Kettering University
College of Engineering
Department of Electrical and Computer Engineering
Project Report

Course No. & Title: CE-426 Real Time Embedded Systems

Instructor's Name: Dr. Girma Tewolde

Project Title: GPS Controlled Clock

Date Submitted: 3/18/22

Team Members:

Noah Maceri

Thomas McCarty

Jonathan Myny

INSTRUCTOR SECTION

Comments: _____

Grade: Team Member 1: _____

Team Member 2:_____

Team Member 3:_____

TABLE OF CONTENTS

Background Info **2**

Project Objectives **2**

Hardware Used **2**

Implementation **5**

Conclusion **10**

References **11**

Background Information

GPS clocks are integrated systems that connect via a satellite network in order to provide various reports of information, most commonly time and location. Traditionally, GPS clocks are relatively expensive. Most clocks on the market use an RF signal to synchronize their clocks which is not as accurate as GPS. For the above reasons, we sought to create a GPS controlled clock that outputs various technical data.

Project Objectives

The following statements were the main goals that the project team set to ensure successful creation and implementation of our clock:

Use cost efficient hardware

Create a unique and visually appealing interface

Implement all backend data handling using ARM's RTX5 RTOS

Provide more information than a standard desk clock can

Hardware Used

To begin, the platform we chose was the Raspberry Pi Pico. The Pico features a dual-core Arm Cortex-M0+ processor with a flexible clock running up to 133 MHz ("Pico Specs"). This CPU is overclockable to 270MHz. It also features 264KB on-chip SRAM, 2MB on-board QSPI Flash, and 26 multifunction GPIO pins, including 3 analogue inputs. This platform was more than capable to build our GPS clock on.

Next we selected our GPS module. Waveshare's Pico-GPS-L76B was selected. This hat fits the pico slots directly on top and provides access to the GPIO below. This was incredibly important since we needed to place a screen on top of this module. As for the module itself, it communicates exclusively over UART. It uses formatted NMEA packets to send and receive data. Below you will find the NMEA packet format. It is important to understand the format of these packets to fully understand the code.

2.1. Quectel NMEA Packet Format

Preamble	TalkerID	PktType	DataField	*	CHK1	CHK2	CR	LF

Packet Contents:

Preamble: One byte character.
'$'

TalkerID: Two bytes character string.
"PQ"

PktType: 1-10 bytes character string.
An identifier used to tell the decoder how to decode the packet.

DataField: The DataFileld has variable lengths depending on the packet type.
A command symbol ',' must be inserted ahead of each data filed to help the decoder process the DataField.

* : 1 byte character.
The star symbol is used to mark the end of DataField.

CHK1, CHK2: Two bytes character string.
CHK1 and CHK2 are the check sum of the data between Preamble and '*'.

CR, LF: Two bytes binary data.
The two bytes are used to identify the end of a packet.

NOTE

The maximum length of each packet is restricted to 255 bytes.

Figure 1: NMEA Packet Format (Quectel)

The GNSS module uses pin 0 as UART TX and pin 1 as UART RX. Below you will find the complete pinout we utilized.

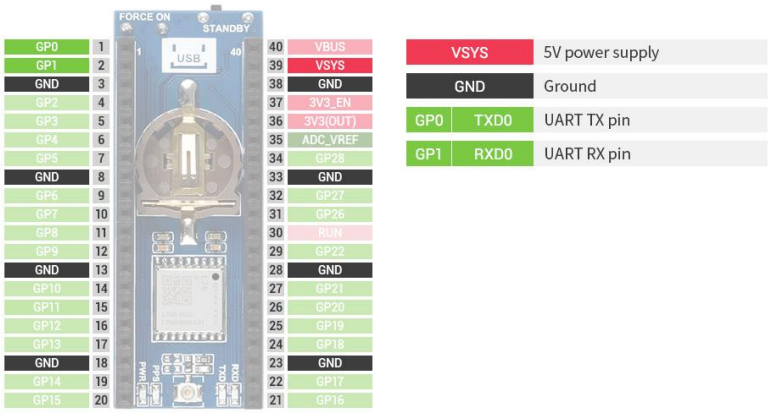

Figure 2: Pico-GPS-L76B Pinout (Waveshare)

This module is quite simple, only requiring 4 pins to function as intended.

Next we selected our screen. Opting to continue with Waveshare's hardware for the best compatibility, we selected the Pico LCD 1.3". This screen communicates exclusively over SPI so it does not interact with the GPS module. It features a 240x240px screen with 4 buttons and a digital joystick. The pinout of the LCD screen is significantly more complicated than the GPS module.

