REMOTE REDUNDANCY: ARTIFICIAL RELIABILITY IN A REMOTE CAMERA DEVICE

Joshua Pidgeon

Bachelor of Engineering
Mechatronics Engineering

MACQUARIE
University
SYDNEY·AUSTRALIA

Department of Engineering
Macquarie University

November 16, 2017

Supervisor: Dr. Mohsen Asadnia

ACKNOWLEDGMENTS

I would like to acknowledge Brett Richardson of Outback Tech for his assistance in identifying a suitable application for this project.

I would like to thank Dr. Mohsen Asadnia for his supervision and academic advice for this project.

I would like to acknowledge Dr. David Inglis, Dr. Rex Di Bona, and Andrew Proschogo for their guidance and advice for this project.

I would also like to acknowledge the support of my wife, Amanda. Without her support and encouragement, this project would not have reached its conclusion.

I also thank my Creator God for his design and equipping.

STATEMENT OF CANDIDATE

I, Joshua Pidgeon, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not previously been submitted for qualification or assessment at any academic institution.

Student's Name: Joshua Pidgeon

Student's Signature:

Date: November 16, 2017

ABSTRACT

A self-supervised redundancy system has been developed for open-source microcontrollers, to tolerate hardware faults, and to recover from software faults. This device is intended to enable reliable deployment in rural and difficult access situations. An exclusion lock is used to prevent additional microcontrollers from simultaneously controlling the system. Watchdog timers provide resetting capability, to enable error recovery. The implementation of this system, into a camera monitoring device, was not completed, because of clashes in the initialisation of the modules.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter deals with the introduction of the research topic in Section 1.1. The overview of the project is discussed in Section 1.2.

1.1 Reliability in Remote, Rural, Difficult Access Locations

For as long as electronic device have existed there have been electronic devices that have stopped working.

Some of these faults have been attributable to environmental factors such as corrosion, user interference, or simply wear and tear.

Some devices stopped working because of manufacturing faults. These faults may be due to oversights during quality control. They may have been from low manufacturing standards. They may have been from sub-standard materials used for manufacturing.

Some devices stopped working due to faulty designs, such as unforeseen interactions between components. Sometimes the design flaws are in the software. Simple flaws may be recursive loops that become fixed in an infinite loop. Other flaws may be more subtle, such as neglecting to account for a counter overflow.

Other faults can only be attributed to component wear and tear. Whether it's the contact points on a toggle switch or the inner workings of a cellular module fixing onto a cellular tower, all components wear out over time.

When these faults occur, they can often occur without warning, and if there's no backup plan, then time and effort are required to fix the problem. If a connection has been lost, the fault may only take time and effort to resolve. If parts need to be replaced, then direct financial cost may also be incurred. Also, if the device has a high priority for operational uptime, other tasks may be neglected while this device is corrected. Also, if the device is relied upon, the affected system may not be be able to operate until the component is repaired or replaced.

The disruption caused by faults would be far less severe if a device could identify faults, and still continue working at full capacity. This would reduce the need of interrupting other tasks to commit resources to repairs. It would also reduce or perhaps even eliminate any inflicted downtime due to component failure.

An even better solution would for the device to attempt to fix itself. For microcontrollers, this may be a simple as rebooting the microcontroller to re-initialise values, and states.

This capability of coping with faults is called fault-tolerance. In the same way that there is a range of causes of faults, there is a range of levels of fault-tolerance. This range includes simple duplication, through fault identification even to the point of being able to remedy the fault.

A very basic fault-tolerance merely duplicates some or all of the components in a system in parallel. This approach has the effect of continuing to deliver results even if one of the parallel circuits fails. This indifferent parallel system will continue until there is a fault in each of the parallel paths. While some processing components may be reset by an incidental system reboot, the system does not identify whether or not a fault has occurred.

A more dynamic fault-tolerance is similar to the first, employing additional sensors or monitoring lines to be able to track the occurrence of faults. This approach has the greater benefit of being configurable to alert the user to faults. This in turn allows flexibility in scheduling repairs that otherwise may have been urgent unscheduled repairs. If completed before any other parallel paths have encountered faults, continuous uptime may be achievable.

This project seeks to take this level of fault-tolerance another step further. If faults are identified in the operation of a microcontroller, the microcontroller will be reset, allowing overflows and infinite loops to be overcome. While this method cannot recover from broken connections, it still has the benefit of being able to identify any identified faults to the user.

It is believed that this capability will have particular usefulness for low-production devices. Relevant devices include monitoring devices used for academic research as well as prototype monitoring devices.

Fault-tolerance needs to be able to identify at least in some basic way, that a fault has occurred.

1.2 Project Overview

This section deals with the summarisation of the entire project. The aim of the overall project is to develop a camera monitoring device to capture images when prompted from an external trigger. This device will employ redundant 8-bit microcontrollers to maximise fault-tolerance in rural and remote applications. The given requirements for the device as specified by the industry partner, Outback Tech are as follows:

1. The device must capture an image within 500 milliseconds of external triggering.
2. The triggering should accommodate a zero voltage input.

3. The device must send captured images to a specified email address via an included 3G cellular connection.

4. The included cellular connection must accommodate a detachable external antenna for the addition of range boosting devices.

The zero voltage input is taken to mean merely that the triggering device must only have the effect of closing a passive switch. The system is not expected to accommodate an input voltage on the triggering input terminals.

Research was conducted in the in Session 1, 2017 at Macquarie University into the most suitable redundancy arrangements. A combination of voting and parallel configurations was concluded as the most resilient system.

Further research and development is being conducted at Macquarie University to design the required control algorithms, the necessary electronic circuits, and the subsequent code to achieve this goal.

Chapter 2

Background

2.1 Introduction

A device's reliability is a measurement of how likely the device is to fail within a certain timeframe. Some commercial components are tested extensively, often at great cost to the producer to be able to specify a given reliability. Item's produced without rigorous attention to detail may be more likely to fail, that is they may have a higher probability of failure within a given period.

Redundancy configurations can be seen in safety devices, for example, car braking systems [1]. This inclusion of redundancy allows a system to achieve its purpose even if one of the channels does not actuate or connect. In this way, a system can tolerate faults in the system.

Incorporating redundancy into design can allow a system to continue to work even though it encounters faults. Since no system is perfect, and wears out over time, every system can expect to encounter faults.

Rather than focusing great energy and cost in trying to reduce the likelihood of faults to occur in a system, a better approach may be to design a system to accomodate faults and even to be able to recover from faults. While engineers have been aiming to design fault-tolerance into microcontrollers for many years, fault-tolerance in a single component does not allow for that component to completely fail [2].

This literature review aims to identify relevent factors to consider in the design of a fault-tolerant system using embedded microcontroller redundancy.

2.2 Background

Microcontrollers have been used for many years for hobbyist projects, academic experimentation, and various low-production devices. The reliability of commonly accessible microcontrollers is sometimes not high enough for prolonged deployment. This is particularly a problem for extended experimentation applications as well as for difficult access situations.

The sometimes poor reliability can be attributed to design flaws, programming inadequacies, and sometimes poor physical implementation.

This project seeks to build a device that will be tolerant to these faults, through the use of existing industrial practices. Factory safety circuits, elevator emergency brakes, and aeroplane control systems all use redundancy to improve reliability.

The main concept of redundancy is to employ alternative communication or control channels so that if one fails, the connection can still be completed.

2.3 Duplication

Some systems have been developed that run control systems in parallel [1, 3]. These systems incorporate communication between the parallel systems, but they also duplicate much of the supporting circuitry. Inputs are duplicated close to the sensors, even to the point of complete duplication of sensors. Outputs are duplicated even to the point of separate communication buses contiuning all the way to the acuators. This approach allows a system to operate at full functionality even if one component entirely breaks down.

Other systems have a secondary microcontroller functioning as a checker for the primary controller [4]. Rather than taking over as the primary controller, the secondary controller confirms or refutes the logic of the primary controller. This can be used to guide the primary controller or even to disable the primary controller.

Some systems implement duplication of controllers, and additionally have an overarching master microcontroller [5]. This approach allows the master to identify a slave controller that is functioning as required and assign the processing responsibility to that slave controller.

Some systems incorporate a combination of these two systems where multiple systems run in parallel with identification and assignment of the master controller being handled concurrently by the slave controllers [5]. The master controller is the one which handles the data processing rather than simply identifying which slave handles the processing. If a master unit is deemed to not be operational, the identity of master passes to the next available controller. In this way, if the main controller fails, another takes its place.

A hybrid arrangement of parallel duplication and joint evaluation can harness the benefits of two of these arrangements [1]. This system has two paris of parallel controllers comparing results to determine the primary set. It has the benefit of identifying errors as in the full duplication checking. It can also reassign which set of parallel controllers will have the primary influence over the output control.

2.4 Master Identification and Assignment

As previously mentioned, master assignment may take the form of a higher level microcontroller choosing the primary controller from between multiple devices [2]. After an

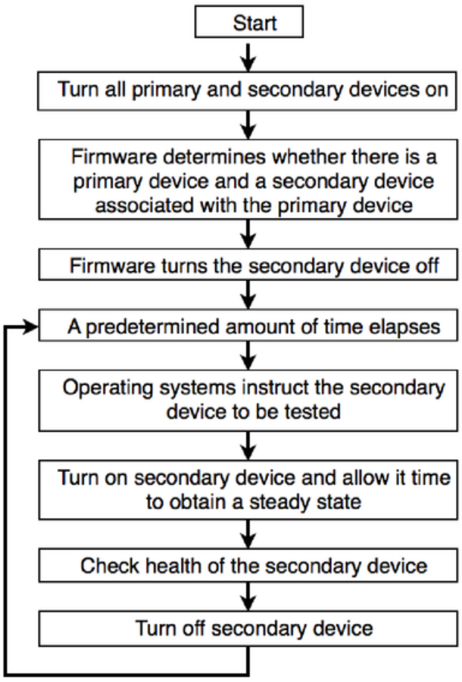


Figure 2.1: Flow chart for evaluating secondary device. [2]

evaluation process, as shown in Fig. 2.1, an operational device is selected as the primary and activated.

Another form of master assignment is carried out between peer controllers [6]. This particular application only duplicates the controller for a communication bus, however, through an evaluation flow chart more complex than that shown in Fig. 2.1, the choice of primary controller is decided between peers. This is achieved by defaulting to one controller, running some initial tests, and switching to the secondary controller if the primary fails.

2.5 Voting

For peer master assignment among equally ranked controllers, a voting system can be implemented. This feeds input from all available controllers into digital logic which then assigns the identity of primary controller to one of the peers [1, 5, 7]. For systems with greater than two peers, this voting can be used more dynamically. Although a two controllers may give an erroneous output, it is less likely than one making a erroneous ouput.

While these dual errors may be less likely, they are not impossible, and can prove difficult to diagnose.

2.6 Fault-Detection

When one ocntroller is evaluating another controller in a two controller system, the identification of a fault may reflect a fault in the tester or the testee. If the first controller has a problem and cannot read appropriate signals from the second controller, it may incorrectly conclude that the second controller is not operating properly. A smoothing predictive method has been developed for overcoming such faults of misdiagnosis [1].

This smoothing predictive redundancy method includes previous evaluations in deciding whether or not a device is functioning as expected. Previous readings are included with more recent readings given greater weight than earlier readings. This has the effect of ironing out irregularities if there was an occasional inaccurate reading. This is particularly applicable for processing of analog signals.

2.7 Restartability Analysis

When a controller has been determined to have failed, restarting it may help to set it running correctly again [5]. This should be part of further evaluation of the controller's operation. Algorithms should be implemented to make allowance for a controller to re-enter the system upon restarting.

2.8 Conclusion

The design of a fault-tolerant system needs to consider the level of fault-detection required. A trade-off between the number of embedded microcontrollers and the depth and accuracy of fault-diagnosis will determine whether a system merely copes with failed components, or the system can recover from malfunctioning components.

After the issue of identifying faults, the next biggest hurdle would be the selection and assignment of the primary controller to take on primary responsibility within the system.

The number and arrangement of included microcontrollers will greatly shape the effectiveness of voting. The optimum arrangement is a combination of static and dynamic redundancy.

Chapter 3

Supervised Parallel Redundancy

3.1 Introduction

This chapter details the development of a simple supervised system of parallel redundancy. This system uses three analogue inputs to generate requirements for system outputs of a servo motor and two light emitting diodes (LED). The outputs rely on in the inputs in varying forms to demonstrate the capability of the system.

The chapter is arranged in the following order. In section 3.2, the control flow is discussed. Section 3.3 describes the circuit used for this system. Section 3.4 deals with the Arduino Code development for each of the MCUs. Section 3.5 describes the process of experimentally testing the system. The results are presented in section 3.6, which are then discussed in section 3.7.

3.2 Control Flow

This system consists of a dedicated supervising MCU monitoring the system output along with two parallel supervised MCUs. The supervisor selects one of the supervised MCUs to perform the system function, then monitors the performance. If the outputs are produced as expected, the selected MCU maintains the primary role. If the outputs are not produced as expected, then the primary priority is given to the other supervised MCU. This decision-making process of the supervisor is shown as a flowchart in fig. 3.1.

Both supervised MCUs perform the same function. This means that both MCUs will run from the same set of code. Each will assess the inputs, calculate the outputs, and produce their outputs as expected.

The supervisor MCU also assesses the inputs in the same way as the supervised MCUs. However, the supervisor does not produce the outputs for the overall system output. Instead the supervisor also assesses the outputs and compares these outputs with the calculated outputs based on the inputs. In this way, the supervised MCUs have their performance monitored.

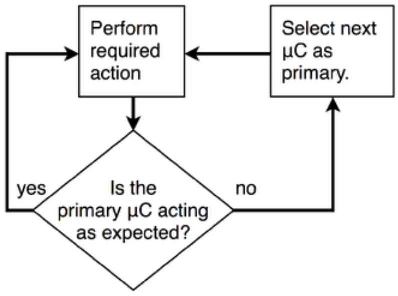


Figure 3.1: Operational flow chart of supervised system.

3.3 Circuit Design

The design of this supervised redundant circuit can be seen in Appendix B.2. One supervisor MCU enables and disables the outputs of two parallel MCUs that perform the core function. All three MCUs read the input values from a 10 kΩ potentiometer (pot) and two normally open (NO) pushbuttons (PB). The processed outputs of the parallel MCUs control a pot and two LEDs. The supervisor MCU controls these outputs through the use of optocouplers. All three MCUs are Atmega328P DIP ICs.

3.3.1 Enable Signals

The supervisor outputs two enable signals which are the inverse of each other, to enable or disable the outputs of the supervised MCUs. This control is achieved by the activation and deactivation of the LED light source within each of the optocouplers on each output of each parallel MCU. The output lines of the parallel MCU are each connected to the photo diode pins of their respective optocoupler. Pull-down resistors of 25 Ω are used for each of the LED inputs of the optocouplers. All optocouplers are 4N25 DIP ICs from Fairchild Semiconductors.

The inputs and outputs of each of the parallel MCUs are monitored by the supervisor and the appropriate MCU is selected based on the measured operation. If MCU A is operating as expected, it remains as the designated MCU. If an error is detected in circuit A, the enable line for its output optocouplers will be taken low, and instead the enable line for MCU B will be taken high, enabling the outputs of MCU B. An LED with its accompanying resistor has been added to each of the enable outputs for debugging purposes during experimentation.

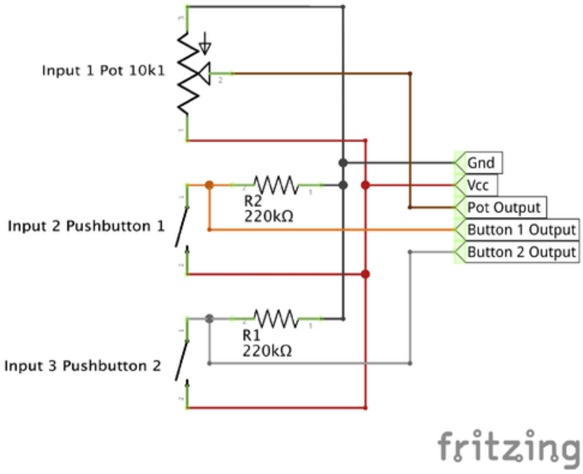


Figure 3.2: Experimental input circuit diagram.

3.3.2 System Inputs

The inputs consist of a pot and two pushbuttons.The 10 kΩ pot is connected between ground to 5 V, with the wiper connected MCUs as an input. Each NO PB has a 220 kΩ resistor between the button and ground to reduce the power consumed by the experimental system. Each button output is connected to the resistor side of the button, and is pulled high when the button is closed.

3.3.3 Experimental Inputs

For experimental purposes, the input components of the pot and PBs can be moved to a separate board. The diagram for this peripheral input board is shown in fig. 3.2. If this peripheral board is connected as specified in the circuit, the button outputs behave as NO PBs. If the Gnd and Vcc supplies are swapped, the button outputs would behave as normally closed (NC) PBs.

3.3.4 Clock Circuit

Although it has been omitted from the circuit diagram for reasons of available space, each MCU has an independent 16 MHz crystal oscillator. Each crystal has two accompanying 20 pF ceramic capacitors. The crystals provide external clocking for the MCUs. As each MCU has an independent crystal, the system operate on asynchronous timing. An example of the oscillator circuit for an Atmega328 is shown in fig. 3.3.

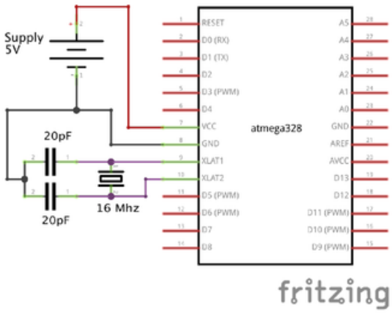


Figure 3.3: 16 MHz oscillator clock circuit for Atmega328.

3.3.5 In-System Programming

A method of updating the code on an Atmega328 while still fitted to a circuit enables rapid deployment of code. The circuit shown in fig. 3.4 shows a simple method of in-system programming (ISP) using an Arduino Uno. The Atmega328 has its own clock circuit on the breadboard. The power for the breadboard system may be supplied from the Arduino Uno or from an independent power supply. The Arduino Uno must have its included MCU IC removed.

Using this method requires a maximum of only five wires to upload iterations of Arduino code to the MCU. This process requires the two lines used for the serial connection, transmit (Tx) and receive (Rx). The reset line is required for initialising the upload sequence. The ground connection is required to complete each of these signal circuits. Finally, the 5 V output from the Arduino Uno is optional depending on alternative power supplies. If the system has its own power supply, then only four wires are required for ISP.

3.4 Arduino Software Code

3.4.1 Supervised MCU

The sketch for both of the parallel supervised MCUs is shown in Appendix C.2. The sketch begins with the declaration of a servo object and the pin identifiers for the three inputs and three outputs required for performing the main function of the circuit. Input and output working variables are next declared. The setup initialises all declared inputs and outputs. The servo object is also attached to its specified pin in the setup. The loop reads the inputs, calculates the outputs, then writes the output values to their respective output pins.

The output calculations of the circuit demonstrate three different typical controller functions. The first function, translates an analogue pot input into an integer servo

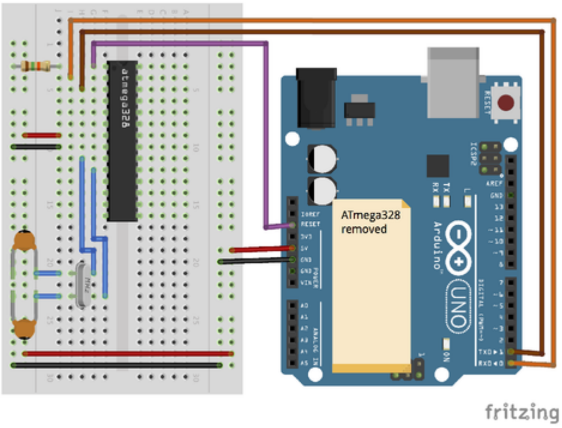


Figure 3.4: Circuit used for communication with MCUs on a breadboard.

position. The second function outputs an exclusive OR (XOR) of two binary inputs. The third function is a single output reflection of a single input.

3.4.2 Supervisor MCU

The sketch for the supervisor MCU is based on the sketch for the supervised MCU with some significant changes as list in Appendix C.3. The supervisor does not produce the system outputs directly. Rather the supervisor compares its calculations of the system outputs with the measured output of the overall system. The binary outputs are either matching or not. The PWM output value may be exact, but may only be close. Line 54 of the code in Appendix C.3 checks if the difference between the calculated output and the measured output is within a given percentage of the possible value, 180. If any of the three comparisons aren't matched, then the boolean flag, allOK, is set to false.

If the system is not all OK, then the enable outputs are toggled. Once these true or false enable values are written to the output pins, this will enable or disable the associated optocouplers respectively. This toggling then facilitates or nullifies the effectiveness of the parallel supervised MCUs. This process of reading, checking, comparing, and enabling is cycled constantly within the loop function of the Arduino code.

