12-2020

Digital Power Supply Controller

Michael Gendreau
University of Central Florida, mike.gendreau@knights.ucf.edu

EEE 4775 Real-Time Systems

Digital Power Supply Controller

Final Project – System Implementation

Michael Gendreau
12-12-2020

Introduction

This paper details the implementation of a real-time system in the context of a digital power supply controller. Attention is primarily given to the software development of the project given its relevance to the concept of a real-time system. Having said that, the need for hard deadline tasks in this system is not very evident. As discussed later, the consequences of a missed deadline for most tasks in the system should be no more than an unreliable test sequence for the device under test (DUT). However, since many concepts related to embedded real-time systems can be applied to the software development in this project, I present the various system tasks with a stricter-than-life priority and deadline type assignment, with corresponding weight in the subsequently utilized scheduler.

A bench power supply is a piece of electronics test equipment used to accurately and precisely supply a device under test with DC voltage and current. These instruments are vital for any electronics lab – in my case, I needed the equipment for my senior design project since the on-campus labs were closed due to social distancing requirements. Our project has power supply requirements for 12VDC at up to 2 amps, and in order to accurately gauge its performance at that supply voltage, a high level of voltage and current resolution (1 mV, 1mA) is necessary on the display readout from the instrument. This allows constant monitoring of system parameters like power consumption and current peaks without allocating another instrument, like a digital multimeter, to the task.

In the search for a suitable instrument, reality set in that a bench power supply with the desired capabilities was not nearly affordable. Newer models of power supplies with digital control and displays with 1 mA, 1 mV resolution start, on the low end, around $500. The solution was to purchase a much cheaper out-of-date model with analog I/O and implement an upgrade using an embedded MCU and custom PCB for digital manipulation of the supply voltage and current. This effort involved reverse-engineering of the analog control and display front-end of the instrument, achieved by operating the instrument with the covers removed and probing various signal lines to determine what their expected inputs were.

Figure 1 GPC 3020 - Analog Power Supply as Purchased

Digital Conversion - Hardware

In order to set voltage and current outputs on each of the two channels of the power supply, an analog voltage proportional to the output voltage or current is applied to the internal instrument amplifiers used to drive the linear voltage regulators. With the analog front-end, these voltages are set by a pair of potentiometers on each channel. To make this function digital, the potentiometers were replaced with encoders, and the analog input voltages were connected to digital-to-analog converters (DACs). To replace the analog gauges with precise digital readouts, the gauges were replaced with LCD displays, and their input signals were fed to analog-to-digital converters (ADCs). This sets the stage for the overall design of the PCBs – analog signal conditioning, a pair of ADCs, digital I/O, a pair of DACs, an LCD display, and a microcontroller to manage it all.

The subsystem design can be seen in Figure 2. The interconnection of the various devices is achieved by an I2C bus – each of the aforementioned devices are slaves on the bus and the microcontroller is the master. Because the I2C bus is speed-limited by the slowest slave on the bus, the inter-device communication on the board tops out at around 200-300kHz clock rate. This upper bound on the communication speed greatly influences the software design since most tasks that get scheduled require full control of the I2C bus, and this resource usage can't be preempted during use without first terminating the current transmission.