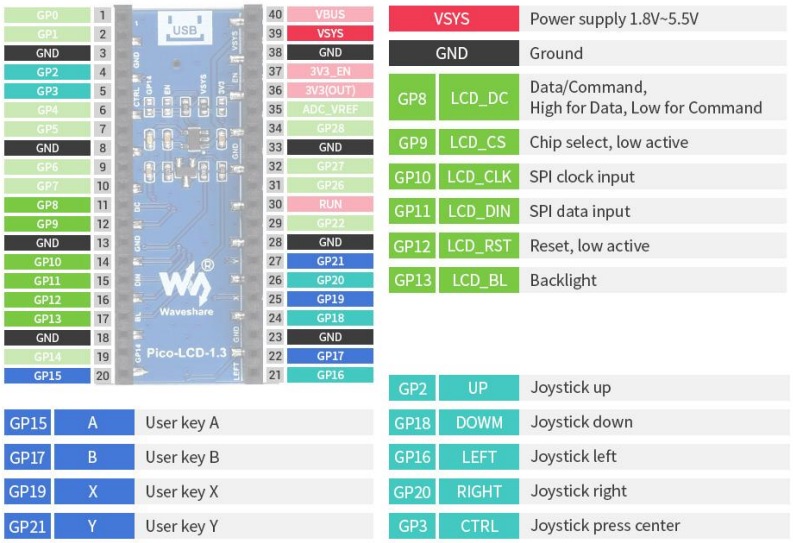

Figure 3: Pico LCD 1.3 Pinout (Waveshare)

All the hardware components in total cost just around ~$29, significantly less than any other GPS clock on the market. When all assembled the device fits a small form factor, perfect for sitting on any desk or nightstand. The LCD is quite bright and we chose to use white text on a black background for ease on the eyes.

A picture of our fully assembled clock is shown below.

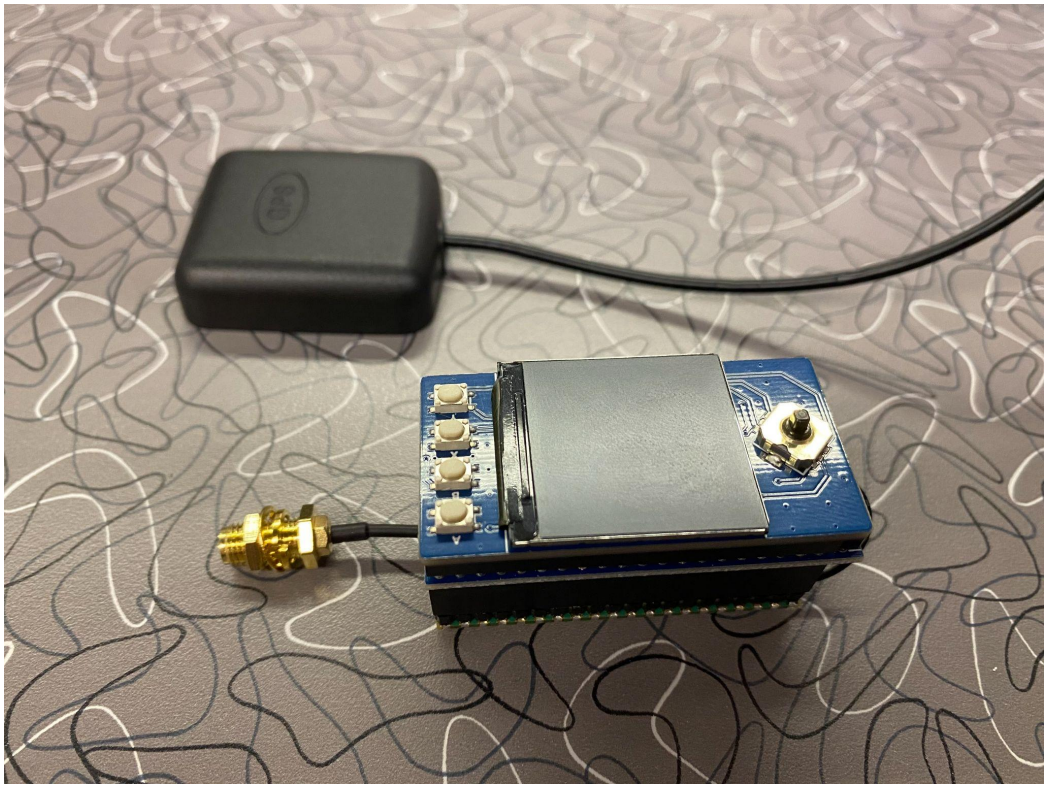

Figure 4: Fully assembled GPS Clock (With antenna shown but detached)