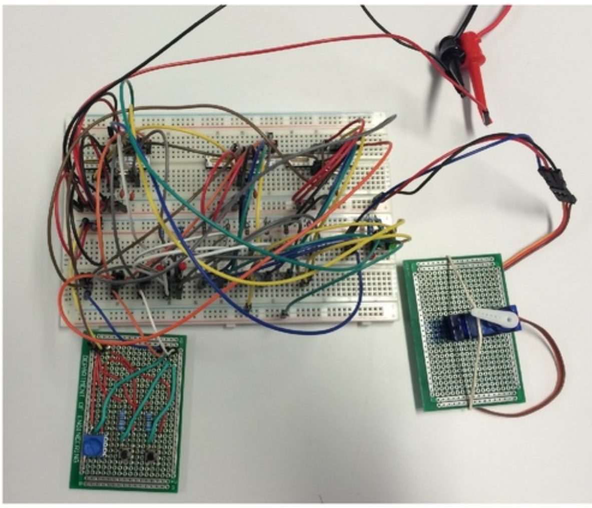


Figure 3.5: Experimental configuration of non-resetting supervised system.

3.5 Experimentation

3.5.1 Experimental Setup

The experimental experimental configuration can be seen in fig. 3.5. The input pot and PBs were soldered onto a prototype printed circuit board (PCB), as seen in fig. 3.6. The power was supplied through a laboratory benchtop power supply, GW Instek PSW-3202. Measurements were taken using two four-channel Agilent oscilloscopes, model DSO-X 2024A. Matlab was used to build the plots from the gathered data.

Atmega328P DIP ICs from Atmel were used for all MCUs during experimentation. 4N25 optocouplers from Fairchild Semiconductors were used for this experimentation. A Tower Pro 9 g micro servo, model SG90, was used for the servo output of the system.

3.5.2 Introduced Errors

To prompt the enable lines to be toggled, errors were simulated in the circuit. The removal of one or two wires from the optocouplers to the outputs caused the final output values to mismatch those calculated by the supervisor MCU. If one wire was removed, the focus fo the system would be pushed to the MCU without compromised connections.

By simulating an error for each of the parallel MCUs, on different outputs, the system could be caused to toggle the enable lines constantly. Removing a wire from one output of MCU A would cause the system to enable MCU B instead. Removing a different output wire from MCU B would then cause the system to enable MCU A, again. Because the

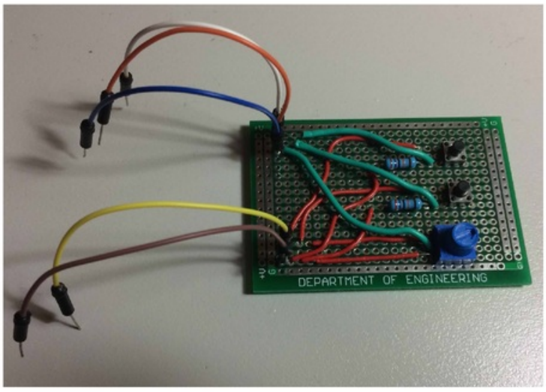
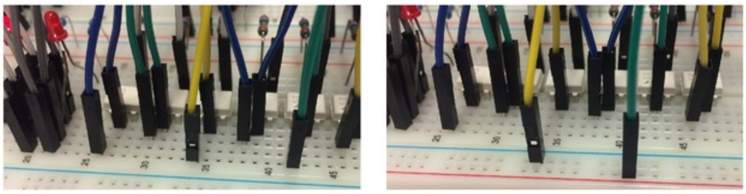


Figure 3.6: Experimental input board.



(a) (b)

Figure 3.7: Wires from the parallel MCUs to the optocouplers (a) connected and (b) disconnected.

process of evaluation and toggling was constant, with two disconnected wires on different MCUs, the system would rapidly toggle the enable lines.

Fig. 3.7(a) shows the wires used for simulating the faults connected according to the circuit diagram. Fig. 3.7(b) shows the same wires disconnected to simulate the errors. The yellow disconnected wire is the output wire from MCU A to the red LED, LED 1. The green wire is the output wire from MCU B to the green LED, LED 2. No other connections used the those side rails on the breadboard.

3.5.3 Behaviour Of Enable Signals

Measurements were taken of the enable signal A under different system conditions, all using the introduced errors to prompt system cycling. A reading was taken for the open-circuit output of enable A. A reading was taken for the closed -circuit output of enable A without the servo motor attached to the system. Lastly, a reading was taken for the closed-circuit output of enable A with the servo motor connected to the system. This experiment is to observe the effects of different components of the system on each other.

During this experiment serial comments were added to the Arduino code for debugging the switching process. This also served as a regulator for time period between inversions of the enable signals. This in provided for the various signal recordings to be plotted on one graph.

3.5.4 PWM Output Without Errors

The PWM signals of both supervised MCUs were monitored along with the enable signal A without any introduced system errors. This experiment is to demonstrate the alignment of PWM outputs from two different MCUs even without introduced errors.

3.5.5 PWM Output With Errors

Measurements were taken of the system PWM output along with both enable signal outputs, A and B while introduced errors cause the system to constantly cycle between primary MCUs. This experiment is to demonstrate the effects of a cycling rotation of primary MCUs on a time sensitive digital signal.

3.5.6 Binary Outputs Without Errors

The system LED outputs were measured along with the button inputs and the enable signals. No wiring errors were introduced during these measurements. This demonstrates the normal operation of the system.

3.5.7 Binary Outputs With Errors

Using the simulated wiring errors, measurements were taken of the LED outputs while the buttons were actuated in a gray code pattern. This provided observation of the behaviour of the core switching capability of the system.

3.6 Results

3.6.1 Behaviour Of Enable Signals

Fig. 3.10 shows the variations in the level of the enable signal for controlling the outputs of MCU A. The system is rotating between primary MCUs due to introduced wiring errors. Subtle variations in the level can be seen in the final output with the servo connected to the system. These variation coincided with the servo changing its angle.

The

An overall reduction in the signal level can be seen comparing the level of the unconnected signal with the level of the signal when connected into the system. This reduction is a voltage drop from 5 V down to approximately 2.6 V.

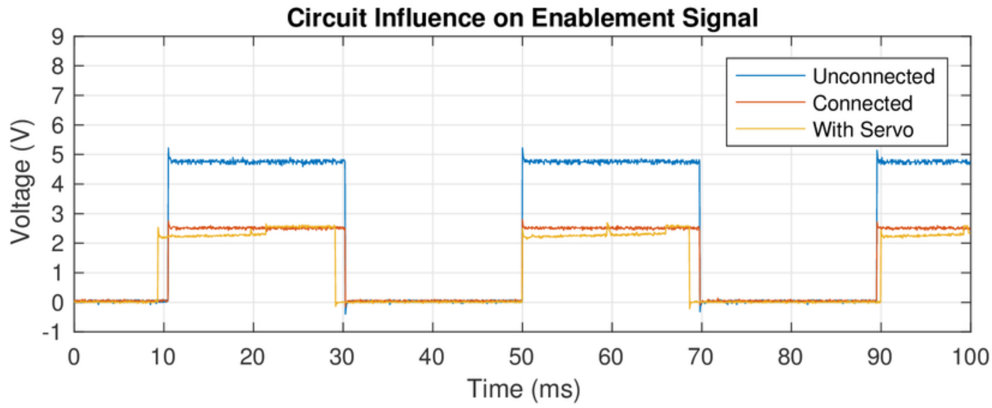


Figure 3.8: The output of enable A from the supervisor MCU shown unconnected to the circuit, connected to the circuit without the servo motor connected, and connected to the circuit with the servo motor connected.

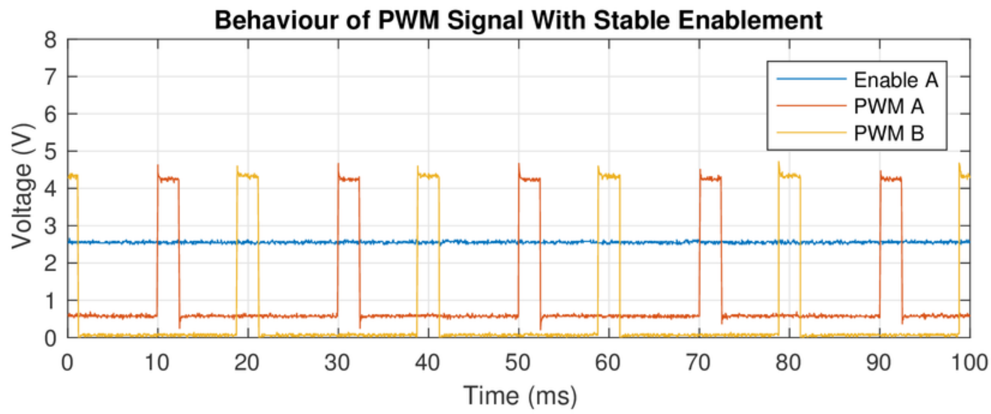


Figure 3.9: Output servo PWM signal alongside a stable enable signal for MCU A .

3.6.2 PWM Output Without Errors

The behaviour of the PWM output signals of both supervised MCUs can be seen in fig. 3.9. As shown in section 3.6.1, the enable signal is steady at approximately 2.6 V, not under constant rotation. PWM signal B has a phase offset from signal A of almost half a period. When PWM signal B is low, the signal still has a measured level of approximately 0.6 V.

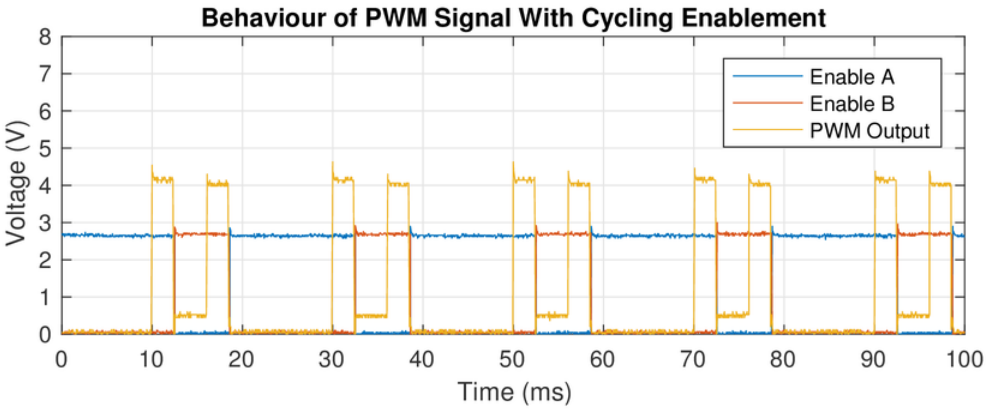


Figure 3.10: Output servo PWM signal alongside a cycling enable signal for MCU A .

3.6.3 PWM Output With Errors

Fig. 3.10 shows the PWM signal output of the system while the enable signals are cycling through inversions. This is the PWM signal that is received by the attached servo motor. The signal is the combination of segments of the PWM signals of both MCUs A and B. Because of the a synchronous timing of these two signals, shown in section 3.6.2, the signal frequency is inconsistent from one transmitted instruction to the next. this can be seen to align with the inversion of the enable signals. Additionally, the variation in the voltage of low signals can also be seen.

3.6.4 Binary Outputs Without Errors

The normal operation of the system inputs and outputs can be seen in fig. 3.11, shown in orange. This plot demonstrates the expected behaviour of the system. The output for LED 1 behaves like an OR-gate of the two button inputs. The output for LED 2 is a copy of button 2.

The system can be seen to invert the enable signals whenever a button is activated or deactivated. The outputs still show their expected outcomes regardless of the primary MCU.

3.6.5 Binary Outputs With Errors

Fig. 3.11 also shows the behaviour of the binary system outputs in blue when wiring errors have been introduced. Because of the wires selected for errors, only one binary output can be activated at a time. This can be seen to cause no major issues while only one high output is required, but significant issues when two outputs are required to raise.

After button 1 is activated at 1 s in fig. 3.11, there is a slight disruption before the output goes high. There is no particular disruption of timing when button 2 is activated

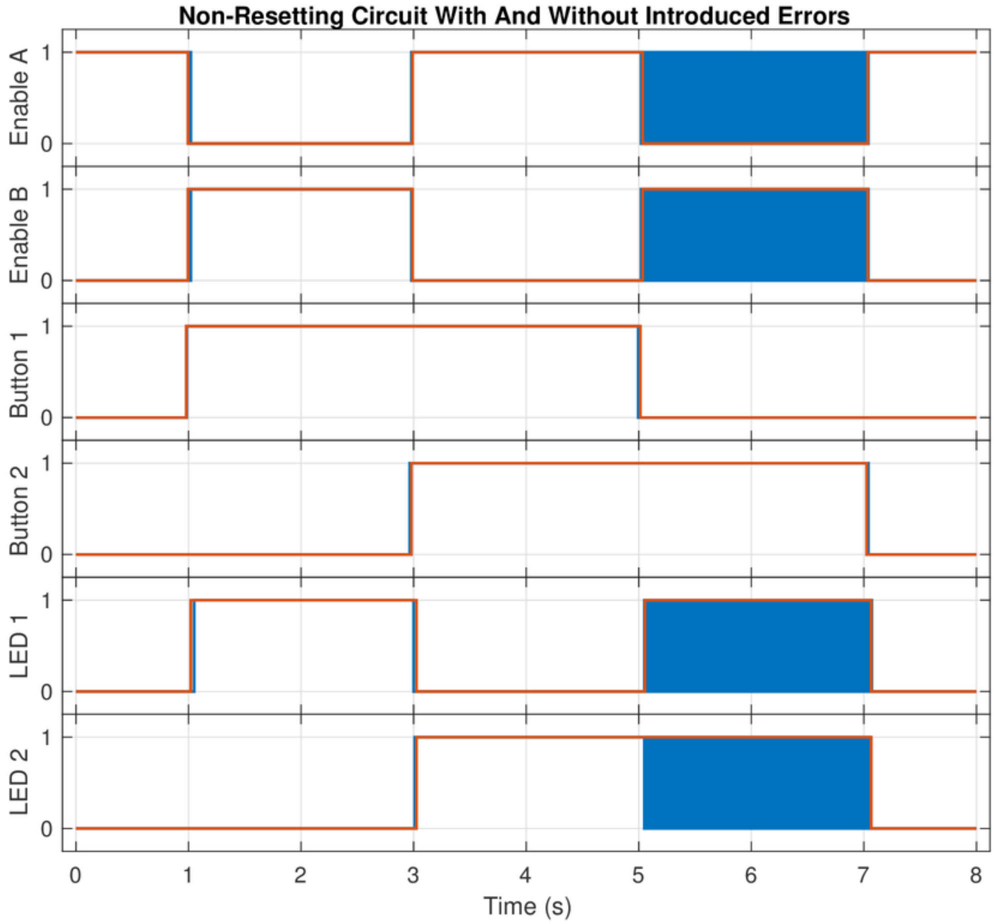


Figure 3.11: Digital logic comparison of enable lines, button inputs and LED outputs of non-resetting system under normal operation shown in orange, and under simulated erroneous operation shown in blue.

at 3 s.

When button 1 is deactivated after 5 s, both binary outputs are supposed to be active. For the duration of this state of button inputs, the system can be seen to rapidly toggle the enable lines. This in turn rapidly toggles the two binary outputs, as their sole sources are connected and disconnected. The resulting output signals are at the desired levels for a percentage of the time.

3.7 Discussion

3.7.1 Regular Enable Inversion

The orange plot lines in fig. 3.11 also show that the system inverts the enable lines even without simulated errors. At every change of buttons, the enable lines were toggled. Asynchronous timing of the three MCUs would have contributed to the momentarily mismatched results that have prompted this change. Button bouncing may also have an impact.

3.7.2 System Settling With One Affecting Error

There is slight disruption at the initial activation of button 1 at approximately 1 s of fig. 3.11. This is because enable A was already high when the button was pressed. When the enable lines toggled to enable B, the system detected that the correct output was not being produced, and so toggled back to enable A. The result of this can be seen in the plot in blue for output LED 1. Momentarily the LED 1 output is low, then reverts to high.

When button 2 is also selected, the system toggles to enable B. If button bouncing and asynchronous timing had not caused this switch, the system would have switched anyway due to the disconnected wire from MCU A to LED 2. With both of these combinations of buttons, only one output LED is required at a time, so the system behaves in a relatively stable state.

3.7.3 System Settling With Two Affecting Errors

Once button 1 is released after 5 s in fig. 3.11, the system attempts to turn on both output LEDs. Since both the wire from MCU A to LED 2 and the wire from MCU B to LED 1 are disconnected, the system has to cycle the enable lines so that both LEDs are activated as much as possible. The resulting output is effectively a partial duty cycle of the intended output.This is clearly seen between 5 s and approximately 7 s. Once button 2 is released, the system resumes a stable state with all inputs and outputs low.

3.7.4 Reduced LED Duty Cycle

Without the servo connected, when the system cycled constantly due to simulated errors, the brightness level of the LEDs was constant when viewed by the human eye. It was noticeably duller, as it was operating at approximately half duty cycle, due to being rapidly switched on and off.

3.7.5 Servo Motor Noise

The servo motor and the main circuit were run from the same power supply in the laboratory. Often when there was a change in servo position, there was a brief surge in current drawn by the servo. This caused simultaneous noise for all of the other components running from the same power supply. An example of this noise can be seen in the enable signal in fig. 3.8.

The noise from the servo operating also caused disruption for the output LEDs. With the servo motor reconnected to the system, variations in brightness were clearly discernible by the human eye that coincided with the movements of the servo motor.

3.7.6 PWM Signal Alignment

Fig. 3.10 shows the system output of the servo PWM signal while the enable lines are being toggled. The PWM signal that results from switching between MCU outputs clearly does not have an even period. The alignment of the PWM signals in fig. 3.10 is also out of alignment, almost as far as a half a period. The system was restarted between recording the values for the two plots, which would explain the difference between the two alignments. The variations in PWM alignment would be due to the asynchronous clocking of the system.

3.7.7 Servo Motor Alignment

Manufacturing discrepancies between components may also have had an impact on this variation in the matching of the PWM signals. The circuit for the peripheral components of MCU A matched that of MCU B. Yet, it was observed the the servo motor would not always return to exactly the same angle after the disruption of toggling the enable lines. The estimated variation would have been only a degree or two, but it was visible to the human eye.

This could have been caused by slight variations in the analogue to digital converters (ADC) on the MCUs. It could also have been due to a variation in the exactness of the oscillator circuits of the MCUs. Either way, there was slight variation between the produced PWM signals of the two MCUs. These variations may also have been related to the mismatched low voltage levels of the two PWM signals.

3.7.8 Experimental Input Board

The experimental input board was produced to overcome further disruptions caused by the temporary nature of prototyping on a breadboard. The push buttons and the pot did not locate securely enough in the breadboard to withstand movement endured under operation. Additionally, jumper wires were bumped every time the buttons were pressed or the pot was adjusted. Owing to the temporary nature of using a breadboard, bumping the jumper wires could clearly be seen to have an influence on the the servo motor. This would have been due to the sensitivity of the ADC on the MCUs. Producing the separate input board solved the problem of loose input components. It also provided the physical separation from the jumper wires to eliminate physical interference.

Table 3.1: Structural combinations of inputs and outputs.

Structure	Definition	Input(s)	Output(s)
SISO	Single Input, Single Output	Button 2	LED 2
SIMO	Single Input, Multiple Output	Button 2	LED 1, LED 2
MISO	Multiple Input, Single Output	Button 1, Button 2	LED 1
MIMO	Multiple Input, Multiple Output	Button 1, Button 2	LED 1, LED 2

3.7.9 Input And Output Combinations

The implemented combination of inputs and outputs served to demonstrate the versatility of input and output combinations. All structural combinations of single and multiple inputs and outputs were demonstrated. These combinations are listed in Table 3.1. This set configuration of inputs and outputs serves quite well for simulating errors and then testing the system performance.

3.7.10 Blind Looping

A significant limitation of this system is that it does not identify that a prospective primary MCU is even functional before assigning it to be the primary MCU. Although this wasn't experimentally tested, if one of the MCUs was incapacitated or altogether missing, there is nothing in this system to prevent it still being assigned as the primary MCU in the event of a single error with the other MCU. While the system would promptly invert back to the existing available MCU after checking the outputs, functionality would have been significantly disrupted.

3.7.11 Non-Resetting

This system has no capacity for resetting a non-functioning MCU. While wiring errors may be overcome, at least momentarily, a jammed MCU will not recover. Perhaps a full system reboot may return the disfunctional MCU to operation, but this cannot be described as self-recovering.