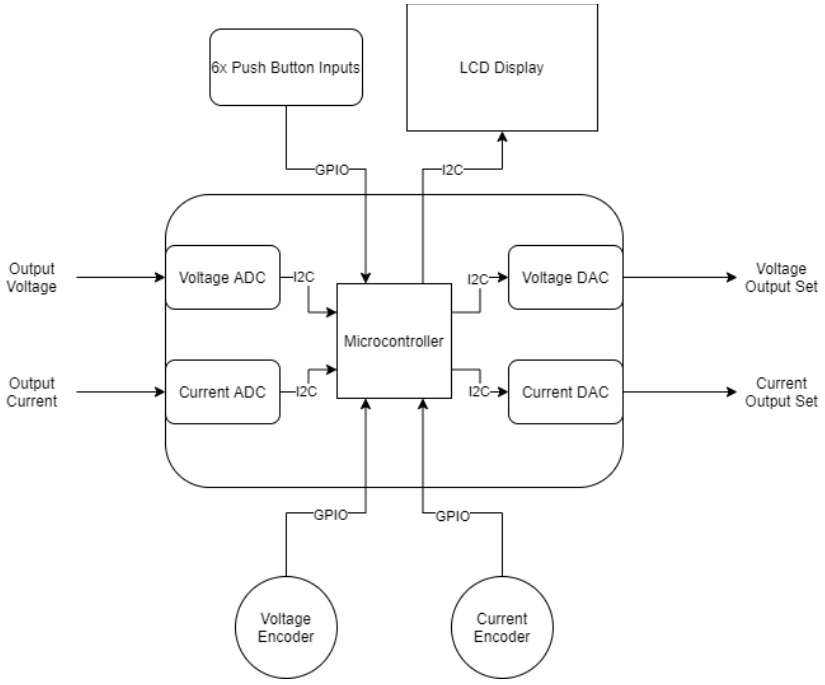

Figure 2 System Interconnect Block Diagram

Digital Conversion – Software

Scheduling Algorithm

The first step in designing the software for this system is defining the type of scheduler necessary for the set of all jobs run by the microcontroller. The following table defines all significant jobs to be run by the microcontroller and describes the periodicity and priority of each job.

Job Name	Job Description	Periodicity	Priority	Deadline
Update_Outputs	Update V + I DACs on I2C Bus	Periodic	High	Hard
Read_Inputs	Read V + I ADCs on I2C Bus	Periodic	Medium	Hard
Update_Display	Refresh LCD Display Fields	Periodic	Low	Soft
Update_EEPROM	Update nonvolatile memory	Periodic	Low	Soft
But_Push	Button Pushed on Front Panel	Aperiodic	High	Hard
Encoder_Turn	Encoder Turned by 1 Increment	Aperiodic	Medium	Hard

With these jobs defined, it's evident that for this low-level embedded system, a frame-based clock-driven scheduler is best suited to schedule the many periodic tasks needed for operation. A frame-based scheduler has some strong benefits in this application: low-complexity, guaranteed timing performance and schedulability for periodic jobs, and flexible slack length dependent on frame sizing and task durations. Frame-based schedulers, for these reasons and more, are frequently found in small-scale embedded systems that don't frequently need changes made to their periodic schedules. That leads the discussion to the downsides of a frame-based scheduler: low flexibility, poor integration of hard and soft deadlines, fixed job release times, and the need for accurate, constant job execution cost information in order for the precomputed frame to satisfy schedulability requirements. Any change in the job parameters will require a new precomputed frame to be written and pre-established at runtime.

Other schedulers are possible for this system. Most notably, priority-driven schedulers would be a strong candidate for handling the tasks outlined above. Priority-driven schedulers will be capable of online scheduling with no pre-computed fixed frames given a unity bounded utilization, and the differences in priority for each soft and hard deadline for both periodic and aperiodic jobs will be discriminated during runtime. An example of such a priority-driven scheduler that is freely available is FreeRTOS. It uses a mix of round-robin scheduling and preemptive deadline monotonic static priority scheduling, with configurable options for defining how those schedulers are implemented exactly. With an extremely capable RTOS freely available, why then wouldn't it be the solution for this project's scheduler? The biggest limitation in this project is the nonvolatile program flash memory on the microcontroller of choice, the MSP430G2553. Unfortunately, 16 kilobytes is all there is to work with, and FreeRTOS would likely occupy half of that space – space that cannot be afforded given the size of the rest of the program.

Timing Analysis

Knowing the scheduler of choice, the next step in designing the software for this system is determining the parameters of the jobs to be scheduled. Most important among these parameters is the execution cost, or total active duration per release, of each job. For this simple system implementation where context switching is infrequent and most jobs utilize the I2C bus, simply measuring the duration of clock activity on the I2C is a useful approximation of each job execution cost. I2C transmissions occur at frequencies of no more than 300 kHz as previously established. With a system clock rate of 16 MHz, an order of magnitude above the I2C clock, I2C transmission duration occupies an overwhelming majority of the execution cost of every task that gets scheduled. Consequently, a good estimate of execution cost can be obtained with an oscilloscope probing the I2C bus. The following figure shows an example of this measurement.

