

0.1 FPGA

The purpose of this section is to discuss some of the challenges and benefits that is related to the use of an FPGA in this project. The FPGA connects the microcontroller all the components on the pan-tilt system in order to be able to control the system. It collects the data from encoders and homing sensors, and it creates and updates the PWM signal to each of the H-Bridges.

One of the main differences between an FPGA and a microcontroller is that on an FPGA everything is implemented in hardware. This also means that all of the processes running, is running in parallel. When implementing the different components this needs to be taken into consideration.

The FPGA used in this project is a Artix7 mounted on a BASYS3 kit.

refrace a bit

ref

0.1.1 System overview

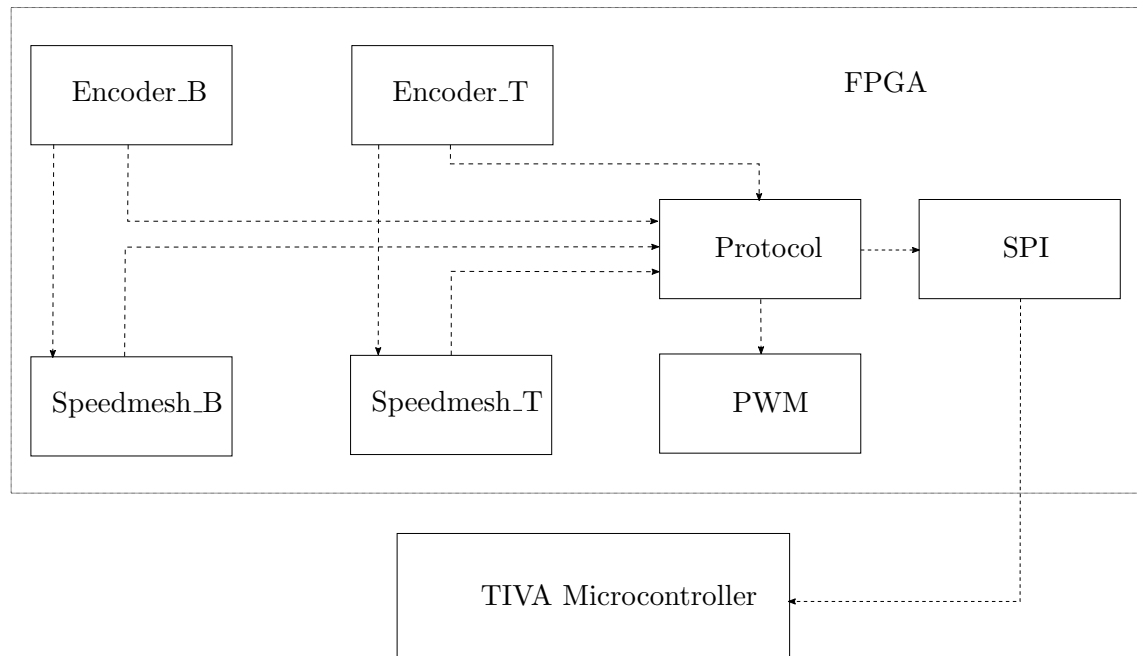


Figure 1: *System overview.*

The system consist of different modules, implemented in VHDL. To store the data that needs to be exchanged between the FPGA and the Tiva microcontroller, different registers is created with the necessary size. Se table 1.

The connection between the modules can be seen in the system overview. Se figure 1

12 Bit	9 Bit	1 Bit
Position_T	PWM_T	Home_T
Position_B	PWM_B	Home_B
Velocity_T	-	Reset_T
Velocity_B	-	Reset_B

Table 1: *FPGA Registers*

0.1.2 PWM

The PWM module creates a PWM signal for both motors. It takes a 9 bit vector as input for each motor, and output the signals necessary for the H-Bridges to work. It takes the 100Mhz system clock from the FPGA as a input, and then the module contains a clock divider that creates the necessary clock frequency, for the desired PWM frequency. MSB in the 9 bit input vector determine witch way the motor should turn. The module is design so that it is possible to just define the system clock frequency and the desired PWM frequency, and the correct divider is calculated when the code is synthesized. This is done so that it is easy to change the PWM frequency for testing.

0.1.3 Protocol

The protocol module uses a SPI module created by for communicating with the Tiva microcontroller. The role of the protocol module is to decode the data received over SPI, and prepare data for transmission, in regards to the previously determined protocol. Since all the different processes, like receiving data over SPI and updating data registers, all run in parallel, it is important to make sure that the protocol module cannot update the data registers while the SPI module is receiving data form the microcontroller. This is done by placing a latch on the output from the protocol module, witch is triggered by a busy signal from the SPI module.

ref.