3.7.12 Time-Dependent Signals

Special consideration should be given to time-dependent signals, and high-accuracy signals. The included LED outputs would not necessarily be greatly affected by a brief disruptions. A slight fade will be noticed, but may not be a dire issue for many applications. However the position of a servo motor may be a critical point in a process line. If the position of a servo twitches and slightly changes angle every time the primary MCU is reassigned, then complications may be introduced to the machinery being operated in a particular application.

3.7.13 Shared Power Sources

For applications with relatively high current requirement, consideration should be given to the separation of the circuitry. Shared circuitry, particularly the power supply, can lead to mutual degradation of both the enable signals and PWM signals.

Chapter 4

Supervised Parallel Resetting Redundancy

4.1 Introduction

Section 4.2 details the behaviour of the system through the control flowchart. Section 4.3 explains the design of the required circuit diagram. The developed Arduino code is laid out in 4.4. The process of experimentally testing the system is explained in Section 4.5. The plotted results in Section 4.6 are then discussed in Section 4.7

The system laid out in this chapter builds on the design of the previous chapter. Rather than blindly switching to another available MCU, this system checks for an active life signal from the destination MCU before switching. If the other supervised MCU is available and showing signs of life, the system will enable the replacement MCU as the primary and reset the former primary MCU.

The pot input and PWM servo output have been omitted from this circuit. This takes the focus off the disruptions of the PWM signal due to switching primary control to another MCU, and instead focuses the behaviour of the system as a whole.

This system is expected to accommodate situations where a supervised MCU becomes inactive due to erroneous programming. This layer of resilience is intended to add to the capability of accommodating wiring errors.

4.2 Control Flow

The foundational flowchart for the development of this system is shown in fig. 4.1. The main cycle of analysing the inputs and responding to discrepancies can be seen on the left-hand side. After the supervisor performs its calculations, the same as those on the supervised MCUs, the supervisor then compares its calculations with the measured outputs of the system. If the outputs match expectations, the system proceeds to the next cycle.

If the calculated outputs of the supervisor do not match what is measured from the

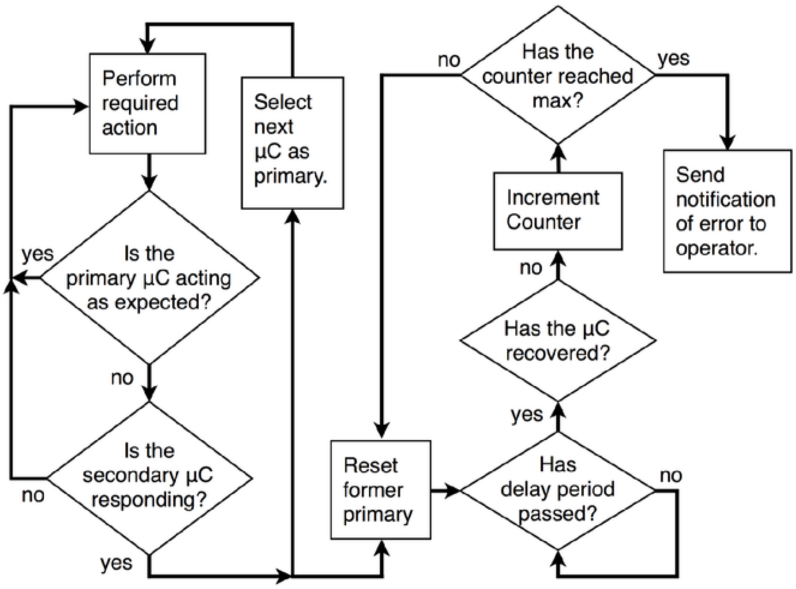


Figure 4.1: Operational flow chart of supervised system.

system outputs, the supervisor will try to shift primary control to the other supervised MCU. If the replacement MCU does not produce a discernible response signal, the supervisor won't change the primary responsibility.

If the available MCU provides a response when it is checked, two steps are conducted. Firstly primary priority is changed to the available MCU. Secondly, the former primary is then reset. This initiates a full restart of the MCU that produced an erroneous output for the system.

Again, this system doesn't blindly reset then continue on its process. The system will wait for a prescribed length of time before checking for a response from the rebooted MCU. If the MCU has rebooted back to operation, the reset process is completed. If no response is detected, the process of resetting and waiting for a response.

If the MCU has succumbed to a fatal error, no amount of resetting will bring it back to operation. So a maximum number of resets will be attempted before giving up. In a practical situation, if this system lost the functionality of one of its MCUs, it would be desirable to know that a non-recoverable error had occurred. So a notification step has been added to alert the operator to the non-recoverable fault, if the maximum count is reached.

4.3 Redundant Circuit

The circuit for this resetting system is based extensively on the circuit used in the non-resetting system of the previous chapter. The circuit is shown in Appendix B.3. The core circuit consists of a supervisor MCU monitoring the inputs and outputs of two parallel MCUs. The outputs of the supervised MCUs are activated and deactivated by the use of optocouplers controlled by enable lines from the supervisor MCU. The input consists two NO PBs. The outputs consist of two LEDs.

The modifications to this circuit are the addition of four extra wires between the supervisor MCU and the two supervised MCUs as well as an alert notification LED. Reset lines have been added to each of the supervised MCUs from the supervisor. These are shown in the circuit diagram in an aqua colour. Life wires to indicate response from the supervised MCUs to the supervisor are shown in a blue colour. A blue alert LED with its accompanying resistor has been added as output of the supervisor MCU. This LED is for notifying of a non-recovering MCU.

4.4 Arduino Software Code

The Arduino Code for the parallel supervised MCUs is listed in Appendix C.2. This sketch is extensively based on the code for the supervised MCUs from the previous non-resetting system. The overall system LED outputs are calculated by each of the supervised MCUs. Since no software is required for an external reset, no additional code is required to add that function to this sketch. The only addition is the output of a signal to indicate that the MCU has started again after a reset. The pot input and servo output of the first supervised sketch have been removed.

4.5 Experimentation

This circuit was experimentally implemented in a breadboard arrangement as seen in fig. 4.2. Though largely based on the experimental circuit of the previous chapter, necessary changes were made to fit with the circuit diagram in Appendix B.3. An alert LED was added to indicate a non-responsive MCU. Components for the servo motor were removed, however the soldered pot remained on the experimental input board.

The system was powered from a GW Instek programmable power supply, model PWT-3203. Measurements and plot data were captured using two four-channel benchtop oscilloscopes, Agilent DSO-X 2024A. The data was filtered and plotted using Matlab scripts.

Errors were simulated in the form of MCU output errors, and complete MCU failure. The output errors were introduced as a disconnected wire from each MCU to an output LED. A different output was used for each of the MCUs as shown in fig. 3.7(a) and fig. 3.7(b). The complete MCU failure was simulated by using a hand-held switch to pull the reset input for MCU A to ground. By holding the reset pin for a MCU to ground, the MCU will not respond until it is released.

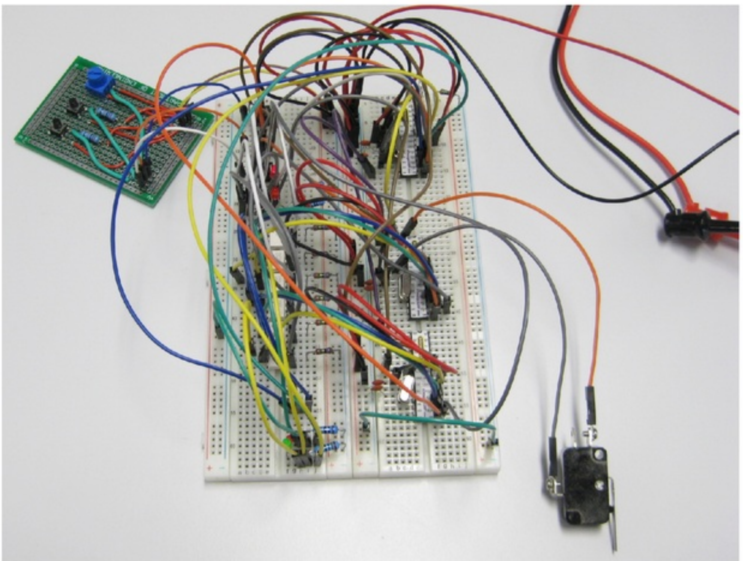


Figure 4.2: Experimental resetting supervised circuit with a switch to simulate a non-responsive MCU.

Plot data was captured for the system operating without any simulated errors. This allowed observation of the system behaviour under normal incidental cycling as had been observed for the non-resetting system. Measurements were taken over the duration of the input buttons being cycled in a gray code pattern.

Data was collected for the system operating with MCU A disabled temporarily using the hand-held switch. This allowed observation of the limited reset function of the system. The alert feature could also be checked to fit with the flowchart in fig. 4.1.

Measurements were taken with introduced wiring issues in the system, as described above. This plot data would show the recovery pattern of the circuit after switching to the next available MCU and resetting the former primary MCU. The system response to multiple errors would also be shown. As each MCU had a different disconnected wire with an outlet, the supervisor MCU cycles the primary priority between the two parallel MCUs.

4.6 Results

4.6.1 Normal Operation

The logged data for the resetting system under normal operation can be seen in fig. 4.3. Data is shown for the enable lines and life lines of both parallel MCUs, alongside data of both PB inputs and both LED outputs.

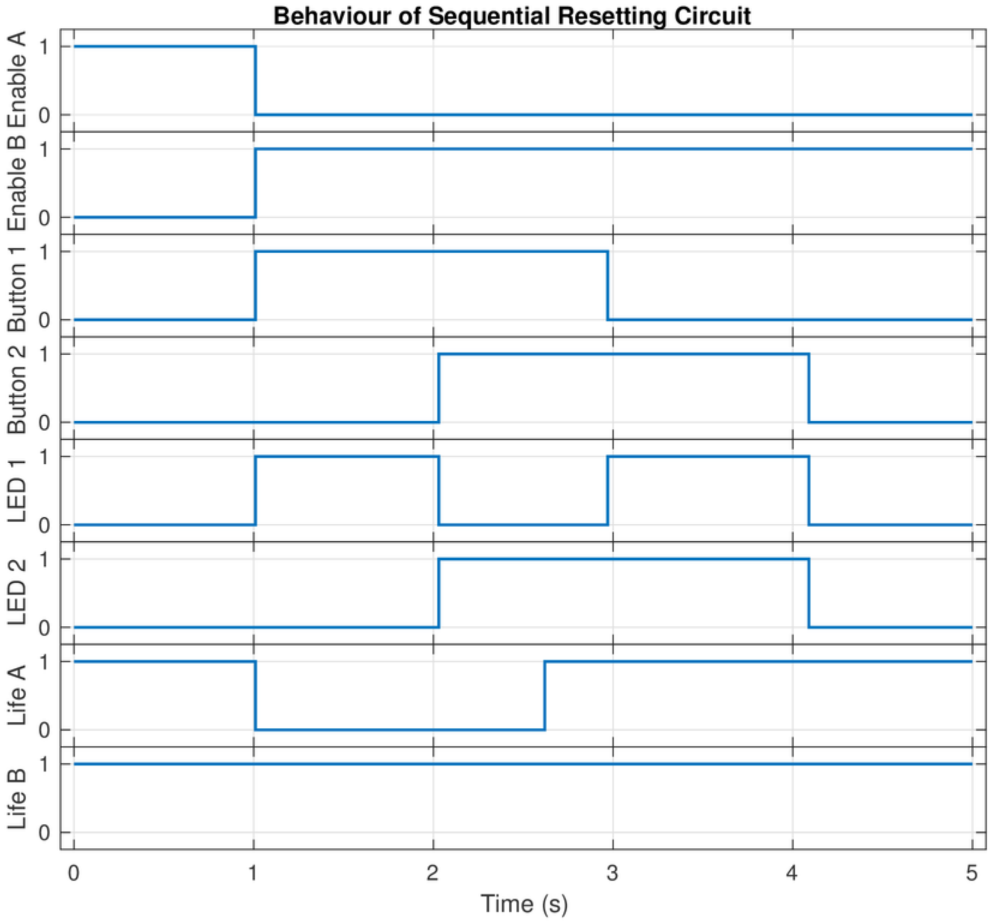


Figure 4.3: Binary plot of resetting supervised circuit in operation without errors.

4.6.2 Simulated Non-Responsive MCU A

The data of the outputs of enable A, reset A, life A, and the alert LED has been plotted in binary form in fig. 4.4. This shows the behaviour of the system attempting the specified three times to restart the erroneous MCU A. This plot also shows the system behaviour if a non-responsive MCU returns to active operation, even after giving up on restart attempts.

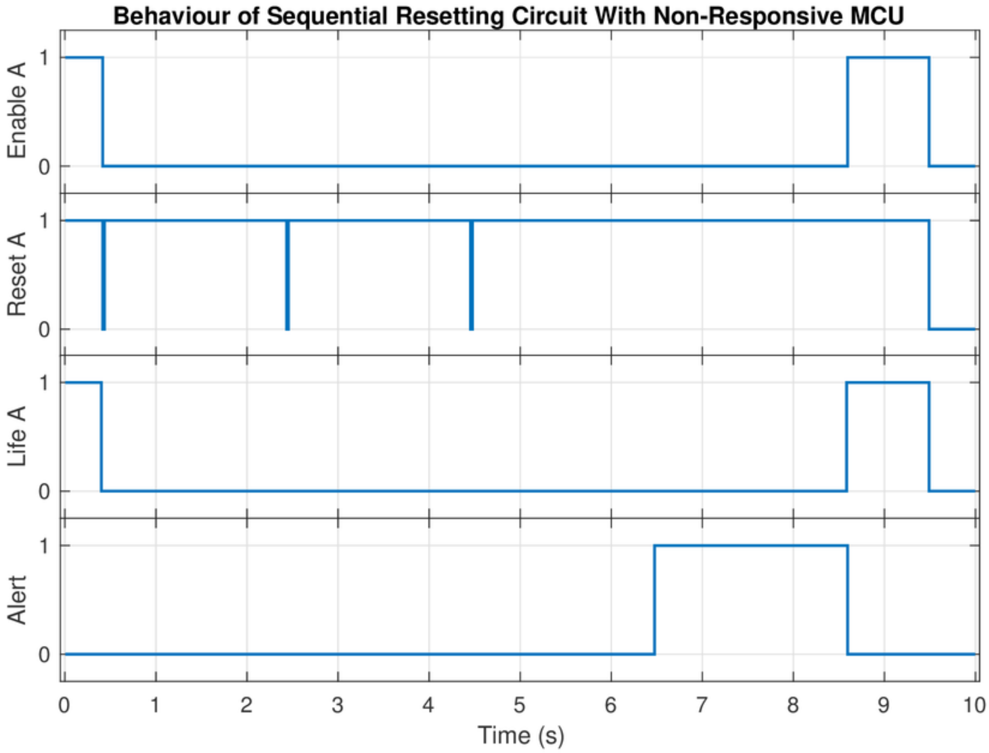


Figure 4.4: Resetting supervised circuit attempting to reset a non-responsive MCU with signals converted into a binary plot.

4.6.3 Simulated Wiring Errors With Automatic Response Recovery

Fig. 4.5 shows the behaviour of the system with individual wiring errors for each parallel MCU. There is a section of the code that checks for the response of a reset MCU and if there is no response, it proceeds to reset the erroneous MCU again. This section of code causes a situation where it is possible for both the primary and secondary MCUs to be reset at once. As can be seen at the 2 s line, the enable signals toggle to have signal A high, and reset B. However, an error was still detected in MCU A, so it was also reset. The effect is that only MCU B actually spends any active time at the same time as being the designated primary. This would have the effect of only having one output being active, and even then on a half duty cycle.

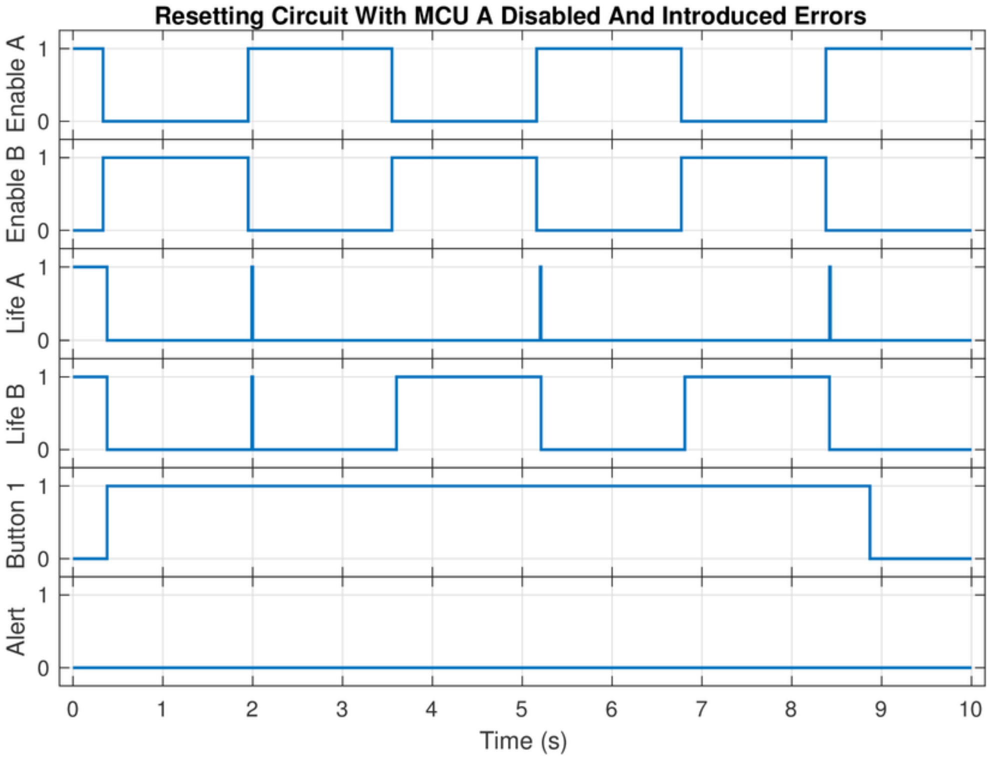


Figure 4.5: Binary plots of reset input and life output for both MCU A and MCU B under continuous cycling, with automatic response recovery.

4.6.4 Simulated Wiring Errors

In fig. 4.7, the resetting system is shown cycling between available MCUs, without the automatic response recovery capability. As the supervisor cycles through this shortened code, each cycle has to wait for the reset MCU to restart, reinitialise and be ready for operation, outputting a high-active life signal.

4.7 Discussion

4.7.1 Normal Operation

As shown in fig. 4.3, under normal operating conditions, the resetting system behaves similarly to the non-resetting system. The first output, LED 1, functions as an XOR of the two input buttons. The second output, LED 2, is a direct forwarding of button 2. Similar to the non-resetting system, changes in states of the inputs can trigger a change

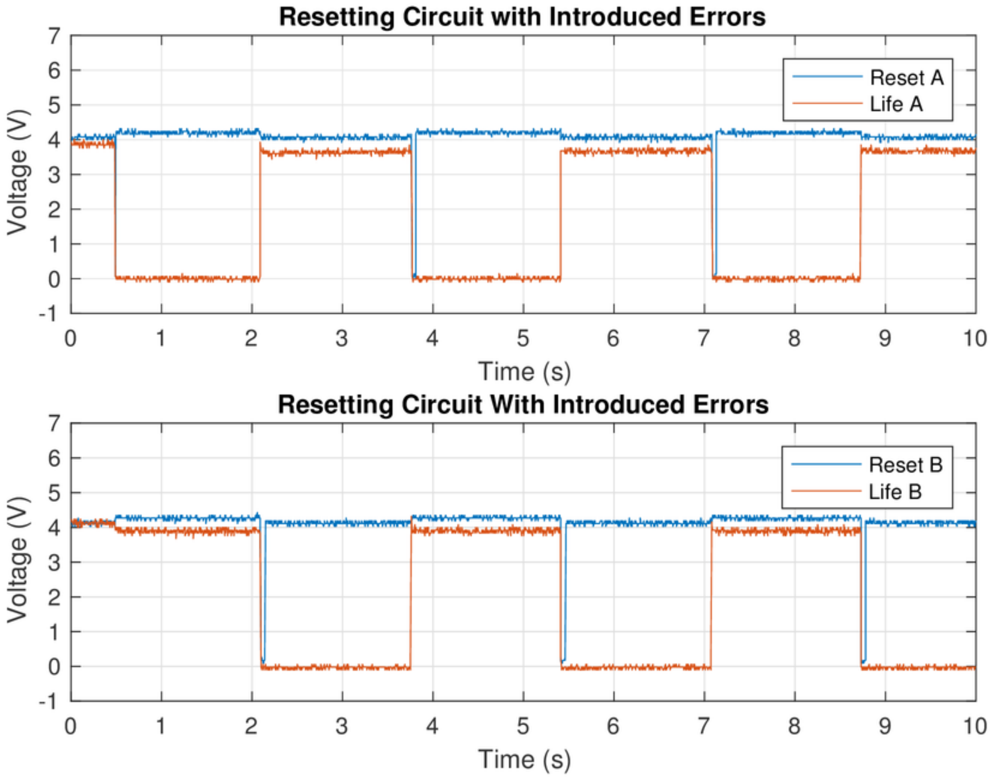


Figure 4.6: Plots of reset input and life output for both MCU A and MCU B under continuous cycling.

of MCU priority. This can be seen in fig. 4.3 at the 1 s line. This disruption most likely results from the system operating asynchronously.