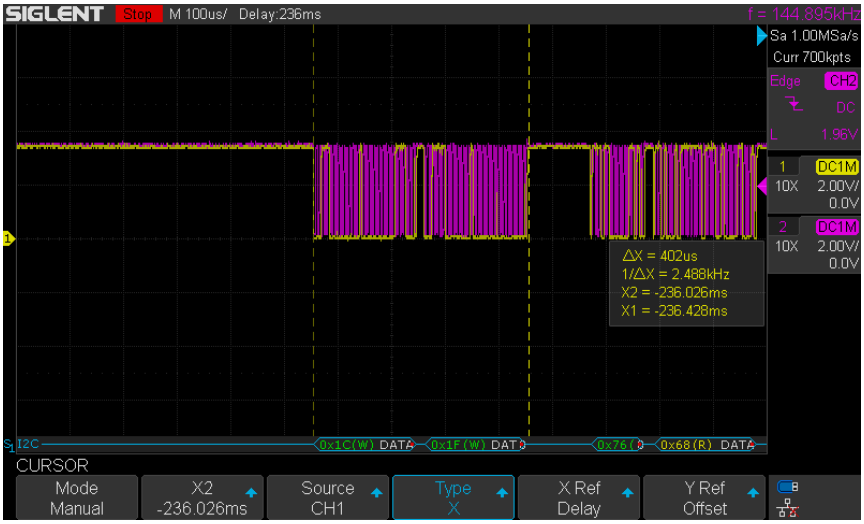

Figure 3 Timing Analysis from Oscilloscope, Update_Outputs Task

The only jobs that don't occupy the I2C bus are "But_Push" and "Encoder_Turn." These are handled by GPIO interrupts with simple variable assignments present in their interrupt service routines dependent on prior conditional statements. Consequently, at 16 MHz, the execution costs of these aperiodic jobs are well below the execution costs of the I2C bus-occupying jobs. Therefore, a conservative estimate of the execution cost of these jobs can be described as, worst case, 5 µs. At 16 MHz, this represents a total of 80 clock cycles. The following table lists the results for this method of timing analysis and our estimates of the aperiodic jobs as well as the relative deadlines of each job. Relative deadlines are determined by the user's requirements for responsivity and other self-imposed specifications for changing the voltage applied to the DUT before damage can occur in testing scenarios. This is especially critical in initiating power-on sequences where applying power on an input before another component of the DUT is powered on could result in damage.

Job Name	**Execution Cost (E_i)**	**Relative Deadline (D_i)**	**Period (T_i)**
Update_Outputs	402 µs	2 ms	300 ms
Read_Inputs	824 µs	2 ms	300 ms
Update_Display	279 ms	285 ms	300 ms
Update_EEPROM	288 µs	2 ms	300 ms
But_Push	5 µs	281 ms	--
Encoder_Turn	5 µs	281 ms	--

From the table, it's obvious the majority of each frame will be spent serving the "Update_Display" tasks. The large execution time for this job can be attributed to the large amount of data needed to be written per job release. The display is a 160x100 pixel LCD with 16-bit gray scale settings, requiring 4-bits of data per pixel. Even when only smaller segments of the display are being updated, the resulting communication is still much longer than the other jobs that require only a couple bytes per transmission.

Frame Definition

The scheduler frame is the smallest unit of scheduled tasks that repeats indefinitely while the system is running. To define how the periodic tasks above are scheduled, we have to define the bounds of the frame size. The lower bound of our frame size is given as $f \geq \max(E_i)\ for\ i = 1,2, \dots n$. This means the frame size must be at least 279 ms long. The upper bound for this system can be determined by $2f - \gcd(T_i, f) \geq D_i\ for\ i = 1,2, \dots n$. With a frame size of 300 ms, we satisfy both lower and upper bounds. It must be noted that, despite the large size of the "Update_Display" job, it and the other I2C-using jobs cannot be divided into smaller "sub-jobs" without incurring substantially larger execution costs. This is due to the data framing required for the I2C protocol – starting and stopping transmissions adds at least a byte's worth of clock cycles and overhead due to the addressing scheme of the bus. Furthermore, subdividing the display refresh job means only updating part of the display at a time, resulting in a jittery screen and loss of immediately available accurate data.