Project Implementation

All the code is implemented in ARM's Keil MDK. We used a support template provided by Gabriel Wang, an ARM employee. This template provided full support for the Raspberry Pi Pico within MDK. It took over the Pico's SysTick, hardware interrupts, and wrapped all MDK specific calls (EventRecorder, etc.) with Pico equivalents. This template also included a patch for RTX5 which allowed us to implement the RTOS on our Pico (Wang). As for debugging, we used a library called pico-debug by Peter Lawrenece. Since we have a dual core device and are only utilizing one core for our code, we can utilize the other core as a debugger. Pico-debug does exactly that, it leaves core0 for code and turns core1 into a CMSIS-DAP debugger. This DAP is fully compatible with MDK (and any other program that can read CMSIS-DAP debuggers). With this debugger we can execute code line by line, get stdout directly to a console in MDK and view RTX thread and memory information (Lawrence). One caveat of this debugging technique is that all code must be loaded into ram. This means all static variables and the machine code itself is stored in RAM. This was quite a major blocking point in our development since the Pico only

features 256Kb of RAM and our code plus RO and RW data sat at around 245Kb (plus room for the CMSIS-DAP debugger itself!). We did end up resolving this by adjusting the frame buffer size and adding more dynamic memory allocation. Overall, this platform allowed us to quickly build the code for our clock.

The code begins with the initialization of the RTX5 kernel. Once the kernel is initialized, we create a single setup thread, our thread switch signals, our message queue, and a mutex to protect the LCD's frame buffer. The setup thread does the following:

- Prints the RTX5 Kernel information and the amount of RAM in usage
- Sets up the SPI (Sets PWM duty cycle)
- Sets up the LCD screen buttons and joystick
    - Creates hardware interrupts for each GPIO pin
- It tests the LCD by alerting the user initialization has begun
- It sends all relevant commands to the GNSS module over UART, raising its baud rate when required
    - `"$PMTK251,115200*1F\r\n"`
        - Set baudrate of GNSS module to 115200
    - `"$PMTK220,400*2A\r\n"`
        - Update position every 400ms
    - `"$PMTK314,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0*28\r\n"`
        - Set NMEA sentence output frequencies (RMC and ZDA only)
    - `"$PMTK255,1*2D\r\n"`
        - Enable fixed NMEA output times behind PPS function
- After the GNSS module is configured, the setup thread creates 3 other threads
    - clock_screen_thread - The main screen of the clock, displays time, date, and lock type
    - location_screen_thead - Left screen, displays latitude and longitude
    - techdata_screen_thread - Right screen, displays number of satellites in view, altitude, and lock type
    - These three threads all then immediately suspended, apart from the clock screen thread

- The setup thread then terminates itself

At this point a screen thread is running. The GNSS module is sending the formatted data over UART once every 400ms and our code is flushing the UART FIFO queue to a string every time it is filled. This raw data is then passed to a processing function called L76X_Update. This function takes all the raw UART data, reads and processes it into variables, and then passes all the data, via a struct, into our sat_data message queue. This message queue is then read by the currently running screen thread and formatted into the frame buffer (Using the provided LCD library). The frame buffer is then flushed to the LCD (with mutex protection) and cleared. Finally, the running thread checks if it has received a signal to start another thread. If it has, it resumes the requested thread and suspends itself. These thread trigger signals are generated from the GPIO ISR. One note is that the ISR is a Pico based (Pico SDK) ISR. This is to interrupt based on a hardware signal and RTX5 does view these interrupts as ISRs (All ISR rules apply).

Some final general notes on the code:

- The RP2040 CPU is overclocked to 250MHz to support the requested baudrate of the GNSS module
- There is one interrupt callback function which handles all GPIO triggers
- Anytime the LCD is written to a mutex is used to protect the LCD from being corrupted
- Anytime a thread is resumed, a helped function translates the osStatus to a readable string
- The UART_RX function (that reads in the UART buffer) has a static character rotation to adjust for the setup time of the GNSS module
    - This rotation formats the output in two separate and distinct lines
- All data is pulled from the raw UART string from a static context, this allows a faster data processing

Results

In the end, our clock worked as intended. The clock's refresh rate was just about ~450ms with the idle thread putting the device in low power mode when active.