0.1.4 Quadrature decoder

0.1.5 Speed measurement

0.2 Microcontroller Operating System

The purpose of this section is to detail how the operating system for the Tiva microcontroller is setup and why. The responsibility of the Tiva is to act as the brain of the entire system while the FPGA acts like a data acquisitions device and a hardware controller. The guiding reasoning behind the choice of operating system is the need to run more then one task on the system. The system FreeRTOS is chosen partly due to the fact that this system is part of another course and therefore beneficial to be familiar with, but also due to allowing a preemption scheduler as opposed to a Run-To-Complete scheduler. At the time of choice it was not clear whether the preemptive capability was going to be decisive and FreeRTOS provides both.

FreeRTOS, an open source RealTime operating system created by Real Time Engineers LTD handles system processes of different types by creating "Tasks" that are the run using a preemptive scheduler which allows the system to not only run multiple tasks in "parallel"¹ but also efficiently allocate the systems resources available.

A task is a single area of responsibility that either by necessity or by convention is combined as a single logical unit. Each task in the system (a full overview can be seen in the task diagram) can then be scheduled based on the priority it has been assigned and whether the task itself has decided that it wants to be scheduled.

¹Due to only one core being supported, true parallel is not possible.

0.2.1 Scheduling

In order to enable the microcontroller to run more than one task some form of scheduling must be implemented. This allows the operating system² to make a choice as to what tasks needs processing capacity at this moment in time. In the FreeRTOS based system running on the Tiva scheduling is done through a priority based preemption capable implementation. Preemption is when the operating system has the ability to pause execution of a process without the process willingly giving up CPU time. This allows time slicing without having the need for individual processes to be designed in a way to accommodate this. Further this allows the system to equally share time between long running processes.

The preemption capability enables the system to respond quickly to higher priority tasks becoming available for execution rather than waiting for the current task to return on its own potentially taking the CPU forever, it will be paused and the higher priority task executed before the lower priority one is resumed and potentially completed.

As mentioned the priority allows the system to select more important³ processes to be run quickly when available.

The scheduler is run by a timer interrupt every 5ms and during this the current task is paused and the scheduler evaluates the next eligible task to run, this may be the same task that was running before and this will then be resumed, but the scheduler must be run every 5ms tick to ensure correct handling of priorities.

0.2.2 Subtask Communication

The multiple task architecture where each task deals only in their own specific area of the system requires, since the combination of tasks are designed to accomplish one or more high level objective, a way to efficiently communicate between tasks running on the system. This is accomplished using a combination of state buffers and queues.

State buffers are simple buffers keeping the last value written to it until a new value is written regardless of reads performed in that time. The implementation of this concept is done through variables of the appropriate task being declared in global scope and therefore available to every function in the system.

All queues in the system are FIFO⁴ queues allowing easy exchange of data between different parts of the system, while maintaining loose coupling between components. This loose coupling allows more parallel development and a more robust end product.

To maintain data integrity all shared-write resources are protected with semaphores allowing a task to obtain exclusive access to a resource and be able to update it without risk of data corruption caused by interrupted write access. These semaphores are done via the implementation that ships with FreeRTOS and allows both binary and counting semaphores, although only the binary implementation is utilized.

0.2.3 Tasks

The entire system has been created as a task diagram in figure 2.

There is a number of task that exists only to provide services to other tasks, those tasks are designated system tasks.

Each task is designated a priority⁵ based on the demands of that particular task. The biggest consideration is the time sensitivity of the task, the need for timing to be consistent

²FreeRTOS

³Defined by the programmer

⁴First-In-First-Out

⁵by the designer

between runs. FreeRTOS in theory supports as many priority levels as RAM is available to handle. Therefore the recommendation from FreeRTOS is to not have more levels than needed. The system uses three levels in this implementation, LOW, NORMAL and HIGH. With HIGH providing the best timing accuracy due to the fact that is a HIGH priority task ready to run, then it will be run at the next scheduler run without having to wait for other tasks to potentially use up a time slice before its run.

Status LED task This task only exist to toggle an LED on the board to verify that the OS is running and has not encountered some problem that could not be resolved automatically. Therefore the task has been assigned a low priority and will be processed with spare capacity.

UART Tasks This group consists of the Transmit and Receive task which are created completely independently. The TX task accepts data via a queue and sends that out the system via UART. The RX task is the inverse of this. The tasks are a part of making the system work and therefore run with normal priority.

LCD Task The LCD task takes items from a queue and adds them to the currently active line on the LCD display. The task is run with low priority as this only need to provide feedback in human observable time frames.

Matrix Keypad Task This task simply adds key presses from the keypad to a queue available to the rest of the system. Is being run with low priority as immediate response is not required for function.

Serial Peripheral Interface Task The SPI task handles all communication between the microcontroller and the FPGA and thus is indirectly responsible for all data acquisition and hardware control. This requires a priority of normal so the task is prioritized but does not require hard timing.

Besides system tasks there are tasks that can be categorized as being the "application" run on the system. These tasks are designated "user tasks".

PID This group of two tasks handle the control for both axis of the pan-tilt system and are the most critical processes. These run with high priority and therefore the best timing reliability that the system can offer.