The sensitivity of the I2C bus to interruptions or division of transmissions also means that preemption of the periodic tasks is not allowed. Timing on the I2C bus would be thrown off if the CPU switched context mid-transmission to handle a button push or encoder turning. With no preemption possible for aperiodic tasks, the relative deadlines for all aperiodic tasks must be equal to the worst-case execution cost of all periodic tasks which would execute just prior to the aperiodic task release. Having hard-deadlines, this guarantees that aperiodic tasks that have major influence on system operation never miss their deadlines. The downside is that the response time for these events has a chance for nearly 300 ms of delay – quite significant at this scale. The following figure shows the singular frame – of note, in order to maximize display refresh rate, only one frame schedule exists. Therefore the hyperperiod of all tasks is essentially the same as their period.

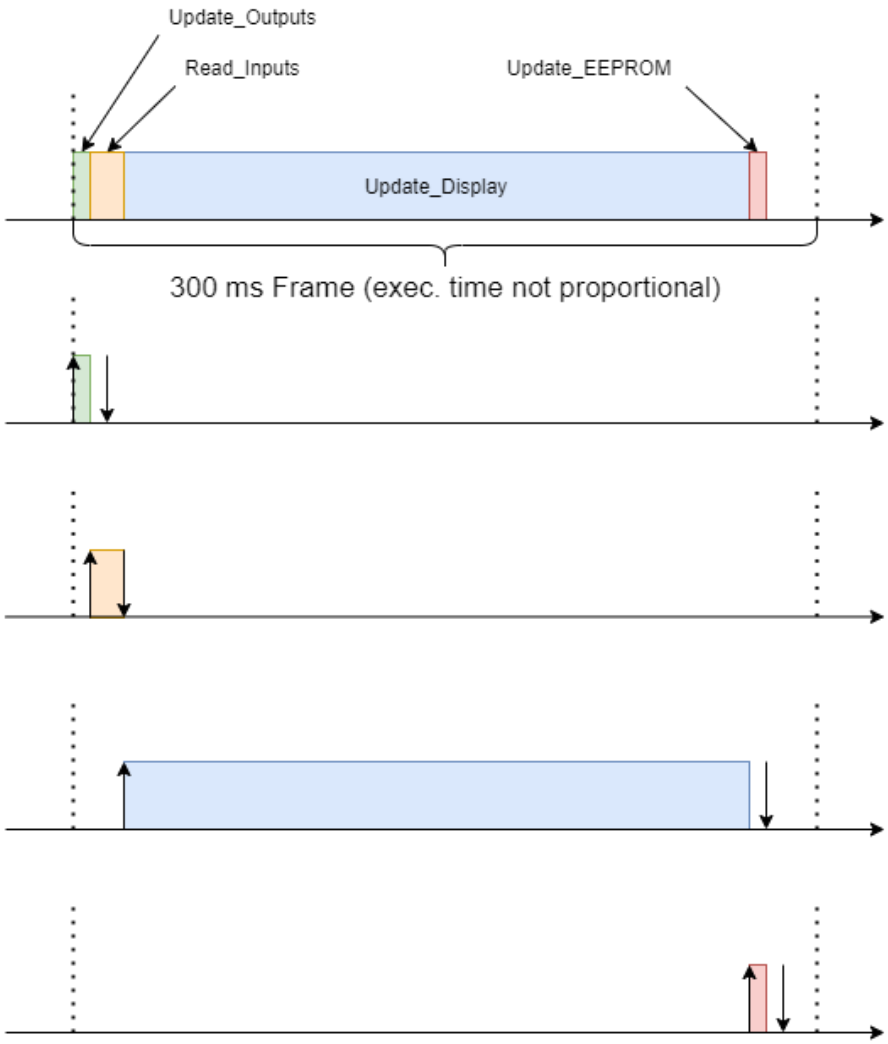

Figure 4 Scheduled Frame Definition

Conclusion

This design, while a good learning experience, demonstrates a lot of shortcomings that could be improved at both the hardware and software level. It is obvious that the use of non-preemptable, slow serial communication protocols is detrimental for responsivity of a system and the ability of an actual real-time system to meet hard deadline specifications. Furthermore, there are opportunities to decrease the total response time of the system for aperiodic tasks. Each frame has about 25 ms of slack, and this is concentrated in one portion after all other periodic tasks are served. If the slack were better distributed throughout each frame, the schedule could better accommodate aperiodic task arrival. Even better, if the scheduler facilitated slack stealing calculation at the frame boundaries, the schedule would give the best response time for the aperiodic tasks possible in a frame-based clock-driven schedule. For simplicity sake, slack stealing was forgone in this project.