The device alerts the user when it is starting up both on the LCD and over UART.

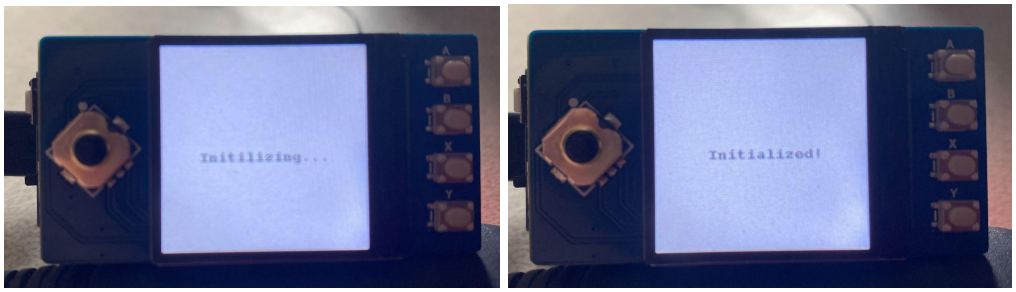

Figure 5 and 6: The device when starting up

Debug (printf) Viewer

```
Kernel Information: RTX V5.5.3
Kernel Version    : 50050003
Kernel API Version: 20010003
Total RAM Usage: 95.7%
Kernel Tick Freq: 1000Hz

Initializing GNSS module, please wait...
Initilized UART at: 9600 baud

Sent: $PMTK251,115200*1F

Restarting UART interface...
Initilized UART at: 33806 baud

Sent: $PMTK220,400*2A

Sent: $PMTK314,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0*28

Sent: $PMTK255,1*2D
```

Figure 7: The printf window on startup

At one point, the team tried to put the RTX5 information on a seperate screen (along with a thread and signal) but the code became too large to flash to the RAM and still debug. This was a block that we could not overcome in the final week of the project so we opted to print the RTX5 info to the console instead. The device screens have the following designs.

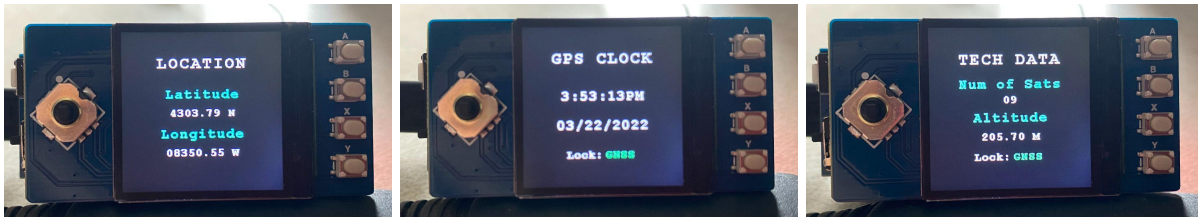

Figure 8,9 and 10: The device in location, clock, and technical data mode (respectively)

When switching threads the device prints information to the printf window to alert the user of the switch.

```
Resumed LOCATION_SCREEN_THREAD thread. Result: osOk
Resumed TECHDATA_SCREEN_THREAD thread. Result: osOk
Resumed CLOCK_SCREEN_THREAD thread. Result: osOk
```

Figure 11: The output from the device when switching between threads

You can also enable the NMEA formatted messages to the console by pressing button Y on the screen.

```
$GNZDA,195710.291,22,03,2022,,*48
$GNGGA,195710.291,4303.7843,N,08350.55W,2,9,1.03,217.6,M,-34.5,M,,*75
-----------------------
$GNZDA,195710.691,22,03,2022,,*4C
$GNGGA,195710.691,4303.7843,N,08350.55W,2,9,1.03,217.6,M,-34.5,M,,*77
-----------------------
$GNZDA,195711.091,22,03,2022,,*4B
$GNGGA,195711.091,4303.7843,N,08350.55W,2,9,1.03,217.6,M,-34.5,M,,*73
-----------------------
$GNZDA,195711.491,22,03,2022,,*4F
$GNGGA,195711.491,4303.7843,N,08350.55W,2,9,1.03,217.6,M,-34.5,M,,*74
-----------------------
$GNZDA,195711.891,22,03,2022,,*43
$GNGGA,195711.891,4303.7843,N,08350.55W,2,9,1.03,217.6,M,-34.5,M,,*70
-----------------------
```

Figure 12: NMEA formatted messages from the device

We also, through the CMSIS-DAP debugger, got a sizable amount of information from the RTX RTOS panel within MDK.