The major difference for this system is that each time the system changes to a different primary MCU, the former primary is reset, as seen between the 1 s line and the 3 s line in fig. 4.3. Assuming all wires are connected and no components are failing as in this circuit, the system continues to operate normally while subtly restarting the former primary MCU.

4.7.2 Limited Resetting

If the restarted MCU does not respond after a specified length of time, the system resets the MCU again. This process can be seen in fig. 4.4 repeating up to a total of 3 resets. This number of restarts can easily be adjusted in the variable declarations in Appendix C.5.

If the supervisor detects a response from a restarted MCU, the resetting ceases. If no

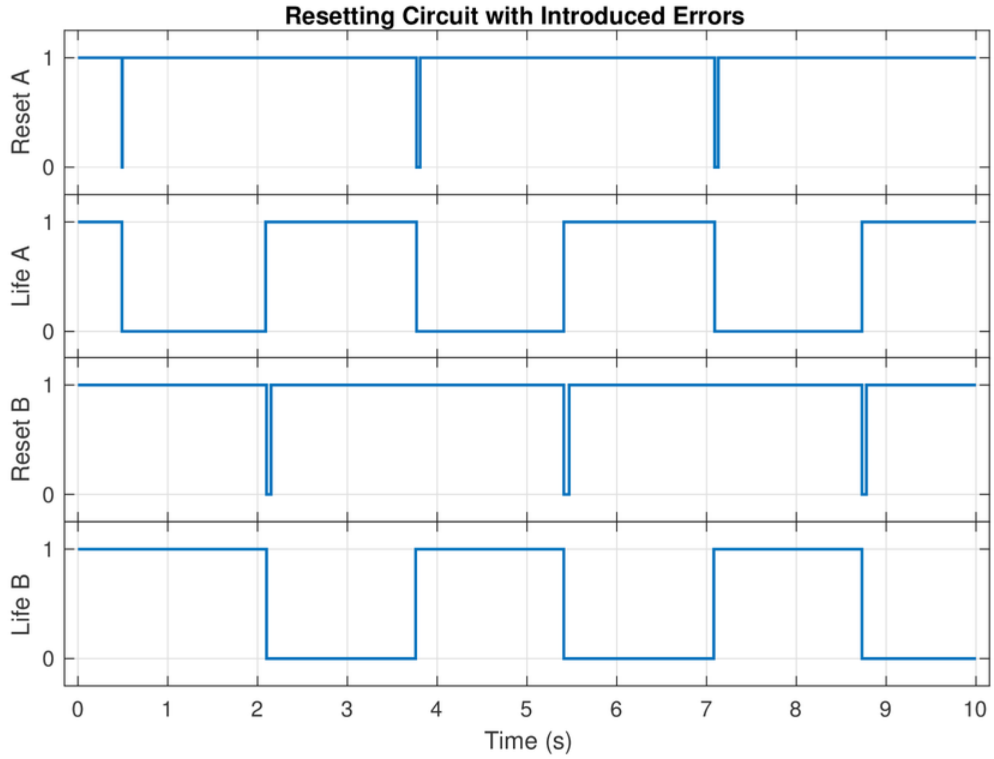


Figure 4.7: Binary plots of reset input and life output for both MCU A and MCU B under continuous cycling without automatic response recovery.

response is detected by the time the maximum resets is reached, an alert is issued to the system operator. For experimental purposes, this was simplified to an illuminated LED. In fig. 4.4 the alert signal can be seen activating at approximately 6.5 s.

As this non-responsiveness was caused by experimentally holding the reset pin on MCU A to ground, the resetting could similarly be released at a controlled time. This reactivation was carried out, and a response can be seen in fig. 4.4 at approximately 8.5 s. As the supervisor detects this response, the alert pin is deactivated, and MCU A is available for normal operation. This can be seen by the Enable A signal switching to high at the same time as it is given primary priority.

4.7.3 Cycled Resetting With Errors

Fig. 4.5 shows the sporadic operation of a system that has introduced wiring issues. As the system is trying to monitor the secondary MCU and reset it as needed, the progression of the code creates the situation that both parallel MCUs are reset at the same time. This issue was not resolved. Instead that section of code was removed to view the constant toggling of the primary priority.

4.7.4 Cycled Resetting With Errors

With the automatic response recovery commented out from the code, the system will continue to cycle whenever it finds errors. The resultant loop of resetting each MCU can be seen clearly in fig. 4.7. This circuit has had one error introduced for each of the parallel MCUs. By the time the old MCU has restarted, declared its variables, and initialised its outputs and variables, around 1700 ms have passed. This means that the gaps in the operation of the outputs will not only be visible to the human eye, there will be significant gaps in the operation of the system.

4.7.5 Independent Resetting

The ideal situation would be to have necessary MCU resets completed without disruption to other system components. However, if there is more than one error, the timing of one MCU being reset may depend on the other MCU having completed being reset. If timing was not an issue, both could be reset as needed, however there is a risk that both, or all supervised MCUs could be out of operation at one time, leaving outputs without their feeds. The solution for this is a trade-off between independence and the consistency of availability.

4.7.6 Unnecessary Resetting

It is unlikely that resetting the MCU will resolve wiring connectivity issues, if not impossible. Resetting MCUs should then be reserved for instances of non-responsiveness and possibly situations requiring re-initialisation of inputs and variables. Non-responsive MCUs could be reset using a watchdog timer. Specific decision making would be needed to determine the need for re-initialisation of inputs and variables. This decision-making would probably be quite dependent on the nature of the desired system and its various inputs and outputs.

Chapter 5

Self-Supervised Parallel Redundancy

5.1 Introduction

This system implements a system of self-supervision. Using three flag signals between the MCUs, the primary MCU excludes other MCUs from conflicting control in performing the system function. However, if the primary MCU does not complete the required function within the expected time, another MCU can prompt the system to reselect a primary MCU. This allows another MCU to take control to achieve the system function, in the event of an error.

The content of the chapter is presented in the following order. The flow of control is developed in section 5.2. The circuit diagram is developed in section 5.3. Section 5.4 details the components and progression of the required Arduino code. The experimentation is documented in section 5.5 with the results presented in section 5.6. Finally the system and its experimental performance is discussed in section 5.7.

5.2 Control Flow

This system is based on what is known as an atomic lock. In this system, once a primary MCU is identified, other MCUs are restricted from taking access while the primary MCU retains control. This avoids the situation of two MCUs attempting simultaneous control. A difference from a traditional view of an atomic lock is that instead of a central circuit or controller identifying an MCU as the active primary, this system relies on the current primary MCU preventing the remaining MCUs from assuming control.

If multiple MCUs simultaneously controlled the main components, issues would be experienced in the system. Conflicting electronic signals could lead to short circuits. Conflicting data signals could lead to corrupted data being conveyed.

Fig. 5.1 shows the flowchart developed as the design basis for this system. This flowchart shows the decision-making process for analysing the current state of the system,

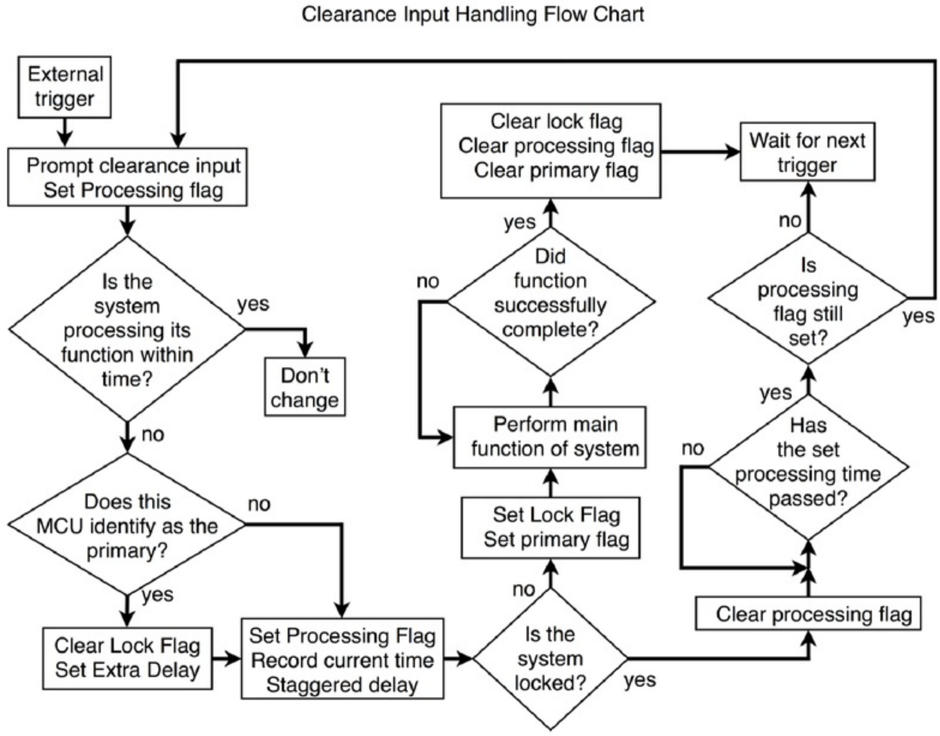


Figure 5.1: Flowchart for self-supervised parallel redundancy system.

selecting a primary MCU, the consequent action of the selected primary MCU, and the consequent action of the remaining available MCUs.

5.2.1 Processing Flag

A processing flag is used to identify that the function is currently being performed. This flag can be set by any or all of the system MCUs. While one of the MCUs has set the processing flag, the system will attempt to complete the specified function. Any non-primary MCUs will clear their processing flag output once they have detected that another MCU has assumed primary priority. The primary MCU will only clear its output for the processing flag once the system function has been completed. Once all of the MCUs have cleared their processing flag outputs, the processing flag will be cleared.

5.2.2 Locked Flag

The locked flag is used to identify that a MCU has assumed primary priority. This flag restricts any other MCUs from assuming simultaneous control with the first MCU. Like the processing flag, any of the MCUs can set the locked flag, however only one achieves this at a time.

5.2.3 Clearance Flag

The clearance flag is an overriding call for attention from the MCUs. Any one of the MCUs can set the clearance flag but it is cleared immediately to prompt just one response. This would occur anytime a non-primary MCU detects that they system is not behaving as expected. As can be seen in the top left of fig. 5.1, the external trigger also prompts the same process. Either of these two sources causes each MCU to evaluate the current status of the system.

5.2.4 Primary Flag

The primary flag denotes that a particular MCU is the active primary MCU. Unlike the processing, lock, and clearance flags, the primary flag is a software flag simply for use in activating certain sections of the code when the MCU is the primary.

5.2.5 State Analysis

The left-hand side of the flowchart in fig. 5.1 details the process of checking the current status of the system after triggering. This system has been designed as an on-demand system. This means rather than looping indefinitely, it is triggered externally to perform a set function.

Since this trigger may not wait for the current function performance to be completed, the current progression needs to be analysed before proceeding to the function performance. To achieve this, an expected time frame is set for completion of the specified function. If the system is interrupted during processing, and the time frame is still within expectations, then no changes are made to the priorities of the MCUs. The primary MCU continues processing as before, and the non-primary MCUs continue evaluating the timing of the completion of the system function. As will be detailed shortly, one of the non-primary MCUs can trigger a reanalysis of the current system state if it calculates that the function processing is taking a longer time than expected.

5.2.6 Primary Selection

The lower left of the flowchart in fig. 5.1 shows the process of selecting the MCU to take primary priority. Assuming that the system has progressed through the state analysis section, the system is required to commence a full process of the system function. This could be from a spontaneous triggering of the external trigger. It could also be due to the previous process not be completed in the expected time, perhaps due to an error.

If the system had just previously been processing the system function, the primary flag would still be set on the acting primary MCU. If there had been an error, it is undesirable for that MCU to take control. If, however, it is the only remaining MCU, it still needs to have opportunity to take control. Thus if the former primary MCU still has its primary flag set, it will wait an extra period of time before attempting to assume primary priority. This gives time for another available MCU to take priority first, without completely deactivating the previous primary MCU.

After setting any additional delays as required, all MCUs set the processing flag and record the current time as the start of processing. This time is used for calculating the lapsed time taken to process the system function. This time may be used by the primary MCU and by all of the non-primary MCUs. So this time is recorded separately by each of the MCUs.

The first MCU to set the lock flag assumes primary priority. Complications could arise from multiple MCUs attempting to set the lock flag simultaneously. If multiple MCUs checked the locked flag at exactly the same moment, then set the lock flag simultaneously, simultaneous control could be attempted. This could lead to short-circuits or corrupted data. To alleviate this error, each MCU needs to wait a unique length of time.

To achieve a unique delay time for each MCU, either each will require individual programming or each needs a hard-wired input to designate a unique identification (ID) to each MCU. Whatever the means, each MCU needs a way to calculate how long it should delay, and thus in what order it should attempt to assume control of primary priority.

After waiting for a unique period of time, each MCU will attempt to assume primary priority. This decided by checking the lock flag. If the lock flag is not already set, the MCU will set the lock flag and assume primary priority. If the lock flag is already set, then the MCU will assume non-primary priority. In this way the first MCU to assume primary priority locks out all remaining available MCUs.

5.2.7 Primary MCU Response

The subsequent response of the new primary MCU is shown in fig. 5.1 proceeding up the central section of the flowchart. The MCU enters a loop of performing the system function, checking for completion of the function, and performing the process again if required.

Infinite Sequence Processing

If the system is required to perform for a particular length of time, the system could merely be checking to see if the lapsed time has exceeded the expected timeframe. Once this time has been reached or exceeded, the primary MCU decommissions itself as the primary.

Finite Sequence Processing

If the required function is a finite sequence of steps, the system can be programmed to check the effectiveness of the processing. If the function is not completed effectively, it may be appropriate to trigger the clearance flag to prompt another MCU to take over primary priority. If the function has been successfully completed, the primary MCU then decommissions itself as the primary MCU.

5.2.8 Non-Primary MCU Response

Fig. 5.1 shows the response of the non-primary MCUs after primary priority has been assumed by one MCU. This response is shown progressing up the right-hand side of fig. 5.1. After checking that the lock flag has been set and thus an MCU has assumed primary priority, the non-primary MCUs clear their outputs for the processing flag. They then wait for the expected process time to pass.

5.2.9 Primary MCU Decommission

After the primary MCU has determined that the process has been completed, the system needs to be notified. The processing flag is cleared to prevent unnecessary reprocessing. The lock flag is cleared to enable any available MCUs to respond when the system is next triggered. Finally, the onboard primary flag is cleared so that no extra delays are initiated at the next triggering.

After the expected time has passed for the system function to be completed by the primary MCU, the non-primary MCUs check the processing flag. If the processing flag is clear then the process is assumed to have completed. Any non-primary MCUs then wait for the next external trigger to recommence the whole process.

If the processing flag is still set after the expected timeframe for processing has lapsed, then it is assumed that there is an error in the primary. All non-primary MCUs will have cleared their processing flag outputs by now. Thus if the processing flag is still set, the system needs to be prompted to change primary priority to another MCU. The whole state analysis is then triggered.

5.3 Circuit Design

The circuit for this system is shown in Appendix B.4. The main components of this system are the parallel MCUs, necessary OR-gate ICs, and resistors for fixed identification of each MCU. A 5V power supply provides the required power, and a pushbutton is used for an input trigger.

A clock circuit is also required for each of the MCUs. This circuit is made of a 16 MHz crystal oscillator and two accompanying 22 pF ceramic capacitors. This clock circuit shown in fig. 3.3. This clock circuit connects to pins 9 and 10, the two oscillator input pins of an Atmega328.

Table 5.1: Values for calculating ID resistors.

ID	Required			Actual		
	Voltage	Resistor 1	Resistor 2	Resistor 1	Resistor 2	Voltage
1	1.67 V	66 kΩ	33 kΩ	68 kΩ	33 kΩ	1.63 V
2	3.33 V	33 kΩ	66 kΩ	33 kΩ	68 kΩ	3.37 V
3	5 V	0 Ω	0 Ω	0 Ω	0 Ω	5 V

5.3.1 Fixed Identification

As discussed in section 5.2.6, the system requires a method of uniquely identifying each parallel MCU. Using an analogue input pin on an Atmega328, a variety of input voltages is used to distinguish the different IDs. Each ID input employs a non-zero input voltage. A voltage divider is used to create the intermediate voltage values between 0 V to 5 V. For this circuit three MCUs are used, so three different values are required in three approximately even steps up from 0 V to 5 V.

The specifications for the resistors were calculated using (5.1) and listed in Table 5.1. Since not all values of resistors are readily available, close available values were selected. The resulting theoretical voltage levels are also included in Table 5.1 as calculated using (5.2). The circuits for each of the three inputs are shown in fig. 5.2.

$$\frac{R_2}{R_1} = \frac{v_i}{v_i - v_o} - 1 \tag{5.1}$$

$$v_o = v_i - R_1 \frac{v_i}{R_1 + R_2} \tag{5.2}$$

5.3.2 Pin Assignment

The pin assignments for each parallel redundant MCU is listed in Table 5.2. The power supply is connected to pins 7 and 8, the main power input pins of the MCU. The clock circuit is connected to pins 9 and 10.

The clearance flag input is assigned to digital pin 2. This pin is the primary of two available interrupt pins on an Atmega328, pins 2 and 3. The clearance input also incorporates the main system trigger. The clearance output pin, analogue pin 1, feeds an OR-gate so that any of the MCUs can prompt a priority assignment on clearance interrupt pin.

The processing flag input is read through digital pin 3. Although this pin is available as an interrupt pin, it is not used as an interrupt in this situation. Therefore the assignment could be reassigned to a different pin if the interrupt pin was required for a chosen system

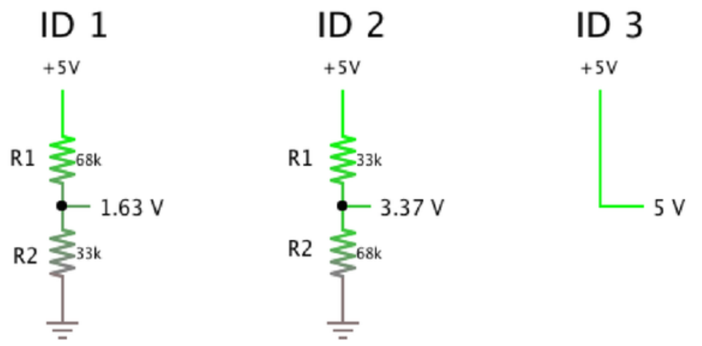


Figure 5.2: Voltage divider circuits used for fixing the MCU IDs according to location.

function. The processing input pin receives the collaborative signal of the processing output pin, analogue pin 3, on all the parallel MCUs.

The lock flag input is received through analogue pin 4. This received signal can be set by any of the parallel MCUs through the lock flag output, analogue pin 2. This signal is used to exclude any other MCUs from taking on the primary role if one MCU has already assumed that position.

Each of the analogue pins used as an input produce their read value using the onboard analogue to digital converter (ADC). While the overall system runs on 5 V, in real applications there is sometimes variation in this voltage level due to manufacturing errors and circuit faults. These variations are taken into consideration by the Atmega328. The system voltage rails are connected to pins 21 and 22, ground and the analogue reference pin, respectively. This provides the MCU with an accurate value of the system voltage for comparison to the analogue pins being read. If a 4.5 V level is read on an analogue pin, and the system reference is also 4.5 V, the measure input will be at the full value. The highest possible value from an analogue pin is 1024, using the onboard ADC. This is because the onboard ADC uses 10 bits of resolution. If the measured input is 4.5 V, but the analogue reference input is 5 V, the integer result will be 922 as shown in (5.3).

$$value_{input} = (2^{10})\left(\frac{v_i}{v_{ref}}\right) => (1024)\left(\frac{4.5V}{5V}\right) = 922 \tag{5.3}$$

5.3.3 Pull-Down Resistors

Additional pull-down resistors have been omitted from the circuit diagram that will be included in experimental configurations of the system. These resistors should be used on all inputs on the OR-gates, in addition to all used inputs on the MCUs. These provide grounding whenever a signal is not high, while still preventing short-circuits when the

Table 5.2: Atmega328 pin assignment for self-supervised redundant system.

