read the data and save it inside a HexFrame buffer to be converted to ASCII readable letters and saved inside AsciFrame buffer for processing. The frame reading happens in sections that is why inside the python scripts the frame is divided into portions of 15 bytes with a break of 100 milliseconds to allow the reading from ttyUSB device file
The reason it is divided in portions of 15 bytes is that the handler does not allow a reading of more than this quantify at a time.
Every time a new frame is received, the driver disables the interrupt if the frame is valid and it is enabled back on at the end of each operation completion, and the interrupt counter is also reset back to cero to start saving in the buffer from index 0.

Convention

For convention, the program constantly display notification at the most relevant events inside the program in the following notation:
Server -> Driver : means Server to DrIver.
Server <- Driver : means Server from Driver.
Server -> Report : means Server reporting or updating state.
Task <- Server: Task from server .
Following that is what really took place, it could be received nothing, it could be sent a message, or a confirmation
For example; **Server -> Driver**: Sent this means that the message was sent from the Server to the Driver.

Error Handling

The code bares error handling at almost every critical point to verify the flow, but most of it is grayed out so that the screen does not get filled with unnecessary notifications.