Note: See appendix for code and additional figures.

Appendix

Scheduler

The following code segment serves as the frame scheduler; a state machine that advances every iteration of this permanent while-loop will move from each operation dependent on timer interrupts that trigger flag variables for each task to move forward. Additional checks are in place to prevent preemption of tasks in the event that the I2C bus hangs longer than expected. Not shown: HW timer interrupts that enable flags.

```
    while (1) {
        // adc readout flag
        if (timerGo2 == 1) {
            if ((byteNum == 4) && (measG != 'v')) {
                TA0CCR2 = TA0R + 164;
                TA0CCTL2 &= ~CCIFG;
                TA0CCTL2 |= (CCIE);
                adc_busy = 1;
                adc_readout('v');
            }
            if ((byteNum == 4) && (measG == 'v')) {
                adc_busy = 1;
                adc_readout('a');
                timerGo2 = 0;
            }
        }
        i2c_FUBAR = 0;

        // display update flag
        if ((timerGo == 1) && (timerGo2 != 1) && (adc_busy != 1)) {
            if (p1Active && presetFlag) {
                eepromRead(0x08);
                while (presetFlag) {}
                byteNum = 4;
            }
            else if (p2Active && presetFlag) {
                eepromRead(0x10);
                while (presetFlag) {}
                byteNum = 4;
            }

            i2c_lcd_update();
            dacOutput();

            if (updatePROM == 1) {
                if (p1Active && (presetFlag == 0))
                    eepromUpdate(0x08);
                else if (p2Active && (presetFlag == 0))
                    eepromUpdate(0x10);
                else
                    eepromUpdate(0x00);
                updatePROM = 0;
            }

            // encoder debounce
            if ((P1IE & CURRENCODE_A != 0) && (aDir != 0))
                aDir = 0;
            if ((P2IE & VOLTENCODE_A != 0) && (vDir != 0))
                vDir = 0;

            timerGo = 0;
            timerGo2 = 1;
        }

        // calculate final received 16bit value, current voltage power
        if (byteNum == 3) {
            if (measG == 'v') {
```

```
                voltageActual = lowerRead | (upperRead << 8);
                if (voltageActual > 0x7FFF)
                    voltageActual = 0;
                else {
                    if (voltageActual > 1) {
                        unsigned long mathOp = voltageActual;
                        if (slaveMasterConfig) {
                            mathOp = (10021 * (mathOp-8) / 10000 - 6); // cal offset (MASTER)
                        }
                        else {
                            mathOp = (10013 * mathOp / 10000 + 8); // cal offset (SLAVE)
                        }
                        if (mathOp > 0x7FFF)
                            mathOp = 0;
                        voltageActual = (mathOp & 0xFFFF);
                    }
                    else
                        voltageActual = 0;
                }
            }
            else
            {
                currentActual = lowerRead | (upperRead << 8);
                if (currentActual > 0x7FFF)
                    currentActual = 0;
                else {
                    unsigned long mathOp = currentActual;
                    if (slaveMasterConfig) {
                        mathOp = (mathOp * 2057 / 10000 + 4); // cal offset (MASTER)
                        if (mathOp < 600)
                            mathOp = mathOp - (4 - (mathOp * 40 / 6000));
                    }
                    else {
                        mathOp = (mathOp * 2128 / 10000 + 2); // cal offset (SLAVE)
                        if (mathOp < 500)
                            mathOp = mathOp - (3 - (mathOp * 30 / 5000));
                    }
                    if (mathOp> 0x7FFF)
                        mathOp = 0;
                    currentActual = (mathOp & 0xFFFF);
                }

                power = (voltageActual * currentActual / 1000); // calculate power from V and I

                while ((UCB0CTL1 & UCTXSTP) != 0) {}
                TA0CCTL2 &= ~(CCIE | CCIFG);
            }
            adc_busy = 0;
            timerGo = 1;
            byteNum = 4;
        }
    }

        return 0;
}
```

Timing Analysis Setup

I2C Bus at Runtime, Measuring Update_Display Task