Pin	Function	Assignment	Pin	Function	Assignment
1	Reset		28	A5	
2	D0/Rx		27	A4	Lock Flag In
3	D1/Tx		26	A3	Processing Flag Out
4	D2	Clear Flag Interrupt	25	A2	Lock Flag Out
5	D3	Processing Flag In	24	A1	Clear Flag Out
6	D4		23	A0	Fixed ID In
7	Vcc	Vcc	22	Gnd	Gnd
8	Gnd	Gnd	21	ARef	5V
9	Osc1	Clock Input	20	AVcc	
10	Osc2	Clock Input	19	D13	
11	D5		18	D12	
12	D6		17	D11	
13	D7		16	D10	
14	D8		15	D9	Primary MCU LED

signal is high. In the event of s dislodged wire these pull-down resistors would prevent a floating value from the input. A floating input value would give erratic readings leading to erratic system behaviour.

Similar pull-up resistors should also be applied to the reset lines of the MCUs. Rather than connecting between the input and ground, a pull-up resistor connects between the input and the system voltage rail. This provides a constant high value. The reset pin on an Atmega328 is low active. This means that to avoid triggering the MCU to completely reset, the reset pin should be held high. Using a resistor rather than a jumper wire protects from short circuits if the reset pin is used for in-system programming.

5.3.4 Gated Distribution

OR-gates are used extensively for the collaboration of the MCUs in this parallel redundancy system. The signal outputs for all of the system flags are channeled through OR-gates before being directed back to each of the MCUs as inputs. If they were not, one MCU may have a high output while another MCU had a low output. This would cause a short-circuit, damaging the circuit.

For any application of this system, OR-gates or an equivalent isolation method should be used on all MCU outputs which collaborate to control a single destination. Input pins do not require OR-gates because reading a pin signal does not pose a risk of short-circuiting.

5.4 Arduino Software Code

The Arduino code for this system is shown in Appendix C.6, developed using the Arduino integrated development environment (IDE) 1.8.5. This code has been designed to be identically loaded onto all parallel MCUs in this self-supervised parallel redundancy system. The only difference between the different MCUs in the system is that each increase in ID has to wait for an additional timeframe to attempt claiming primary control.

5.4.1 Declarations

The Arduino code for this system begins with the declaration of pins and variables. The first declarations are of the system flag inputs and outputs. After these, any additional pins are declared. For this system the only extra pins used are an LED output to indicate the current primary status, and an input pin for a debugging button used to introduce WDT errors.

After all the pins are declared, variables for the exclusion lock are specified. System flag storage variables are declared here. Also, variables for keeping track of the time taken to process the system function. Flags for controlling functionality after the occurrence of a clearance flag or a WDT timeout are also declared here.

Some of the variables are declared with the extra specification of being volatile. This characteristic is used for variables that may be changed within an interrupt subroutine (ISR). By denoting the variable as volatile, the variable is recognised as being able to be changed at any moment in the event of an interruption. Without this denotation, changes made within the ISR may not take effect if the variable is in the middle of being modified outside of the ISR.

Next, components for the WDT are declared. The library for the WDT is included at this point to provide the extra coding required for implementing and adjusting the configuration of the WDT. Here also, the flag for the WDT override is declared. This flag is for restricting the functionality of the main loop after a WDT timeout has occurred.

Finally, any variables are declared for use in performing the desired system function. This where any future variables should be declared for future applications of the system. For this configuration, the only additional required variable is the specified maximum processing time. This variable is really a core feature of the system used by non-primary MCUs for checking the performance of the designated primary MCU. However, it is a specification that will need to be configured to suit every application of the system. If the maximum time is set too low, the system will never conclude its intended function. If the variable is set to high, a primary MCU may be encountering errors without being detected by the non-primary MCUs.

5.4.2 Setup

The setup function contains all initialisation steps and process commencements required for enabling different functions on the MCU. All required pin modes are set here according to required inputs or outputs. The clearance ISR is attached to the required interrupt pin, in this case, digital pin 2. Because digital pin 2 is the first of two interrupts on the Atmega328, it can also be identified as interrupt pin 0. This provides a concise statement of inclusion. The WDT setup function is also called here to establish the desired WDT configuration before moving on to the system function.

The fixed ID is also set in the setup function. Line 51 of the code in Appendix C.6 includes all the steps to map an analogue input voltage to an integer ID value. The input pin is read, using the onboard ADC. This produces a value without units in the range of 0 to 1024. This value is then divided by 1024, giving a percentage and multiplied by the number of parallel MCUs, in this case, 3. Converting this to an integer gives a neat value. Since this value is later only used to determine a waiting period, a non-integer value could also be used if desired.

5.4.3 Clearance Interrupt Subroutine

The clearance ISR is the code routine that is called whenever the interrupt pin is prompted. When the clearance ISR is attached to the interrupt pin in setup function, the interrupt is specified to react to any change, either rising or falling.

This interrupt is called whenever the external trigger prompts the system to perform its function. The interrupt is also called whenever a WDT timeout occurs on any primary or non-primary MCU. It is also called whenever a non-primary MCU calculates that the primary MCU is taking longer than expected to conclude the specified system function.

A series of if-statements determine what situation would have caused the interruption and what course of action to take. If the primary is still processing the system function within the expected timeframe when the interruption occurs, no action is taken. If any action is required, the return-to-start flag is set to skip through sections of the code. This reduces the time taken to reach the section of code where the primary priority is determined. If this MCU was the primary MCU when the interruption occurred and if action is required, the lock flag is cleared. This allows another MCU to take over the primary role when the role is assigned.

5.4.4 Main System Loop

The main system loop continues to cycle through a series of four different sections of code. The first section of code limits the functionality of the MCU in the event of a WDT. The second section assigns the priority of the MCU in the event of a clearance interruption if necessary. The final two sections are either the section for the primary MCU or the section for the non-primary MCU.

WDT Timeout Response

If a WDT error has occurred, two aspects of the system require consideration. Firstly, the MCU needs to handover to another available MCU. A clearance flag prompt is set if the MCU was the previous primary MCU. This assumes that if this MCU was the primary MCU, then the system is currently in the middle of processing the system function. Whether or not the current MCU was the former primary, if the processing flag is set, this section of code keeps it set to help ensure a handover if necessary.

The second consideration of this piece of code is a limitation. Once a WDT timeout has occurred, if a WDT closing function is enabled, the system will continue processing whatever function it was in the middle of just prior to the timeout. The MCU will continue for a length of time equal to the length of the WDT timeout. If another MCU has been assigned as the primary MCU, this could lead to conflicting control actions. Therefore this section of code causes the MCU to return to the start of the main loop, just after starting the loop. In this way, the MCU is prevented from performing parts of the system function that could clash with the newly designated primary MCU. The clearance flag output is cleared before returning to the start of the main loop.

Valid Interruption Response

If the ISR determines that an interruption is valid, the system will set the new process flag. This enables the second section of code in the main loop. The first step is to clear the new process flag. This prevents continued unnecessary and potentially problematic reassignment of primary priority.

Any extra delays are specified next. If the current MCU was formerly the primary MCU, an extra delay of 100 ms is set. This will delay the former primary MCU before it can take on the primary role. This is designed to allow any other available MCUs to take control first. While the former primary MCU will be delayed, in the event of it being the last available MCU, it will still be able to regain primary control if necessary. After this delay is set, the primary MCU flag is cleared.

If the current MCU was not formerly the primary MCU, then no extra delay is set. This allows the non-primary MCU to attempt to gain control before the former primary MCU attempts to regain primary control.

Having identified any extra delays, all functioning MCUs set their processing flag outputs. This means that the processing flag will be set until an available MCU has taken on the primary role. This is intended to make the system resilient to hiccups that may cause the system to forget that it had to perform the system function.

All delays are then performed. The former primary MCU waits for its extra delay, if it was still the primary MCU when a valid interruption occurred. Next, all of the MCUs wait for a delay period that increases according to its fixed ID. In this way, the situation is avoided where multiple MCUs attempt to assume primary priority simultaneously. Such clashes could damage equipment and corrupt data.

Each MCU, after its delay, will then check and respond to the lock flag input. If the lock flag is not set, the first MCU to respond will take on the primary role. This MCU will then set its lock flag output, excluding all other available MCUs. This new primary MCU will also set its own primary MCU flag. This set software flag then enables the relevant section of the main code loop. It also contributes to the next assignment of primary priority, as just discussed.

If the lock flag is already set when an MCU checks, the MCU resorts to a non-primary role. This MCU can now clear its processing flag, since the primary MCU is now performing the processing function. This will also mean that when the primary MCU concludes the process and clears its processing output flag, the system processing flag will be cleared. This prevents unnecessary takeover attempts due to perceived incomplete processing.

Primary MCU System Function

For the primary MCU, the next main section of code is to perform the main function of the system. This could be reading input signals, processing some form of data, or writing outputs. The function may include combinations of all three of these options.

The section of the code for the primary MCU must also have a conclusion section. This code identifies for the primary MCU, that the system function has successfully reached its conclusion, and that the MCU can decommission itself as the primary MCU. For the included configuration, the conclusion is reached when a certain time has elapsed since the last valid interruption. For this to work effectively, the elapsed time must be updated prior to comparing it with the desired conclusion time. For a simple example, this is earlier in the code. For a complex application, this update should be conducted just prior to the comparison if-statement.

As with the elapsed time update, the system function code should also include regular resets of the WDT. The simple example used here does not perform lengthy processes to achieve its goal, so a single reset halfway through the main loop is sufficient. However, complex applications should include resets at a suitable regularity to prevent timeouts.

If the system function has reached its conclusion the primary MCU must decommission itself. This involves clearing the lock flag output and the processing flag output. This prevents another MCU from taking over primary priority when the system function has already been concluded. The software primary MCU flag also needs to be cleared. This prevents unnecessary delays when the system is next triggered externally.

Non-Primary MCU Function

Rather than following the system function code like the primary MCU, any non-primary MCUs complete a short section of code, that compares the completion timeframe of the primary MCU with an expected timeframe. If the elapsed time has surpassed the maximum allowed time for completion, non-primary MCUs check the processing flag input. If the processing flag is still set, then it is assumed that the primary MCU has encountered an error. The clearance flag is then toggled on and off to prompt a reassignment of primary responsibility. If the system has not yet reached the maximum time for processing, or the processing flag is not set when the maximum time is reached, then no action is taken.

Before this checking on the primary MCU, non-primary MCUs clear their own flag outputs for the lock and processing flags. This prevents false alarms for priority reassignment action. While these steps will be completed multiple times as the code for the non-primary MCU cycles, it ensures the prevention of accidental false alarms.

Debugging Function

A debugging function has been added to the design. This function reads an input pin from a button. If the button is pressed, the WDT reset is disabled. In this way, WDT timeouts can be introduced for controlled testing. The code for this function can be seen in lines 148 to 153 of Appendix C.6.

5.4.5 Clearance Flag

Throughout the Arduino code, the clearance flag can be seen as the means of prompting action from the system. The incorporated external trigger initiates action from the system. Whenever an error is detected, either from a WDT or from a non-primary MCU monitoring the primary MCU, the clearance flag is used to prompt the interrupt pin of each MCU. All parallel MCUs contribute to the clearance flag. The clearance flag outputs of each of the MCUs all connect to the inputs of an OR-gate arrangement along with the external trigger signal.

The uniform output of the clearance flag OR-gate is then read by each parallel MCU. In this way, any parallel MCU can prompt every parallel MCU to assess the current situation. From either a primary or a non-primary MCU the system can be prompted to reassign primary priority to another available MCU if necessary.

5.4.6 Lock Flag

The lock flag is used to exclude all remaining available MCUs from primary priority once one MCU has gained the primary role. If a non-primary MCU reads a cleared lock flag after a valid interruption, it will claim primary priority. If, however, a non-primary MCU reads a set lock flag at that time, it will not be able to gain control until after the expected processing time has elapsed. Like the clearance flag, the lock flag is facilitated by an OR-gate(s).

5.4.7 Processing Flag

The processing flag is used for three different situations. It is used primarily for checking if the primary MCU is still processing the system function after the expected time. The processing flag is also used to aid the handover process from a primary MCU with errors to an available non-primary MCU. Finally, the processing flag is used for preventing

Table 5.3: WDT control register bits [8].

Bit	Label	Name	Function
7	WDIF	Watchdog Interrupt Flag	Used in the operation of the WDT
6	WDIE	Watchdog Interrupt Enable	Enables the WDT closing function
5	WDP3	WDT Prescaler 3	Bit 3 for setting the WDT duration
4	WDCE	Watchdog Change Enable	Enable bit for configuration access
3	WDE	WDT Reset Enable	Enables the WDT to operate
2	WDP2	WDT Prescaler 2	Bit 2 for setting the WDT duration
1	WDP1	WDT Prescaler 1	Bit 1 for setting the WDT duration
0	WDP0	WDT Prescaler 0	Bit 0 for setting the WDT duration

interruptions within the expected processing timeframe. As with the clearance flag and lock flag, the processing flag is made possible using OR-gates.

5.4.8 WDT Setup

The WDT setup function is used for establishing and or modifying the internal WDT of the Atmega328 MCUs used for this system. All interrupts are disabled to prevent interruption while the WDT is configured. Next, the WDT is reset, in case a previously installed sketch had implemented a short WDT timeout. The next step is to enter the configuration mode of the WDT. Changes of mode and the actual configuration is achieved by setting particular values in the WDT control register (WDTCSR). The bits of the register, along with their names and descriptions, are listed in Table 5.3.

After the configuration mode has been accessed by setting the WDTCSR to B00011000, the desired configuration is entered. This again is achieved by setting the WDTCSR to the desired values. The configuration used for this application is B01001100 with the most significant bit (MSB) on the left hand, and the least significant bit (LSB) on the right hand. This activates the closing WDT function, called the WDT interrupt, with the bit second from the left, bit 6. Bits 0, 1, 2, and 4 are used to set the timeout duration in milliseconds. Here the duration is set to 250 ms. The available duration options are listed in Table 5.4. Bit 3 is set to enable the WDT to operate, and bit 4 is cleared to exit the configuration mode. Having setup the WDT, the interrupts are then re-enabled, concluding the WDT setup process.

Table 5.4: Available WDT timer durations [8].

WDP3	WDP2	WDP1	WDP0	Duration
0	0	0	0	16 ms
0	0	0	1	32 ms
0	0	1	0	64 ms
0	0	1	1	125 ms
0	1	0	0	250 ms
0	1	0	1	500 ms
0	1	1	0	1 s
0	1	1	1	2 s
1	0	0	0	4 s
1	0	0	1	8 s
1	0	1	0	Reserved
⋮	⋮	⋮	⋮	⋮
1	1	1	1	Reserved

5.4.9 WDT Closing Function

The WDT closing function, also called the WDT ISR, is executed when a WDT timeout occurs. This function allows concluding actions to be taken. In this application the function is used to handover primary priority to an available non-primary MCU. This process is set in motion in three lines of code. The first step is to set the start return flag. This reduces the time taken for the the main loop to return to its start in the event of an interruption. The second step sets the WDT override flag. This causes the main loop to continuously return to the beginning of the main loop preventing interference with the next appointed primary MCU. Finally, the lock flag is cleared to allow the next primary MCU to take on the primary role.

5.5 Experimentation

The self-supervised parallel redundancy system was experimentally tested in a laboratory. The experimental configuration can be seen in fig. 5.3. The Atmel Atmega328P was used for all MCUs in this experiment. A triple 3-input OR-gate DIP IC, model CD74HC4075E, from Texas Instruments was used for this experiment. A quad 2-input OR-gate DIP IC, model 74HC32AP from Toshiba was also used. Power was supplied from a GW Instek

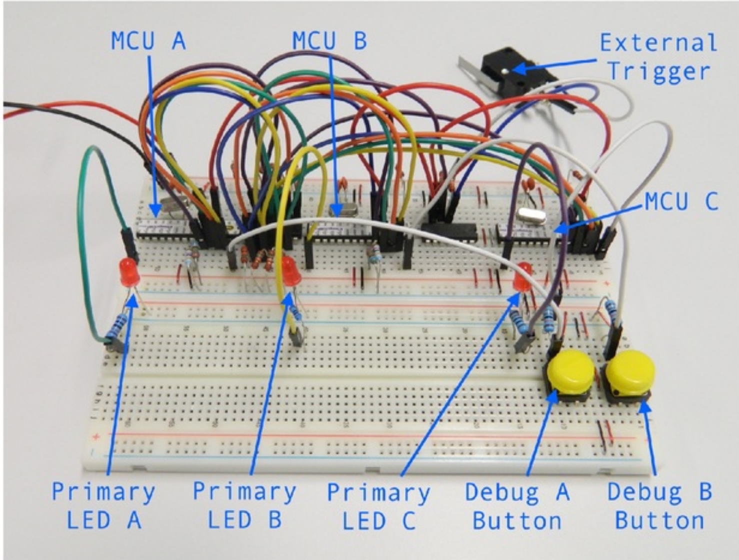


Figure 5.3: Experimental configuration of self-supervised redundant parallel MCU system.

programmable power supply, model PWT-3203.

An Agilent DSO-X 2024A benchtop oscilloscope was used to collect the data for plotting. Plots were prepared from the raw data using Matlab. Plot data was taken in a variety of situations, using introduced circumstances to monitor the behaviour and performance of the system. All data is plotted in split binary plots to emphasise the timing of the different system components.

5.5.1 System Flag Operation

The behaviour of the system flags was monitored with no errors in the system, to demonstrate the basic operation of the system. Data was collected of the four system flags produced by MCU A, the clearance flag, the lock flag, the processing flag, and the software primary flag. The output LED of the primary MCU flag was used to gather plot data for the primary MCU flag. Ordinarily, this flag is merely a software flag, only used by the MCU that produces it.

Data was collected of processing flags produced by all three MCUs after initial triggering. This was collected to demonstrate the staggered attempts by the parallel MCUs, to gain primary control of the system function.

Data was also collected for the system lock flag, and the processing flags from MCUs A and B after triggering. This further demonstrates the exclusion process for preventing any subsequent available MCUs from taking primary priority after the primary role has already been secured.

5.5.2 Varied Triggering

The system was subjected to a sequence of varied triggering, to observe its response depending on the progression of the system function. The trigger was held for a length of time greater than a second but less than the full duration of the system function. Next, the trigger was held for a period longer than the full system function. Lastly, the system was triggered within the duration of the system function being processed.

5.5.3 Prolonged Processing

The next experiment was to test the ability of a non-primary MCU taking over control, if the primary MCU took longer than expected to complete the system function. For this experiment, MCU A was programmed with a modified code sketch. The only modification was the extension of the conclusion if-statement. Instead of concluding after 2000 ms, the conclusion time was extended to 4000 ms. The maximum processing time remained at 3000 ms on all the parallel MCUs.

5.5.4 WDT Timeout

Experiments were conducted to test the system response to WDT errors. These WDT timeouts were introduced using the debugging button to disable the WDT resetting. This code is specified in section 5.4.4. Using this feature, static WDT timeouts were introduced to last the entire experiment. Dynamically introduced WDT timeouts could then be started at a particular time. Buttons were used on MCUs A and B. The button input wire for MCU C was connected to ground, keeping the input low.

WDT Statically Interrupted Operation

Experimental data was collected of the system flags and the primary MCU flag of MCU A. This experiment implemented WDT timeouts on MCU A. This experiment was to demonstrate the system continuing through to the conclusion of the system function, even with the WDT timeout.

The same experiment was conducted with WDT timeouts on MCUs A and B. Data was collected for the primary MCU flags of all three MCUs. This experiment was to demonstrate the timing of the handover process between MCUs.

WDT Spontaneously Interrupted Operation

The handover process was experimentally tested using an introduced WDT timeout part-way through the processing of the system function. The system was externally triggered without any introduced WDT timeouts. Before the conclusion of the system process, a WDT timeout was introduced to MCU A, in order to prompt a handover to another

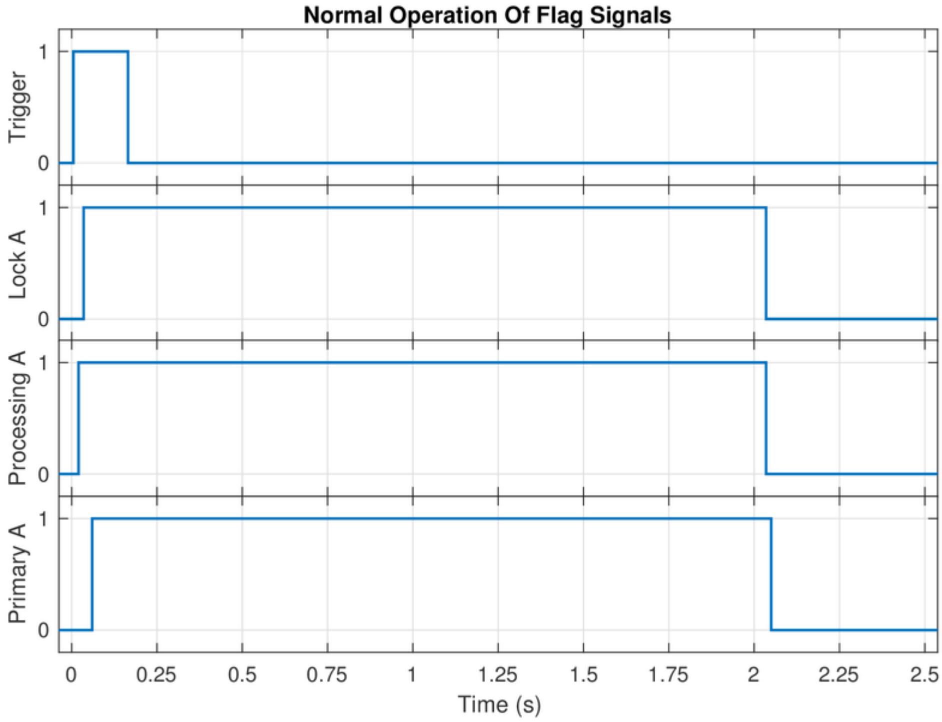


Figure 5.4: All flag signals for MCU A under normal operation in the self-supervised system.

available MCU. For this experiment, the processing output flags of MCUs A and B were recorded alongside the system clearance and lock flags.