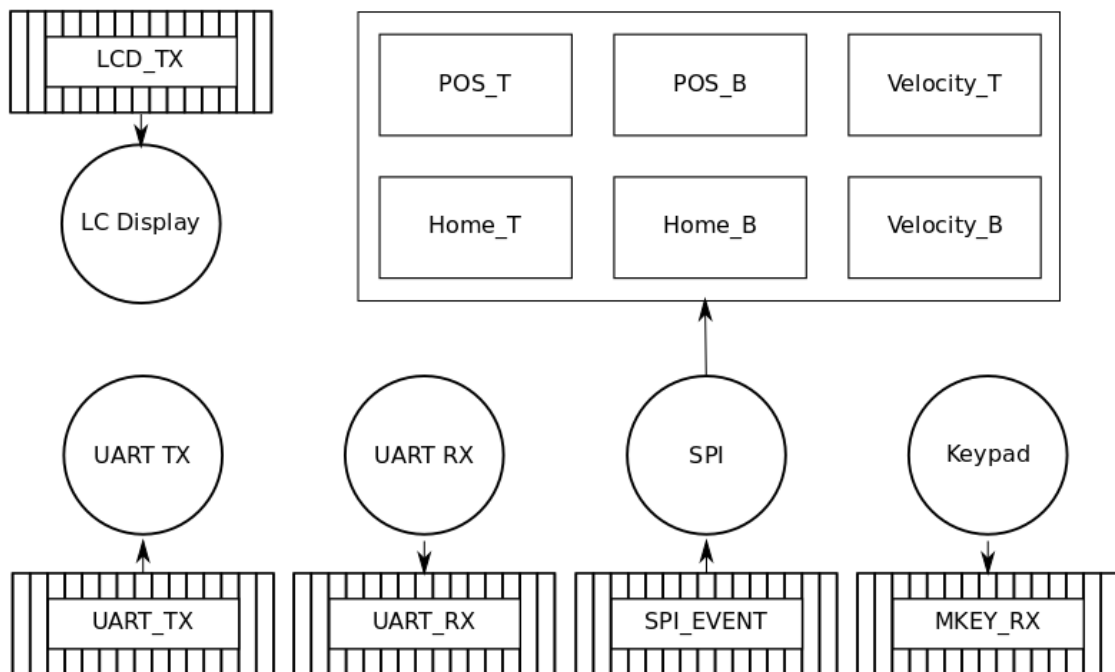
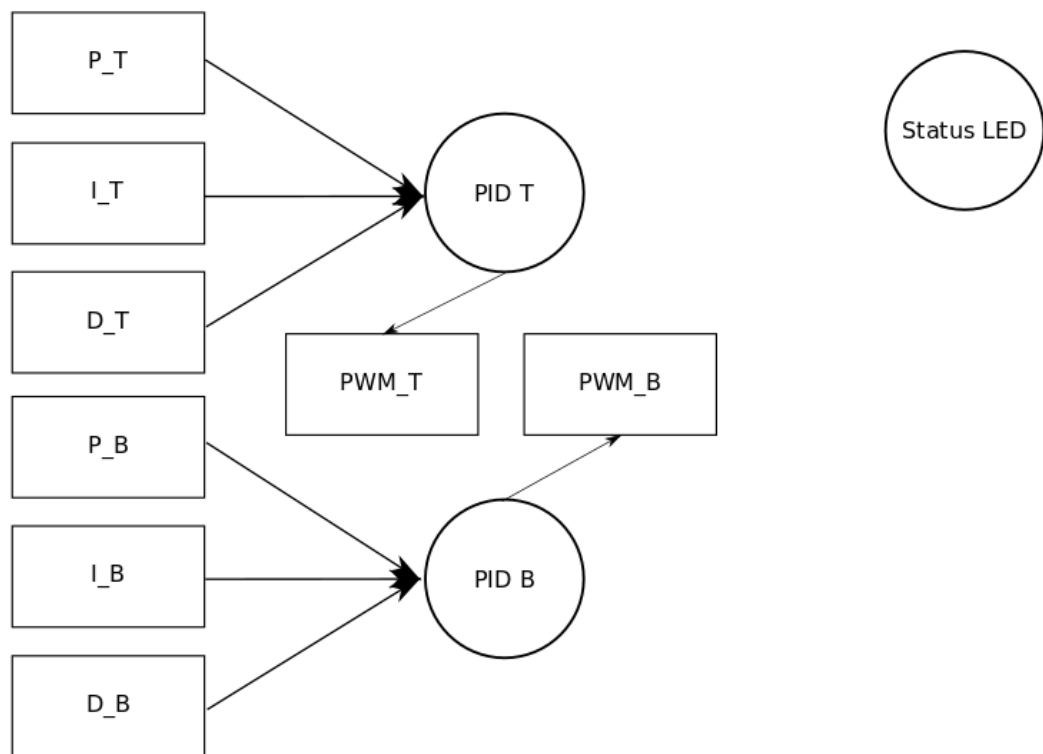


Figure 2: *Task Diagram of entire system*