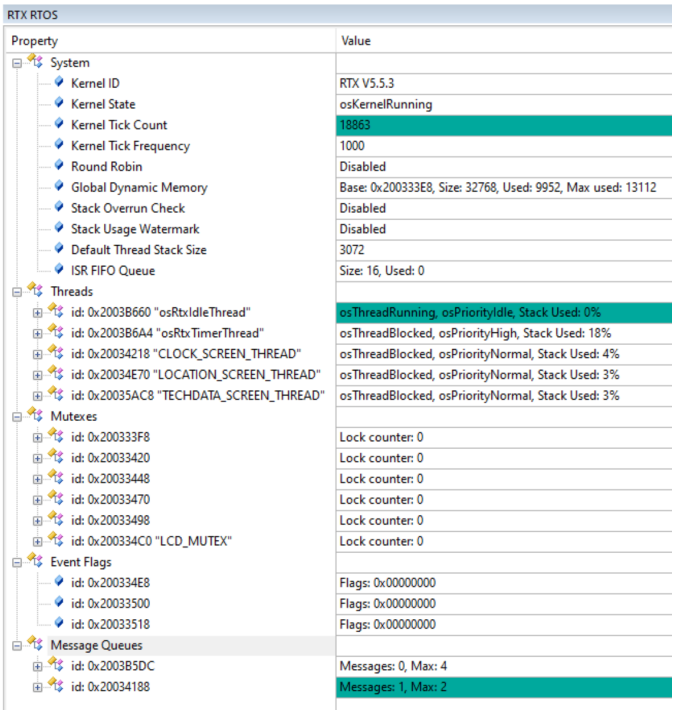

Figure 13: The RTX RTOS panel within MDK

Conclusion

In conclusion, throughout this project we exercised our knowledge of real time operating systems, specifically with ARM's RTX. We used threads, mutexes, message queues, signals, and priority scheduling within our project. In analysis of the project, we noticed that we used 96.7% of our total RAM space.

Program Size:

Code=14048 bytes **RO-data**=15284 bytes **RW-data**=288 bytes **ZI-data**=189060 bytes

Total RAM Usage ***(All data is stored in RAM due to the debugger)*****:**

$$= \frac{Total\ Program\ Size}{Total\ RAM\ Space} = \frac{245{,}112\ bytes}{256{,}000\ bytes} = .9574\ or\ 95.7\%$$

Equation 1: RAM usage calculations

It is evident that our biggest blocker was the total RAM available. We worked around this by downscaling the resolution of the LCD at the cost of actual screen space to put information. This RAM issue could be solved by using a different main board with a larger amount of RAM or flashing the code directly (Although, this removes the ability to live debug). The team feels that this project could indeed be adapted to a product that could be sold in the future. Some changes would have to be made such as switching the main board for one with more RAM and adjusting the code to use the full resolution. More stringent and extensive testing would have to occur to ensure that the device does not crash or corrupt at any point.

References

Lawrence, Peter. “majbthrd/pico-debug: virtual debug pod for RP2040 "Raspberry Pi Pico" with no added hardware.” *GitHub*, 4 February 2021, https://github.com/majbthrd/pico-debug. Accessed 22 March 2022.

“Pico specifications.” *RaspberryPi.com*, January 2021, https://www.raspberrypi.com/products/raspberry-pi-pico/specifications/. Accessed 22 March 2022.

Quectel. “L76 Series GNSS Protocol Specification.” *Quectel.com*, 21 June 2017, https://www.waveshare.com/w/upload/5/56/Quectel_L76_Series_GNSS_Protocol_Specification_V3.3.pdf. Accessed 22 March 2022.

Wang, Gabriel. “GorgonMeducer/Pico_Template: An MDK template for Raspberry Pi Pico.” *GitHub*, 13 July 2021, https://github.com/GorgonMeducer/Pico_Template. Accessed 22 March 2022.

Waveshare. “Pico-GPS-L76B.” *Waveshare Wiki*, 8 December 2021, https://www.waveshare.com/wiki/Pico-GPS-L76B. Accessed 22 March 2022.

Waveshare. “Pico LCD 1.3.” *Waveshare Wiki*, 17 June 2021, https://www.waveshare.com/wiki/Pico-LCD-1.3. Accessed 22 March 2022.