A similar experiment was also conducted using prolonged triggering. For this experiment, the primary MCU flags were monitored for all three parallel MCUs. After a brief external triggering, a WDT timeout was introduced to MCU A, before the conclusion of the system function. The process was repeated with an extended external triggering.

5.6 Results

This section shows the experimental results for the self-supervised system. The collected raw data, using the laboratory oscilloscope, has been mapped to binary values. This mapping and the layered subfigures, have been used to emphasise the system timing.

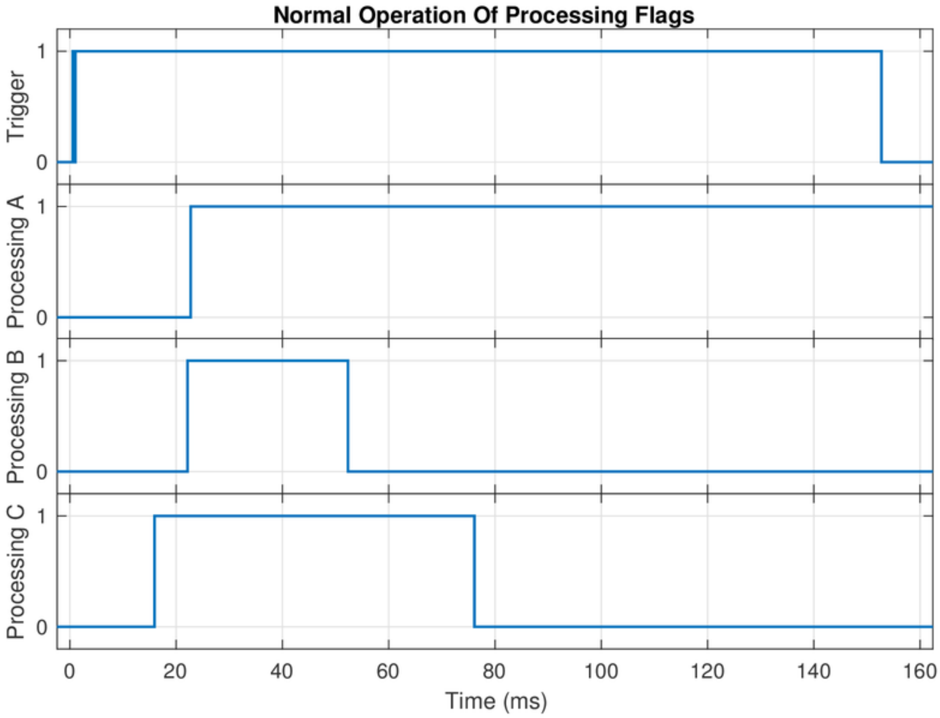


Figure 5.5: Behaviour of system flags.

5.6.1 System Flag Operation

Fig. 5.4 shows the flag signals of MCU A. The trigger input was human activated. The processing flag is immediately raised. Shortly afterwards MCU A activates the lock flag, and activates its own primary MCU flag. All three flags continue to just after 2 s. First, the lock flag and processing flag are cleared almost simultaneously. Then, the primary MCU flag is cleared. All flags are then clear.

Fig. 5.5 shows the output of all three processing flags from the three parallel MCUs. All three are activated shortly after triggering. The processing flag for MCU A continues through and after the end of the plot. The processing flags for MCU B and MCU C clear after increasing periods of time.

Fig. 5.6 shows the behaviour of the processing flags for both MCU A and MCU B after triggering, along with the system lock flag. Both MCU A and MCU B activate their processing flag outputs shortly after triggering. The processing flag for MCU A continues throughout the plot. The processing flag for MCU B falls after approximately 30 ms. The lock flag sets at approximately 30 ms and continues through to the end of the plot.

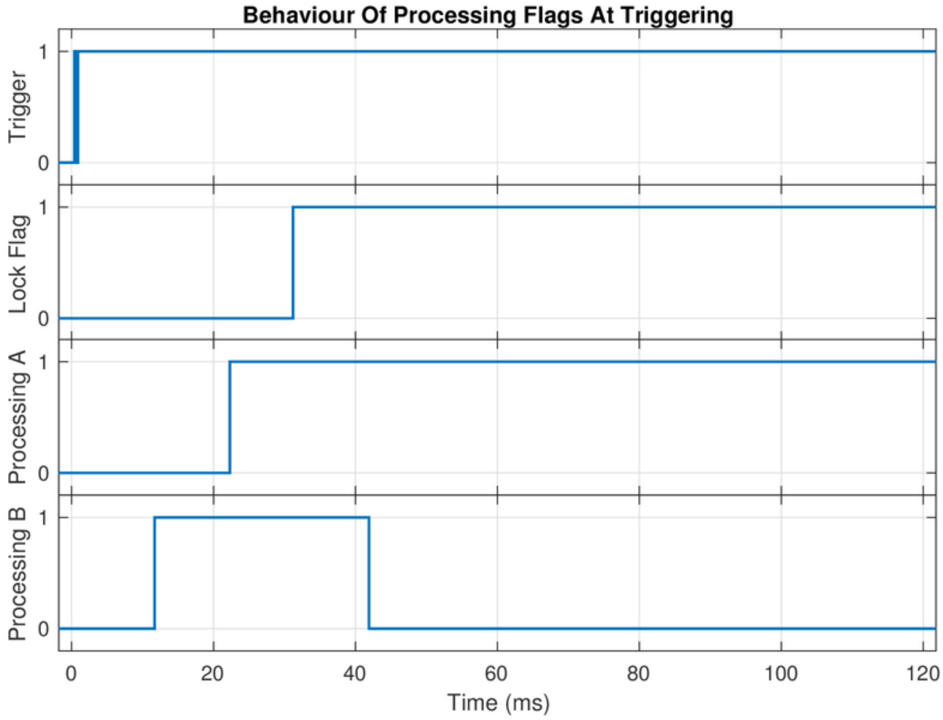


Figure 5.6: Zoomed plot of system flag behaviour after triggering.

5.6.2 Varied Triggering

The system response to a variety of trigger lengths can be seen in fig. 5.7. First a trigger of approximately 1.25 s is activated. Then a trigger of over 3 s is received by the system. Finally, a relatively short trigger of less than 0.25 s is input. The lock flag, processing flag A, and processing flag B are initiated, subsequent to the first and second triggering. These three flags are all also prompted by the falling of the second triggering, however, none of the flags are affected by the third triggering. The lock flag and processing flag A remain set for approximately 2 seconds following each prompting. Processing flag B clears shortly after each prompting.

5.6.3 Prolonged Processing

Fig. 5.8 shows the system response to MCU A, as it takes longer than expected to perform the system function. After the trigger is manually activated, the processing flag is set by both MCUs A and B, and the lock flag is set. MCUs B and C are set to perform the system function for 2 s, but to take over if the function is still progressing 3 s after the trigger or clearance flag has been toggled. For this experiment MCU A has been modified

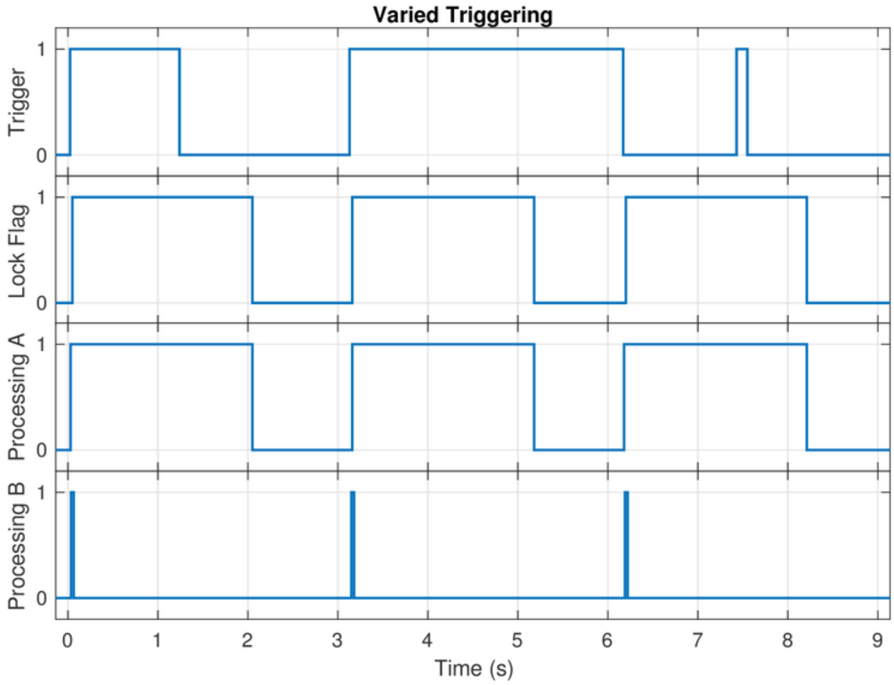


Figure 5.7: Various triggering times and durations for a self-supervised system.

to perform the system function, retaining primary priority for 4 s.

After 3 s of function processing by MCU A, a reassignment of primary priority is prompted and MCU B takes over processing responsibility from MCU A. At that point the lock flag is cleared and reset. MCU B maintains a set processing flag for 2 s, then concludes operation, clearing the system flags.

5.6.4 Watchdog Timeout

Statically Interrupted Operation

Fig. 5.9 shows the behaviour of the system flags, and the primary flag for MCU A, in response to a WDT error. The trigger is set by human activation. The processing flag is subsequently set, the lock flag follows shortly, and the primary flag for MCU A is the last flag to be set. After 0.25 s the lock flag is cleared. Approximately 0.2 s later the lock flag is again set, and the primary flag for MCU A clears shortly afterwards. Just before the primary flag for MCU is cleared, the trigger signal, incorporated into as the clearance flag, is set for a moment. The processing flag and lock flag continue from the reset for approximately 2 s before concluding.

Fig. 5.10 shows the progression of the system finding an available MCU, when others

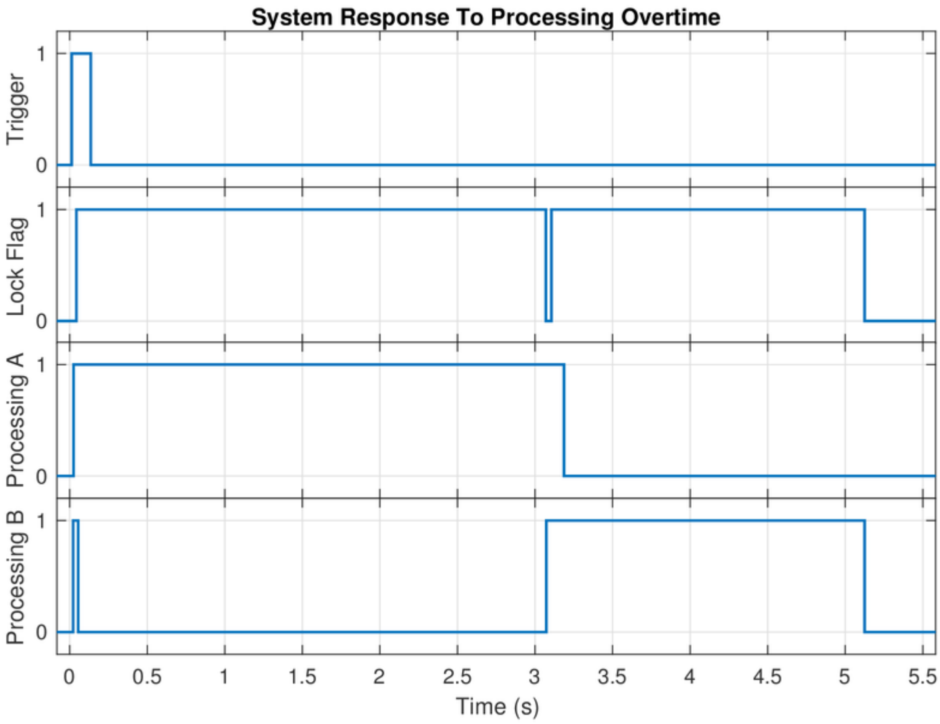


Figure 5.8: System response a prolonged processing time.

have issues causing their WDTs to reset them. After the initial triggering by human activation, MCU B activates primary priority. This only lasts for less than 0.5 s before ending, and MCU C taking over primary responsibility. MCU C retains control for approximately 2 s, before concluding.

A more complicated handover of control between MCUs is seen in the second half of fig. 5.10. After human triggering, MCU A takes control. Primary control rotates between MCUs A and B three times before MCU C secures primary priority, and holds it for approximately 2 s. The trigger signal, also used as the clearance signal can be seen oscillating each time MCU A is releasing primary priority to another MCU. No such oscillations are seen when MCU B releases primary priority to another MCU.

Spontaneously Interrupted Operation

The system response to a spontaneous continued watchdog timeout is shown in fig. 5.11. The trigger is activated briefly by human activation. The processing flags A and B are subsequently set. The system lock flag follows shortly afterwards. The processing flag B clears quickly after setting. The processing flag A continues to be set until almost 1.5 s.

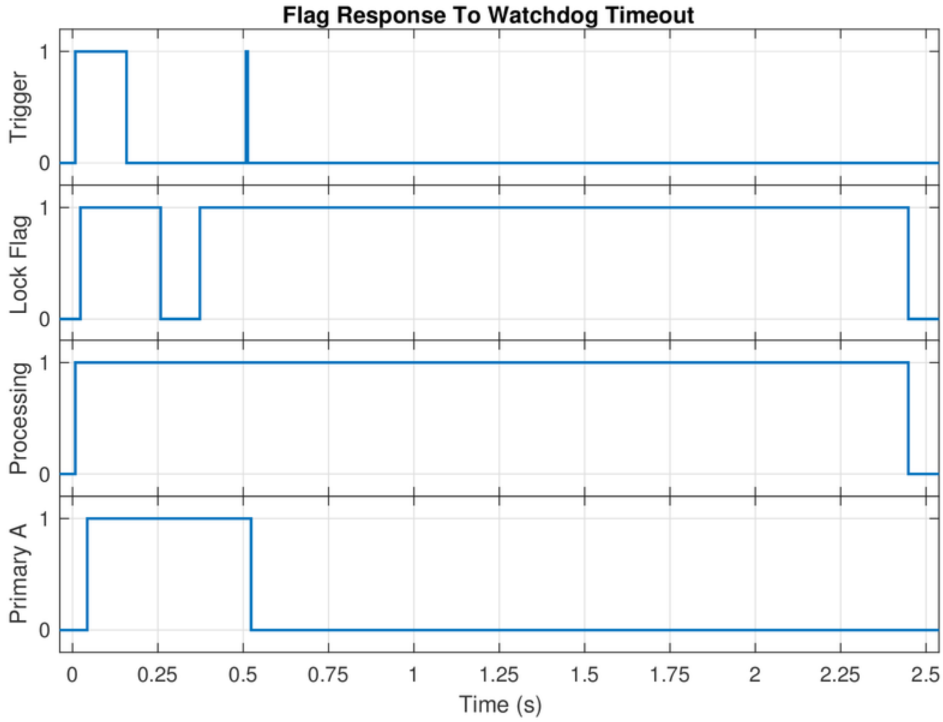


Figure 5.9: Behaviour of system flags after single watchdog timeout.

The lock flag remains set until approximately 1.2 s, when the processing flag B sets again, this time remaining on for approximately 2 s. The processing flag A loops through a cycle of clearing for approximately 0.3 s, then setting for 2.5 s.

Fig. 5.12 shows the system responding to spontaneous individual WDT resets on MCU A. The initial human triggering prompts MCU A to secure primary priority. This is maintained for just over 1 s. At this point, a single WDT error was experimentally introduced. MCU C takes over primary priority and concludes after approximately 2 s. No setting of the clearance flag is visible at the time of handover from MCU A to C.

The second human trigger seen in fig. 5.12 is sustained for a period of almost 3 s. Meanwhile, an ongoing WDT error was experimentally introduced after approximately 1 s. The primary flag of MCU A can be seen to clear after approximately 1 s of operation. MCU B does not assume the primary priority until the trigger is released, approximately 1.5 s after MCU A had released primary priority. MCU B retains control for 2 s.

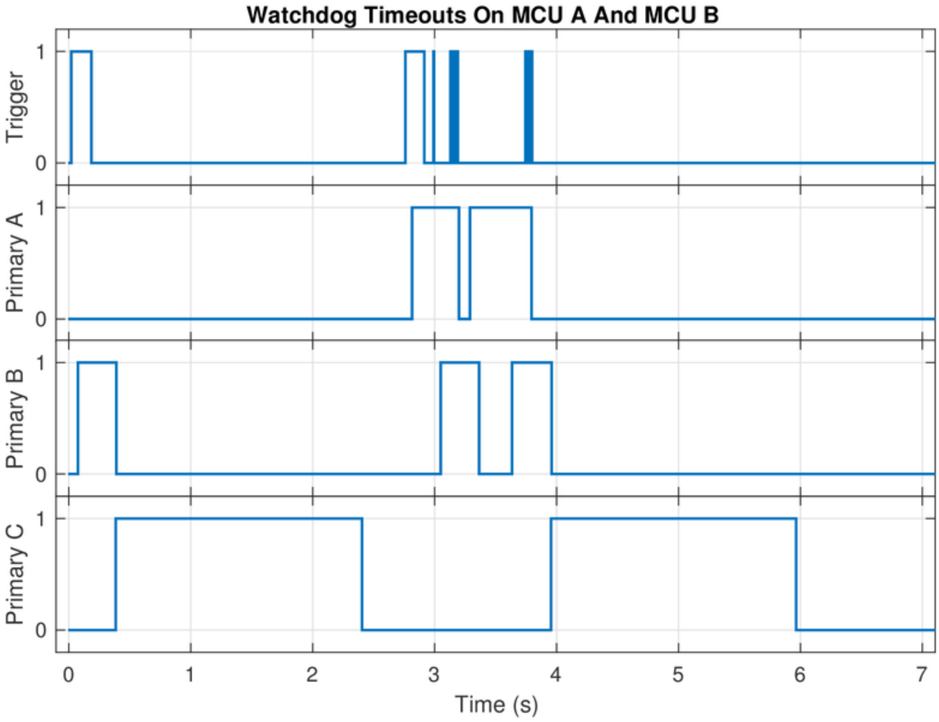


Figure 5.10: Plot of system response to dual introduced watchdog timeouts.

5.7 Discussion

5.7.1 Fixed ID

Each MCU has an ID that is determined by the circuit diagram wiring. By using voltage dividers, intermediate voltage levels can be implemented to minimise the number of pins used for a single use feature. The fixed ID is calculated in the setup function of the Arduino sketch, then not again until the next time the MCU powers on. Each MCU has a non-zero ID. This allows development in the future, to incorporate checking that the input pin does in fact have a supplied voltage level. If reading the pin returns a zero value, the MCU could be programmed to identify this as a fault, and disable the MCU to prevent clashes of IDs. Unless prevented, any such clashes of IDs would lead to conflicting controllers, potentially causing short-circuits and corrupted data.

