FreeRTOS Binary Clock on the ATmega328P Arduino Platform

ED6422 – Real Time System Design

Graham Claffey, Student Member*
18296661
University of Limerick, Co. Limerick, Ireland
Email: 18296661@student.ul.ie

Abstract – The goal of this project is to create a binary clock on the ATmega328P Arduino platform, using an RTOS (Real-Time Operating System) to manage all the functionality under the hood. The RTOS that was chosen for this project is FreeRTOS. We will use this RTOS to send our binary clock calculations to an 8x8 LED Matrix to display the time in a binary format. The LED Matrix chosen for this project is the Max7219. The software implementation for this project will be using Atmel built-in Instruction Set for their chips known as AVR. The AVR instruction set is usually coded in C and is commonly referred to as AVR-C.

Keywords – Atmega328P, Arduino, AVR, Binary Clock, C, FreeRTOS, Max7219

I. INTRODUCTION

A binary clock is as the name suggests a clock that displays its time in a binary format, either 1 or 0, on or off etc. In this case the display will be on the LED matrix, we determine the time by viewing which state the LED is in. Each row of the 8x8 LED matrix will correspond to a certain time value on a regular base10 clock, for instance the first row will be our seconds, second row will be minutes, third will be hours, forth will be days etc. So the rows will account for our time values and each of the columns on the 8x8 LED matrix will have a weighted value representing the number of states the LEDs of that row can be lit before that column lights up. To paint a clearer picture, starting from the right more LED of any row its weighted value is 1 because there is only one state it can be in before it lights up. These weighted column values from right to left is represented in binary as 2 to the power of column number, the right most column being 1 and the left most being column 7.

The other component of this project is FreeRTOS running on the ATmega328P Arduino platform. The ATmega328P we are using is implemented on an Arduino UNO board, this will be were the FreeRTOS Kernel will be installed and use as a so called "base of operation" or physical hub from where we can connect our physical devices. The Adruino will be connected via USB cable to a PC/Laptop which acts as the source of power for the Arduino and all the devices connected to it. The USB will also act as our method to communicate between the PC and our physical media. Our Max7219 will also be connect to the Arduino.

II. HARDWARE

A more in-depth look at the hardware used in this project. Here we will see the inner-workings of the Arduino and the Max7219 to get a better understanding of the capabilities and limitations of both of these components.

Max7219

The Max7219 we have been referring to is actually named because of the chip used in this component, which is the Max7219CGN from MAXIM. The physical make-up of this component involves a PCB (printed circuit board), the Max7219 chip, an 8x8 LED matrix, 5 output (right in Fig. 1.) and 5 input pins (left in Fig. 1.).

Since the output pins are not used we will look at the input pins.

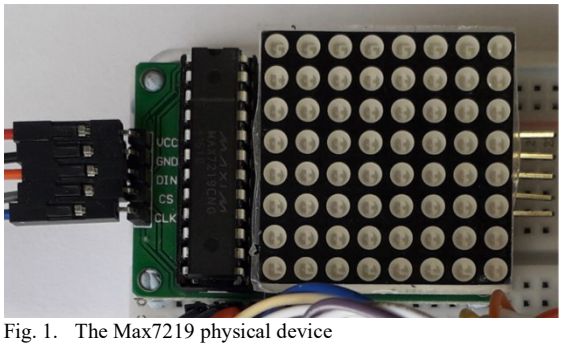


Fig. 1. The Max7219 physical device

The 5 input pins we use in this project are: [1]

1. VCC – Positive supply voltage. Connect to +5V.
2. GND – Ground.
3. DIN – Serial-Data Input. Data is loaded into the internal 16-bit shift register on CLK's rising edge.
4. CS – Chip-Select Input. Serial data is loaded into the shift register while CS is low. The last 16 bits of serial data are latched on CS's rising edge.
5. CLK – Serial-Clock Input. Maximum rate is 10MHz. On CLK's rising edge, data is shifted into the internal shift register. On CLK's falling edge, data is clocked out of DOUT.

Now that we have an understanding of the pins used we can take a look at the inner workings using a functional diagram of the Max7219.