5.7.2 Processing Flag Operation

Fig. 5.5 shows a zoomed in view of the timing of each MCU, setting its processing output flag. Each MCU waits for a set length of time, in order of ID, to avoid simultaneous

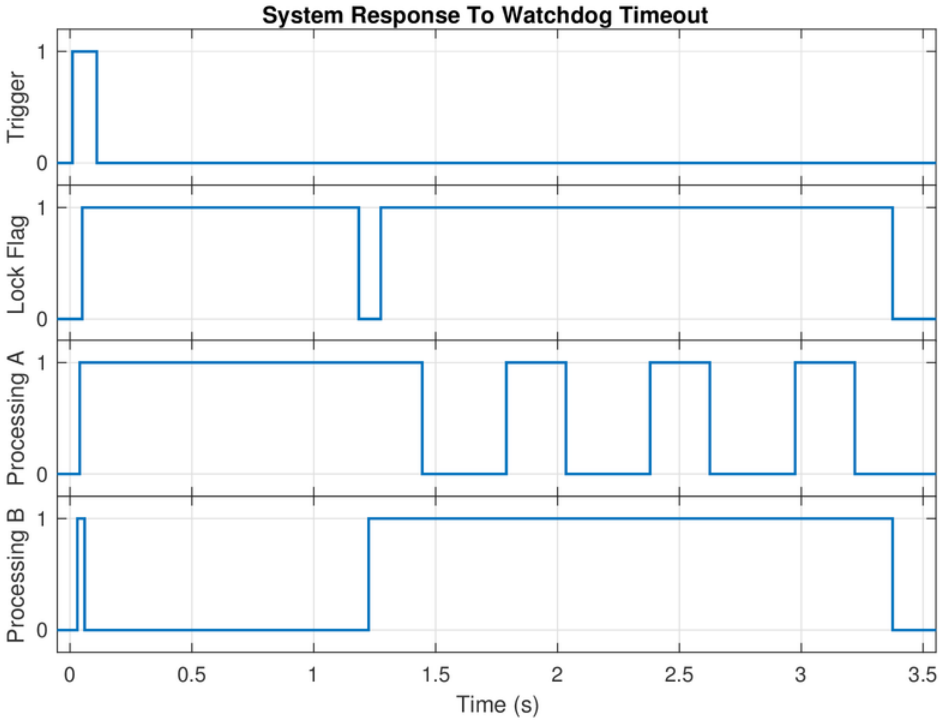


Figure 5.11: Plot of system response to an introduced watchdog timeout.

performance of the system function. Rather than having the MCU with the first ID waiting unnecessarily, each MCU waits for one less scalar period than its actual ID. For these experiments, the scalar period is 30 ms. All three MCUs enable their processing output flags approximately 20 ms after the rising trigger. The processing output flag for MCU A continues, as MCU would have set the lock flag and assumed primary priority. By the time MCU B has waited for its single delay period, 30 ms, the lock flag would have been set, with MCU A holding primary priority. Thus, MCU B clears its processing output flag. Similarly MCU C clears its processing output flag after a double delay period, 60 ms.

The alignment of the setting of the processing flags from each MCU varied in its exact initiation. This can be seen by comparing fig. 5.5 with fig. 5.6. The alignment of the start of processing flag A, compared to the alignment of the start of processing flag B, is vastly different. This is because the majority of the primary assignment Arduino code is in the main loop of the code.

Although the trigger prompts an immediate response from the MCU, using an interrupt pin, the code then has to skip its way to the end of the loop to return back to the start. This skipping forwards is achieved by adding an extra check into each of the if-

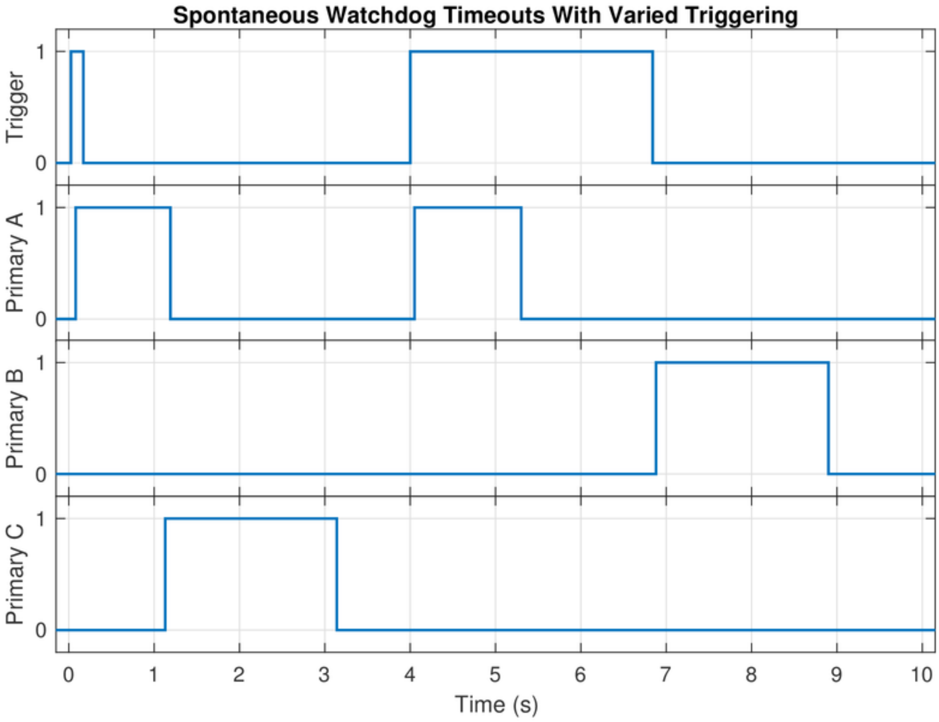


Figure 5.12: Spontaneous WDT timeouts together with varied triggering.

statements throughout the code. Having the assignment of priority within the main loop is necessary to prevent delays from being performed within the interrupt subroutine. It was also necessary for preventing the former primary MCU from retaining control, after it has failed to complete the system function within the expected time.

Although there is some variation in the moment the processing flag is set by each MCU, the scaled delay is sufficient to offset priority assignment, to reflect the order of IDs most of the time. Increasing the delay time scalar would further ensure that there was no conflict of the moment when the MCUs could assume control. However, excessive increase of this delay scalar would cause unnecessary delays for the system. A future development could be to optimise the length of delay, for the effective offset of MCU priority assumption, without unnecessary delays.

5.7.3 System Flag Operation

Fig. 5.6 shows greater detail of the system locking, after the first available MCU assumes primary priority. MCU A does not have a delay to offset from other MCUs trying to gain control. However, as discussed in Section 5.7.2, the process of exiting from the main operation loop to the priority assignment section of the code, can cause slight offsets in timing, as seen in fig. 5.6. With MCU A gaining control first and setting the lock flag, MCU B continues to keep its processing flag output set until completing the system function. By the time MCU B completes its delay from being the second ID MCU, the lock flag is already set, preventing MCU B from taking control.

5.7.4 Varied Triggers

Clearance Triggers

The main system trigger has been incorporated into the clearance flag of the system. At a circuit level, this is achieved by another input for the OR-gate of the clearance flag. At a coding level, this means that either one will cause the interrupt subroutine to be called. Each instance needs to be assessed based on the current state of the code. If it is determined that there is no need for change, i.e. the system is still processing within the expected time, no change is implemented. This is seen in fig. 5.7.

In this way, the trigger input and the clearance flag both serve as a wakeup call to any active primary MCU to assess whether it is actually achieving the desired purpose. Any non-primary MCUs set and clear the clearance flag when they detect that the process is taking longer than expected. This effect can be seen in fig. 5.8.

Repeated Clearance Triggers

The primary MCU sets and clears the clearance flag whenever the WDT experiences a timeout. The results of this reassignment of priority can be seen in figs. 5.9, 5.10, 5.11, and 5.12. In some of the result plots, only the effect of the clearance prompting can be seen, not the spike on the trigger line. This is because any prompting of the clearance flag by MCUs is immediately reset, to avoid unnecessary interruption. The effect of this on the data logging using the laboratory oscilloscope is that sometimes the sampling rate is not fast enough to pick up the very brief activity. Therefore, while the spike itself may not be visible, its effect certainly is evident in the behaviour of the accompanying system signals.

Sustained Triggers

Sustained triggers can be seen in figs. 5.7 and 5.12. The most important consideration in this topic, is that the interrupts on the MCU are programmed to activate for both rising and falling signals. The first sustained trigger in fig. 5.7 is released while MCU A is still processing the system function within the expected time. In this case, a trigger either rising or falling will not affect the system.

The second sustained trigger in fig. 5.7 demonstrates a different situation. Where the trigger remains set until after the primary MCU has completely processed the system function, then a new process is commence once the trigger is released. This can be used to the advantage of some applications. If the system was to trigger at the passing of a car through a driveway, but the car stopped in the driveway, the system would be triggered both at the arrival and the departure of the car.

Rising And Falling Triggers

This feature of processing at the rising and falling of the trigger also has another feature that should be considered for each application. Fig. 5.12 shows the trigger being sustained through the failure of an MCU, due to a WDT timeout. In this case of a failed MCU, the clearance flag was still held high by the input trigger. Therefore, the toggling of the clearance flag by the non-primary MCUs would have no effect. The process would not be toggled again until the trigger was released. This may cause the system to return to the problematic MCU first, before finally reverting to an alternative MCU. The end result is, that the process would eventually be completed, however, a potentially significant delay would be experienced while the system is held high.

This same behaviour was observed in the laboratory with the configuration for overtime processing. If the trigger was held for an extended period, while a primary MCU was taking longer than expected to complete the process, a non-primary MCU could not prompt a priority reassignment. This was not possible when the sustained external trigger was effectively blocking the clearance flag.

An alternative solution would be to separate the trigger and clearance flags onto separate interrupt pins. This alternative would use at least one more pin, consuming already limited pins.

5.7.5 WDT Closing Procedure

WDT Effect on System Flags

When a WDT timeout occurs, it can be set to perform a shutdown function. As shown in fig. 5.11, when the timeout occurs on MCU A, at approximately 1.2 s, MCU A holds its processing flag set for a further 0.25 s. This is the same length of time as the timer length set for the WDT. The WDT closing function also prompts the clearance flag to handover primary priority to another MCU. This function is configured to clear the lock flag and to retain the set processing flag, preparing the system for the next available MCU to assume control. This functionality is clearly visible in fig. 5.9.

Simultaneous Primary MCU Flags

This closing function also maintains the primary flag as set. This activates the additional delay after a reassignment of the primary priority, in turn preventing this MCU from resuming primary operation, however briefly. This also explains why in fig. 5.11, both MCUs A and B have their primary MCU flags set simultaneously just after the handover. Additionally, the introduced WDT timeout was sustained for the remaining duration of the experiment. Therefore, every time the WDT reached its timeout, it would initiate the WDT closing function, which also activates the primary MCU flag. This is not trying to takeover primary priority. It is merely to prevent resumption of primary priority, in the event of a subsequent clearance before it reboots.

Prevention Of System Interference

Once the WDT closing function has been executed, however, the MCU returns to the main loop for whatever time is left of the countdown timer length, 0.25 s or 250 ms. There is a risk with a set primary MCU flag, that this MCU could try to perform some of the system function in that small time. This is alleviated by setting the return-to-start flag. A WDT reset flag is also set in the WDT closing function. When set, this WDT reset flag causes the main loop to return to the start immediately after starting. This has the effect of disabling all interfering influence of the MCU, after the WDT timeout has occurred.

5.7.6 Infinite Sequence Application

The used example code is configured for an infinite progression of steps to complete the system function. For the example, the completion comes when a certain length of time has passed since the commencement of processing. The single function is the enabling of an output LED for a set length of time, 2 s. This doubles as a helpful debugging and demonstration tool. The effect is an LED that is held on for the specified time. For the MCU, though, it is cycling through the main loop including the if-statement for primary MCUs. It will continue this until the conclusion if-statement identifies that the desired time has been reached and decommissions the MCU as the primary.

This configuration should easily be adaptable to other infinite step applications. The key will be in deriving the best way of identifying the conclusion if-statement. This could be, "If a particular signal is received, conclude". The question could be, "If a particular combination of inputs is detected, then conclude". The if-statement may simply remain "If the prescribed time has passed, then conclude". The options for this conclusion if-statement are many and varied. Suitable queries may fit within the category of received information or elapsed time.

5.7.7 Finite Sequence Application

This system can also be adapted to be used as a finite progression of steps. The set steps could be programmed into the main system function section of the code. The conclusion if-statement is again the key. This question could be, "If the new file exists, conclude". It could also be "If the new file has a size within an expected range, conclude". Suitable questions may resemble confirmation of output information.

5.7.8 Sequence Feedback

For situations of outputting particular signals or data, it would be helpful to use some form of feedback, to feed the conclusion if-statement. If outputting a signal, another pin could be used to confirm that the signal was electronically produced. If an error was encountered, such as a broken solder joint or a blown output pin, the MCU may perform the correct code without achieving the desired electronic signal. A feedback loop through a different pin could be used to detect such errors. If the output is produced data, feedback may take the form of checking the files existence, checking the file size, or receiving confirmation notification from another data module that receives the data.

5.7.9 Constant Sequence Application

The system could also be used in applications requiring a constant repetition of a sequence. By programming the conclusion if-statement to prompt the clearance flag, the system would cycle into the next iteration of the sequence. This would also reset the elapsed time count on the non-primary MCUs.

This system would effectively operate like the WDT. MCU A would continue to operate as the primary, resetting the non-primary MCU counters, while it is running as expected. Like the WDT, the primary needs to respond to the non-primary MCUs before the configured timer limit.

5.7.10 Expected Timeframe Predictability

For any of the sequences, an expected timeframe is required for programming the parallel MCUs. This enables the non-primary MCUs to monitor the performance of the designated primary at any one time. This time frame is also used to prevent untimely interference from rapid re-triggering and clearance prompts.

5.7.11 WDT Resetting

From these experiments, the incorporation of a WDT is a useful method of resetting errant MCUs. However, care needs to be taken to include regular WDT resets in the code, to allow the system to continue through its desired sequence. Excessive WDT resetting consumes processing capability, time and power, unnecessarily. Inadequate WDT resetting will not allow the system to function.

Care should be particularly given to applications that include if-statements, and for loops. If all possible progressions are not considered, a particular progression path may be encountered that consumes more time than the WDT allows for, without resetting.

The WDT timeout length should also be considered for preventing unnecessary timeouts. If the timeout is too short, frequent resets are required. However, if the timeout is too long, occurrence of errors won't be identified as quickly, and unnecessary time is wasted on the WDT closing function.

5.7.12 return-to-start Escapes

As with resetting the WDT, frequent inclusion of escapes for the return-to-start feature should be included in any application sequence. This will increase the speed of response to clearance flag prompts. This in turn also serves to prevent possible clashes caused by MCUs simultaneously assuming primary priority. Frequent escapes for the return-to-start function also minimise the interference, that a resetting MCU will carry out after the WDT conclusion function.

5.7.13 Expandability

This system has been designed to be multipliable. An increased number of parallel MCUs would require a matching set of voltage divider inputs for the fixed IDs. Suitable combinations would be required to retain relatively even spacing between the different voltage levels. This would, in turn, reduce the potential for errors in the mapping performed by the Arduino code in the startup function.

The startup function would also need to be edited to suit the number of parallel MCUs being used. Currently, the mapping produces discrete integer IDs for each MCU. Since the fixed ID is only used for staggering the delay, this could be converted to a real number with similar range between each ID.

Additional OR-gates would be required for an increased number of parallel MCUs. This can be achieved using at least two different methods. One simplistic method is to use OR-gates with as many inputs as there are MCUs used in parallel. A more practical and realistic solution would be to connect multiple 2-input or 2-input OR-gates in series, to provide a greater number of inputs. These will be required for the redundancy system. Some OR-gates may also be required for the chosen system function.

The selected number of MCUs to install in parallel would need to be considered. Too few parallel MCUs may pave the way for an inability to recover from errors. Excessive numbers of parallel MCUs adds cost to the system, and complication to the production of the system. These considerations will need to be traded-off, to determine an ideal quantity.

5.7.14 Pin Assignment

The pin assignment for this system has mainly used the analogue pins of the MCUs for the redundancy system. This retains the majority of the digital pins for use in performing the system function. However, except for the fixed ID pin, the redundancy system pins are all handled as digital pins. This is achieved using the Arduino functions digitalRead() and digitalWrite(). Therefore, these signals can be easily reassigned to digital pins, in order to leave the analogue pins available for use by the system function.

5.7.15 Multiple MCU Output Management

Any time there is an output required from the system for the system function, a funnel method will be required for giving all MCUs access while preventing short circuits. This can be achieved using OR-gates for low power applications. OR-gates have a current limit on their outputs. If this current limit prevents use for a particular application, alternatives are available.

Relays, transistors and optocouplers can be used to facilitate the necessary isolation. However, each of these methods require at least one extra control pin to be used for enabling and disabling the isolation component. A simple alternative would be to use a diode on each output, preventing reverse current flow. Any of these alternatives can potentially be used, but the inherent voltage drop for each of these systems will need to be considered for any application of the alternatives.

Chapter 6

Self-Supervised Redundant Camera Device

6.1 Introduction

This chapter presents the application of the self-supervised redundant system of chapter 5 to a camera monitoring device. This device is intended to capture an image after external triggering. This image is to be saved onto an SD card, then emailed to a specific email address.

The circuit developement is shown in section 6.2. Section 6.3 details the code development for this application. The experimentation is laid out in section 6.4. The results are presented in section 6.5. The application and the results are discussed in section 6.6.

Components were selected based on their ability to operate with the Arduino compatible MCUs. A camera module, an SD card module, and a cellular access module were required to achieve the desired outcomes.

6.1.1 Camera Module

The Adafruit TTL Serial camera was selected for this system. This camera can be focused to a maximum distance of 15 m. It uses an input voltage of 5 V and uses a 3.3 V logic level voltage. Both of these are within the voltage limits of an Atmega328 IC. The availability of an Arduino library was also a significant factor in the selection of this camera.

6.1.2 SD Card Module

A Catalex micro SD card module was selected for this system. This module is readily available and is compatible with Arduino capable MCUs. The Arduino IDE includes a library for using with these modules.

6.1.3 3G Cellular Module

The Adafruit FONA 3G module was selected for the cellular access module. This device was the only 3G access module found that would work with Arduino compatible MCUs. The datasheet for the included cellular access IC claims to be able to access the data capabilities of a 3G cellular connection [9]. Adafruit provide a library for this module for use in the Arduino IDE, however, with limited cellular data integration. This module was selected with the intention of further developing this library's capability.

6.1.4 Development Plan

This system was developed in three stages. The first stage was to confirm that the camera and SD card modules would work in an Arduino configuration. The second stage was to develop and test the functionality of the 3G cellular access module. The third stage would be to integrate these two circuits along with the redundant configuration into the final redundant camera system.

6.2 Circuit Development

6.2.1 Pin Assignment

The Atmega328 pin assignment for this project is listed in Table 6.1. This application of the self-supervised redundancy system only requires digital pins, so no modifications were required to the pin assignment used in the development of the redundancy system for the system wiring.

6.2.2 Camera And SD Card Proof Of Concept

Appendix B.5 shows the circuit diagram for the proof of concept circuit for the camera and SD card modules. The camera and SD card modules are wired to the Atmega328 as specified in Table 6.1. A 16 MHz crystal oscillator provides the clock input along with its required 22 pF capacitors. A NO PB provides an input trigger on the digital pin 7, incorporating a 15 kΩ pull-down resistor. The whole system is powered from a 5 V supply.

Because the camera uses a 3.3 V logic level voltage, a voltage divider is required to reduce the 5 V output of the Atmega328 down under 3.3 V. Using two 10 kΩ resistors reduces the signal amplitude down to 2.5 V. While being reduced below the maximum input level for the camera, it will still be greater than half of that level, 1.65 V, so it will still be enough to trigger high inputs for the camera. Similarly, the 3.3 V output of the camera is greater than half of the 5 V logic level voltage of the Atmega328, so the camera output signal will still be received by the Atmega328.

Table 6.1: Atmega328 pin assignment for self-supervised redundant camera system.

Pin	Function	Assignment	Pin	Function	Assignment
1	Reset		28	A5	
2	D0/Rx		27	A4	Lock Flag In
3	D1/Tx		26	A3	Processing Flag Out
4	D2	Clear Flag Interrupt	25	A2	Lock Flag Out
5	D3	Processing Flag In	24	A1	Clear Flag Out
6	D4	Cellular Reset	23	A0	Fixed ID In
7	Vcc	Vcc	22	Gnd	Gnd
8	Gnd	Gnd	21	ARef	5V
9	Osc1	Clock Input	20	AVcc	
10	Osc2	Clock Input	19	D13	SD Card Clock
11	D5	Cellular MISO Rx	18	D12	SD Card MISO
12	D6	Cellular MOSI Tx	17	D11	SD Card MOSI
13	D7	Camera MISO Rx	16	D10	SD Chip Select
14	D8	Camera MOSI Tx	15	D9	Primary MCU LED

6.2.3 Cellular Proof Of Concept

The circuit diagram for the proof of concept configuration of the FONA 3G module is detailed in Appendix B.6. Communication in this configuration relies on a computer connected to the circuit, using the circuit in fig. 3.4. This circuit is therefore simplified to one Atmega328, the FONA 3G module, the clock input as the main circuit components. A 5 V provides power to the main circuit. The FONA 3G module also requires its own lithium polymer (LiPo) battery for steady operation, as included in the circuit diagram.