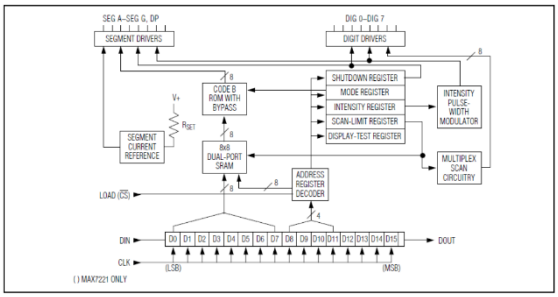


Fig. 2. Functional diagram of the Max7219

ATmega328P Microcontroller

The ATmega328P Microcontroller used in this project is a SMD (Surface Mounted Device) package. Mounted directly to the Arduino UNO board were the 32 pins are connected to the various peripherals on the board. It is an 8-bit AVR RISC-based microcontroller that runs at a frequency of 16MHz. A more detailed breakdown of the specifications [2]:

- Program Memory Type: Flash
- Program Memory Size (KB) 32
- CPU Speed (MIPS/DMIPS) 20
- SRAM (KB) 2,048
- Data EEPROM/HEF (bytes) 1024
- Digital Communication Peripherals 1-UART, 2-SPI, 1-I2C
- Capture/Compare/PWM Peripherals 1 Input Capture, 1 CCP, 6 PWM.
- Timers 2 x 8-bit, 1 x 16-bit
- Number of Comparators 1
- Temperature Range (°C) -40 to 85
- Pin Count 32
- Low Power Yes
- Operation Voltages 1.8-5.5V
- A/D converter 10-bit

The device also has a programmable watchdog timer with an internal oscillator, and five software sel ectable power saving modes.

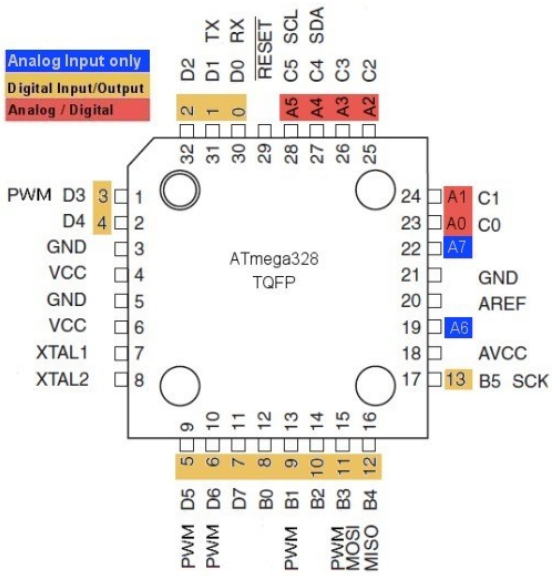


Fig. 3. ATmega328P pinout diagram

III. FREERTOS

FreeRTOS is a real-time operating system kernel for embedded systems. FreeRTOS also acts like an API allowing users to implement and customise their very own projects using FreeRTOS. FreeRTOS is free open-source software distributed under the MIT Licence. FreeRTOS is designed to be small consisting of only a few files, mainly written in C, but there are a few assembly instructions included.

The Setup of FreeRTOS for this project came with some considerations such as, FreeRTOS is not directly setup to be compatible with the ATmega328P, but there is an officially supported microcontroller that is very similar to the ATmega328P. This is the official ATmega323 port, so what we needed to do was to customise this port to suit our needs with the ATmega328P. We downloaded the FreeRTOS kernel and then stripped out all the unnecessary files that were either for specific microcontrollers or implementation of features that we did not require.

File changes

Two files that will be talked about due to the changes that were made for this project are:

- *port.c*
- *FreeRTOSConfig.h*

port.c – The changes that were made in this file were related to timers and output compares.

```
/* #define portCOMPARE_MATCH_A_INTERRUPT_ENABLE    ( ( uint8_t ) 0x10 ) */
#define portCOMPARE_MATCH_A_INTERRUPT_ENABLE    ( ( uint8_t ) 0x02 )
```
Fig. 4. portCOMPARE_MATCH_A_INTERRUPT_ENABLE

Also one of the Timers need to be changed for our needs.

```
/*
ucLowByte = TIMSK
ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
TIMSK1 = ucLowByte;
*/
ucLowByte = TIMSK1;
ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
TIMSK1 = ucLowByte;
```

Fig. 5. TIMSK changed to TIMSK1

Finally for port.c the timer1 compare vector was changed from signal output compare.

```
/*
 * void SIG_OUTPUT_COMPAE1A( void ) __attribute__ ( ( signal, naked ) );
 * void SIG_OUTPUT_COMPAE1A( void )
 * {
 *      vPortYieldFromTick();
 *      asm volatile( "reti");
 * }
 */
void TIMER1_COMPA_vect( void ) __attribute__ ( ( signal, naked ) );
void TIMER1_COMPA_vect( void )
{
    vPortYieldFromTick();
    asm volatile ( "reti" );
}
```

Fig. 6. Timer 1 Compare Vector changed

FreeRTOSConfig.h – As for the configuration file for FreeRTOS, we added/switch on features are they were needed in relation to the project. The FreeRTOSConfig.h file comes with a host of settings to tweak. Here is what the final product of that tweaking looked like after the project was finished[3].

```
#define configUSE_PREEMPTION                    1
#define configUSE_IDLE_HOOK                     1
#define configUSE_TICK_HOOK                     0
#define configCPU_CLOCK_HZ                      ( ( unsigned long ) 16000000 )
#define configTICK_RATE_HZ                      ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES                    ( 4 )
#define configMINIMAL_STACK_SIZE                ( ( unsigned short ) 85 )
#define configTOTAL_HEAP_SIZE                   ( (size_t ) ( 1500 ) )
#define configMAX_TASK_NAME_LEN                 ( 8 )
#define configUSE_TRACE_FACILITY                0
#define configUSE_16_BIT_TICKS                  1
#define configIDLE_SHOULD_YIELD                 1
#define configQUEUE_REGISTRY_SIZE               0

/* User defined configurations. */

#define configSUPPORT_DYNAMIC_ALLOCATION        1

/* Co-routine definitions. */

#define configUSE_CO_ROUTINES                   1
#define configMAX_CO_ROUTINE_PRIORITIES         ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               0
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskCleanUpResources           0
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
```

Fig. 7. FreeRTOSConfig.h setup at the end of the project

API Features Used

In this project we used a variety of API features to help implement the creation, running and display of the binary clock. Some examples of the features used[4][5][6][7]:

- ISR() – Interrupt Service Routine.
- SemaphoreGiveFromISR() – to give the Semaphore Handler.
- SemaphoreTake() – to take the given semaphore.
- xSemaphoreCreateBinary() – to create a binary semaphore.
- xQueueCreate() – to create a queue.
- xQueueSend() – to send data to the queue.
- xQueueRecieve() – to receive data from the queue.
- xTaskCreate() – to create our tasks.
- vTaskStatScheduler() – to start the task scheduler.
- vTaskDelay() – to delay the task for a specific amount of time.
- QueueHandle_t – setup a queue handler
- xSemaphoreHandle – setup a semaphore handler.

IV. SOFTWARE DESIGN IMPLEMENTATION

Overview of the Software design behind the binary clock project.

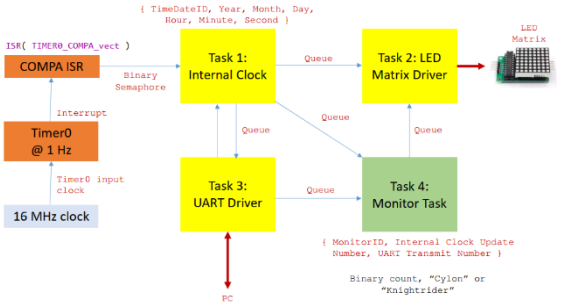
We start with the 16MHz clock (F_CPU the ATmage328P default clock speed). Using Timer0 which is an 8-bit timer, we use prescaling and overflow to achieve a frequency of 1Hz. With Timer0 set at 1Hz we can use this to do an output compare vector using the ISR. This causes the ISR to trigger every second. When the ISR triggers we send a binary semaphore from the ISR to Task1.

Task1 is responsible for clock calculations, were it sends the results of these clock calculations a queue via xQueueSend().

Task2 is responsible for the LED Matrix Driver, it waits until there is some data in the queue, where then it will receive that data via the xQueueReceive() API call. Using the data from the queue, Task2 can format the data to be output to the Max7219.

Task3 is responsible for the UART Drivers, using the values of the global variables which are being updated every second, to first convert the data into a char of a base10 representation using the stdlib.h function itoa() and then copy this data into a buffer to be send via the UART.

Task4 is responsible for Monitor Task, this is the status LEDs located at the last to rows, row 6 and 7.



V. CONCLUSION

In conclusion this has been an extremely informative project in relation to learning about FreeRTOS and programming on the ATmega328P using the AVR ISA. As an introductory project to RTOS design and development it managed to succeed in not only bolstering my interest in seeing what kind of other projects that I can create, but also in improving my skills within a multi-task environment. I'm already looking forward to my next project which would be to implement multiple Max7219 on an ARM Cortex chip and creating a binary, digital base10 and Temperature display in which the user can switch between the different modes of operation. The value of this project turned out to be very successful.

REFERENCES

[1] MAXIM Integrated datasheet - https://datasheets.maximinte-grated.com/en/ds/MAX7219-MAX7221.pdf

[2] Microchip - https://www.micro-chip.com/wwwproducts/en/ATMEGA328P

[3] FreeRTOS Configuration - https://www.fre-ertos.org/a00109.html

[4] FreeRTOS Task Creation - https://www.fre-ertos.org/a00019.html

[5] FreeRTOS Task Control - https://www.fre er-tos.org/a00112.html

[6] FreeRTOS Queue Management - https://www.freertos.org/a00018.html

[7] FreeRTOS Semaphores - https://www.fre-ertos.org/a00113.html