This module is wired according to the example Arduino code provided by Adafruit. The pin assignment does not therefore match that of the planned final circuit. Since only wiring between the Atmega328 and the FONA 3G module are wires for a reset line and the two lines used for serial, these can be easily moved to other digital pins on the Atmega328. This is possible since any of the digital pins on an Atmega328 can be used for a software serial connection.

6.2.4 Redundancy Integration

The circuit diagram for the integrated redundant system is shown in Appendix B.7. Since cellular data access was not accomplished through the FONA 3G module, the final configuration excludes the FONA 3G module from the circuit diagram. Future inclusion would be in the same manner as the inclusion of the camera and SD card modules.

The given circuit diagram omits several portions of the circuit for clarity. The clock input circuit as shown in fig. 3.3 is omitted from each of the MCUs. $15\,\mathrm{k\Omega}$ pull-down resistors on each of the OR-gate inputs and MCU inputs have also been omitted. Finally, the flag wiring for the self-supervised redundancy system has been omitted. This wiring should be included as shown in Appendix B.4.

The inputs and outputs of the camera and SD card modules are connected in much the same way as in the proof of concept circuit. The main difference is that all MCU outputs to the modules divert through OR-gates in order to avoid short-circuits. All module outputs branch directly to the MCU inputs.

6.3 Code Development

6.3.1 Camera And SD Card Proof Of Concept

Appendix C.7 shows the Arduino code for the camera and SD card proof of concept. This code sketch is a modified version of the snapshot example that comes with the Adafruit library for the camera. This configuration initialises the camera and SD card modules, then waits for the trigger PB to be pressed. Once the system is triggered, the next sequential filename is calculated, then the captured data is read from the camera module and written to the SD card module, at a rate of 32 bytes at a time. The time taken to process the image is recorded and displayed on the serial monitor after the image processing is completed.

Camera Integration

Integration of the TTL Serial camera requires the inclusion of the several additional Arduino libraries. The Adafruit_VC0706 library includes necessary camera function definitions. The SPI library is required for the communication between the MCU and the camera. Lastly, the software serial library is included to use a serial connection on pins other than the standard Atmega328 serial pins, digital pins 0 and 1. The camera is initialised in the setup loop of the code using a boolean query.

After initialisation, the camera captures an image in a moment when requested. This request takes the form of the boolean query cam.takePicture(). Once this line has been executed, the camera will hold the data of that image until it is instructed to revert back to video mode, even if the takePicture() function is called again. This means that the image data can be accessed as required. Once all the image data has been recorded to the desired destination, the camera is returned to video mode with the command cam.resumeVideo().

This clears the image data from the camera and prepares the camera to take the next image.

One of the system requirements is for the system to capture an image within 500 ms. This proof of concept tests the response time of these components. The time is recorded at the time of triggering already to show the duration of processing. The code compares this timestamp with the current time once the camera has confirmed the captured image. Using this process, the elapsed response time is measured.

Micro SD Card Integration

The SD card module also requires additional libraries for use in this configuration. The SPI library is required for communication with this module. The SD library is required for handling the pin connections and the necessary commands for initialising the card and also reading and writing data on the card. The pin for the chip select signal to the card is also required to be specified. The default pin, digital pin 10, is used in this configuration.

After the SD card connection has been initialised in the setup function, the system can read and write data on the card. First a file name needs to be selected. This configuration selects the next sequential file name to avoid writing over existing data, checking the existence of previous combinations. Once a file name has been selected, a new file is created and opened on the SD card in one command, SD.open(filename, FILE_WRITE). While the retrieved data is collected from the camera 32 bytes per cycle, the data is written to the SD card file at the same rate. Once all of the data has been written, the image file is closed, to prevent further writing.

6.3.2 Cellular Proof of Concept

The Arduino code used for this proof of concept is shown in Appendix C.8. This code is a simplification of the FONAtest example code that comes with the FONA library. Features relevant to GPS, an FM radio, network time, and audio control have been eliminated from this sketch. Some of these features aren't available on the selected module. Some of the features are unnecessary for this application. This elimination also helps to reduce the required memory for future integration into the redundant system.

This code presents a range of available commands accessible through the serial monitor of the Arduino IDE on a connected computer. These commands activate features such as unlocking the SIM card, sending an SMS, making a phonecall, and controlling the GPRS. These brief commands sent from the serial monitor to the MCU then execute the necessary sections of the library files to accomplish the intended action.

For this configuration, the required steps for unlocking the SIM pin have been condensed into a custom function that is called in the setup function. This is to simplify the process of experimentation. An arbitrary pin number has been used for the given configuration.

To facilitate access to GPRS using the FONA module, the code requires an access point name (APN) to be specified. While this enables GPRS, it may not be sufficient for

3G data access.

Library Modification

Modifications were made to the Adafruit FONA library files in an attempt to facilitate email access. An additional command, boolean sendTestMail(void), was added to the h-file. This can be seen at line 173 of Appendix C.9. This connects the function call from the programmed sketch to the detailed function in the cpp-file. Lines 804 to 830 were added to the cpp-file, included in Appendix C.10. This function prepares the necessary header details for an email.

The pre-filled details for the email are sent using commands specified in the datahseet for the SIMCom IC on the Adafruit 3G module [9]. Email specifications include the SMTP server address, SMTP account, SMTP password, sender address, recipient address, subject, and body. The last included command is for the FONA module to send the email. Each of the commands is checked by the code for completion before proceeding to the next command.

6.3.3 Redundancy Integration

The integration of the code for the camera system components into the self-supervising redundant system is listed in Appendix C.11. Since 3G data access was not accomplished this code omits provision for the FONA 3G module.

The resultant code is constructed by fitting the camera and SD card code from section 6.3.1 into the redundant system code from section 5.4. The proof of concept code is divided three ways for this integration. The declarations of libraries, pins, and variables is added prior to the setup function. The necessary initialisation steps are added into the setup function. Lastly, the main function of the proof of concept is added to the main loop of the redundant code in the process for the primary MCU.

The main function of the proof of concept is modified to suit the redundant system. The trigger button of the concept system is eliminated, instead using the existing trigger capability of the self-supervised redundant system. This means that the main system function of the redundant code now captures the image using the camera and writes that data to the SD card, 32 bytes at a time.

The second required modification changes the conclusion if-statement of the redundant system. Two options are presented for the conclusion if-statement. If the process completed in less than half the expected time, then there is most likely an error in the system, such as the SD card cannot be found. Therefore, if the conclusion is reached too quickly, the clearance flag is set to prompt a hand over to another available MCU. If the function conclusion was not reached too quickly, the primary MCU decommissions itself as expected.

6.4 Experimentation

6.4.1 Camera And SD Card Proof Of Concept

The experimental circuit was configured using an Arduio Nano version 3. The clock circuit was subsequently omitted. The 5 V system supply was provided by the Nano. All other wiring was connected as specified in Appendix B.5. 4 GB SanDisk micro SD cards were used for data storage.

Several images were captured by pressing the trigger button. The serial monitor signalled the completion of the image capture. After several images were captured, the SD card was connected to a computer to view the captured images. This process was repeated several times for the three different size options.

The response time was recorded for twenty consecutive images. The average of these response times was taken as the response time of this proof of concept system. This data was gathered using readouts on the serial monitor in the Arduino IDE.

6.4.2 Cellular Proof Of Concept

This circuit was implemented using an Arduino Nano version 3. The 5 V supply was taken from the Nano. The clock circuit was omitted as it was unnecessary. All other wiring was configured as per the circuit diagram in Appendix B.6, including a 2200 mA h LiPo battery.

The phonecall functionality of the FONA 3G cellular module was tested using the serial monitor access to its functions. This was achieved by entering the character "c" into the serial monitor to make the phone call. A headset was connected directly to the FONA 3G module for audio access. To end the phonecall, "h" was entered into the serial monitor.

Similarly, SMS messages were sent, viewed, and managed through the serial monitor. The available SMS commands listed in Appendix C.8 were tested, including read individual message, read all messages, delete individual message, and send message. The total number of messages was also viewed.

After setting the APN for the relevant network provider, GPRS was connected. GPRS was connected by entering "G" into the serial monitor. This command returned a positive confirmation when it connected. The connection was disconnected by entering "g" into the serial monitor. This also returned an affirmation after completion.

6.4.3 Email Sending

The library files were configured with static information to test the sending of an email. Relevant configurations for two separate email addresses were entered into the sections of code for the custom mail sending function in the library cpp-file. This function was based on the other functions in the library, and on the commands listed in the datasheet for the SIMCom IC [9]. This function was configured to be executed after entering "J" into the serial monitor. An additional attempt was made to send an email without a subject or body.

6.4.4 Redundancy Integration

Fig. 6.1 shows the experimental circuit used for testing the application of the camera and SD card in a redundancy system. The main redundant system used the same configuration as used in section 5.5. Triple 3-input OR-gate DIP ICs, model CD74HC4075E, from Texas Instruments were used for all additional OR-gates for the camera and SD card modules.

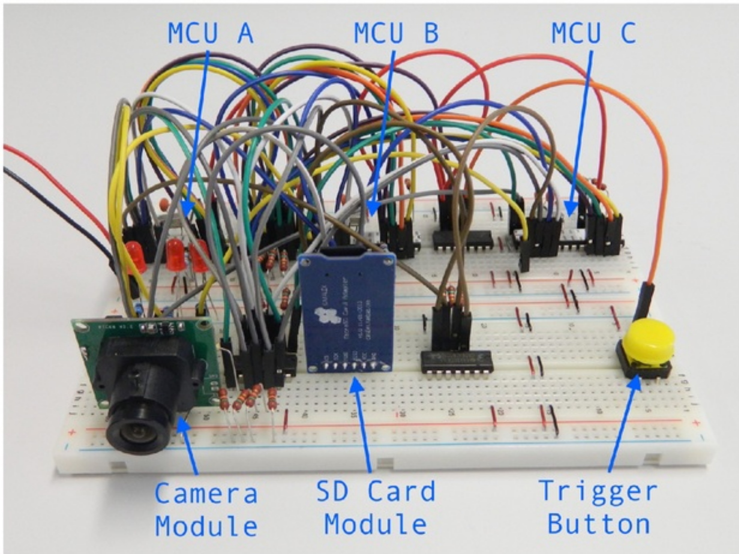


Figure 6.1: Experimental configuration of camera and SD card self-supervised redundancy system.

The system trigger was humanly activated while monitoring outputs on the serial monitor of the Arduino IDE. Several attempts were made to capture an image. The SD cards were then connected directly to a computer to view the captured images.

6.5 Results

6.5.1 Camera And SD Card Proof Of Concept

Processing Duration

Several photos were successfully captured using the specified configuration. The average specifications of the photos are listed in Table 6.2. The larger the photo size, the longer the system took to process. This however is offset by the desired image quality. A large photo is shown in fig. 6.2. An example of the serial monitor output is shown in fig. 6.3, as used for gathering the results.

Table 6.2: Average specifications of available image sizes from the Adafruit TTL Serial camera.

Size	Area	Memory	Transfer Time
Small	160 x 120 px	3 kB	1.5 s
Medium	320 x 240 px	12 kB	6.5 s
Large	640 x 480 px	48 kB	25 s

Figure 6.2: Experimental example of the largest resoltion image from the Adafruit TTL Serial Camera, 640 x 480 px.

Reponse Time

The average response time for the proof of concept camera system was 13.85 ms. This was the average of twenty samples taken from the moment the button was pushed to the moment the camera responded to the MCU that the image had been captured in the camera's memory. There was one outlier value of 23 ms. Excluding this outlier, the

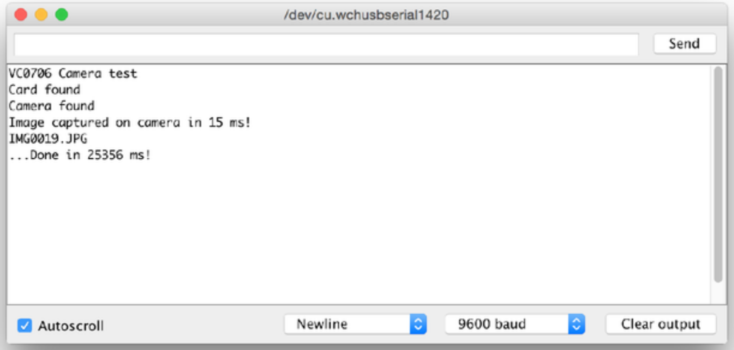


Figure 6.3: A screenshot of the Arduino IDE serial monitor used for capturing results of reponse time and processing duration.

average response time for the remaining nineteen samples was 13.37 ms, with a minimum of 13 ms, and a maximum of 15 ms. An example of the completion time can be seen in fig. 6.3.

6.5.2 Cellular Proof Of Concept

Multiple outgoing phonecalls were successfully made, although they were mono-directional. The receiver of the phonecall could not hear the voice of the caller. However, the caller could hear the receiver's voice.

Multiple SMS messages were exchanged using the serial monitor for a user interface. Incoming messages were received, and viewed in bulk. Some were deleted. Outgoing messages were likewise received as expected.

The GPRS connection returned positive affirmation of a data connection, but emails were not successfully sent. When the email sending process was initiated, the various steps were displayed in the serial monitor. This indicated issues with setting the text of the subject and body.

After clearing the subject and body, sending another email was attempted. This progressed through the whole process returning a positive affirmation that the email had been sent in the last step. However, no emails were received at their intended destination.

6.5.3 Redundancy Integration

The serial monitor was viewed while triggering the redundant camera system. The system kept returning errors when trying to connect to the SD card module. Attempts were made to relocate the initialisation portions of the code to the main loop without success. The result was that the system corrupted the formatting of three different micro SD cards. Even after they were formatted, the system did not capture any images using this configuration.

6.6 Discussion

6.6.1 Image Capture Duration

The camera and SD card worked as expected using the basic proof of concept configuration. The greater the image size, the longer the processing time. Because this consumes the whole attention of the system while processing, The system can't capture another image until the first is processed completely. Any applications for this configuration will require a trade-off between the desired image quality and the available frequency of image capture. This processing time frame is related to the speed capability of SPI.

6.6.2 Image Capture Response Time

The response time of the camera configuration is well within the specification for the project. Even the outlier is well within the desired time. Integration into the redundancy system may extend this response time, allowing for the system to select a primary MCU. However, fig. 5.6 shows that the redundancy system can establish a primary MCU in well under 100 ms. Therefore, a final configuration using these modules should have no problem achieving a system response time under the specified 500 ms.

6.6.3 Cellular Proof Of Concept

The cullular access system has not reached its intended functionality, and will require further research and development. This development should further determine the lines between 2G and 3G data access. Also, the datasheet for the SIMCom IC will need to be studied in greater detail to derive a suitable sequence to access its capabilities.

While the FONA 3G module has a thorough datasheet for the available commands, the implementation of these commands can be difficult with limited feedback. This is particularly an issue when dealing with a third party service provider, the relevant cellular carrier. With outside influence, compared to a project that operates purely within a laboratory, identifying the gaps in information flow can be difficult to troubleshoot. The realisation of this data access may require considerable time and effort.

6.6.4 Redundancy Integration

The redundant parallel MCU-based camera system also has not achieved the planned function. While the required code sections fitted within the anticipated code gaps. The overall process neglected critical requirements. Not only does the SD card module require only one MCU to be writing to it at one time. It seems, also, that only one MCU can initialise the connection with the SD card module at a time. Without considering this requirement, this experimentation suggests that no data capture will be possible. Greater research and development will be required for this system to become a workable solution.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

7.1.1 Supervised Redundancy Model

Capability

A separately supervised redundancy system has been developed, using open-source microcontroller units (MCUs). This system uses one MCU to monitor the performance of two parallel MCUs performing the main system function. The experimental system was setup to process analogue inputs, driving both digital and analogue outputs.

Benefits

This system has the benefit of rapid transition between primary MCUs. This reduces the impact experienced by the outputs from system errors. For some applications that are not time dependent, this may provide a close approximation of the desired system function.

Another benefit of this system is the simplicity of the code. For basic applications the parallel MCUs do not require elaborate coding to perform their function. Additionally, the code for the supervisor MCU is largely based on the code of the parallel MCUs.

This system also has the benefit of error detection. By measuring the final system outputs, errors are quickly identified.

7.1.2 Limitations

Although errors are identified quickly, the source of the errors is not necessarily identified. To identify which MCU experienced the error, all output pins of all supervised MCUs would need to be monitored. This is not possible with many outputs due to the limit of pins on the MCU.

If the source of errors is not detected, then the system blindly switches between primary MCUs, in an attempt to produce the expected signal. This means that switching will happen even if there's not a functional MCU being enabled, for instance if one of the MCUs has become completely unresponsive.

If a greater number of MCUs is added in parallel, to increase the tolerance of errors, then there is nothing to stop unresponsive MCUs from being assigned primary control. Thus, a greater number of MCUs would not increase the tolerance of errors, but compound the effect of errors, particularly when multiple errors are experienced.

7.1.3 Self-Supervised Redundancy Model

Capability

A dynamic redundancy system has been developed that uses the parallel MCUs to monitor each other. This expandable system can tolerate unresponsive MCUs, reassigning primary control to an available responsive MCU. This primary control can be assigned at the time of external triggering, or in the middle of a process, when the existing primary MCU has become unresponsive. This system is suited to applications requiring a finite sequence of steps as well as an infinite sequence of steps.

Advantages Over Separate Supervision

The first benefit of this system, over a separately supervised system, is that the required peripheral control hardware is reduced. This circuit only requires three OR-gates to function in its simplest configuration. This is contrasted with a separately supervised system, requiring multiple types of gates for a hardware priority selector, or a whole other MCU to serve as the supervisor.

The second benefit of this self-supervised redundancy system, over a separately supervised redundancy system, is that of single code development. Where all MCUs are parallel, performing the same processes, and have their positions identified with hardware, they can all be loaded with the same set of code. A separate MCU supervisor would require a whole separate set of code.

Jammed MCU Recovery

Through the use of a watchdog timer (WDT), this system can recover from situations where the MCU has become jammed in a process, and is unresponsive. In the event of the primary MCU jamming, the primary control will be passed to the next available MCU. The WDT will then reset the jammed MCU, in an attempt to recover back to full operational capability.

System Requirements

The Arduino code for the main function requires elements of code to continue performing, and to maximise its efficiency. Periodic resets of the WDT are required to prevent the MCU from being reset. This serves to alert the WDT that the MCU is still responsive.

Additionally, a return-to-start flag will need to be checked regularly during the main system function, in order to enable a rapid hand over in the event of a jammed MCU.

The wiring requires both OR-gates and pull-down resistors. The OR-gates are required for all outputs. This allows access to peripheral components from all MCUs, while preventing short-circuits being caused in the process. The pull-down resistors should be fitted to all inputs of MCUs and OR-gates, to minimise the effect of any dislodged wires, particularly for the redundancy wiring.

Limitations

There is currently no inclusion of immediate output monitoring. This would have to be developed with future applications. This would require an MCU to be discounted from primary assignment if it is not producing the expected results.

7.1.4 Resetting

The first method used for facilitating MCU resets, in the event of function errors, was using the supervising system. If a primary MCU was found to not operate properly, then the offending MCU would be indiscriminately reset by lowering the signal to the reset pin of that MCU. This can cause extensive unnecessary downtime for MCUs if the error is merely a wiring error. Also, an additional output pin is required in order to individually reset each parallel supervised MCU.

The watchdog method, by contrast, is simpler to implement, and more efficient than the previous method. An onboard WDT listens for regular activity from each MCU. No additional wiring is required. The necessary hardware is built into Atmega328 ICs. The code is easily included, provided regular resets of the WDT are included. This method eliminates unnecessary resets due to wiring errors. This method is also independent of supervision methods.

7.1.5 Camera System Redundancy Application

The camera monitoring device was not successfully implemented into a redundant MCU system. Also, since the system was not successfully developed, a working prototype was not produced. Several limitations prevented this configuration from being implemented. This project did not achieve email capability through the cellular module. Nor was the camera and SD card able to be successfully integrated into the code and wiring for the redundancy system.

Additionally, implementation of the coding for the camera and SD card modules cannot be used in the redundancy system, in the same way as a single MCU system. This limitation is due to the inability for multiple MCUs to be connected to the modules at once. Further research is required to include the connection initialisations into the main loop of the system code. This would then allow the modules to be enabled only when needed.

Even if all of these components was made to work, the system would still be limited by the processing duration of the images. At the maximum image resolution of the camera, the system will have a 25 s gap, before it can capture another image. The gap can be reduced at a cost of image quality, but this may similarly limit the application of this camera system.

7.2 Future Work

7.2.1 Comprehensive ID Voltage Dividers

A future development could incorporate a voltage dividers for all ID wiring inputs. A non-zero feature was incorporated for this system. The same method could be used at the top end of the ID range to prevent a full value. If one of the resistors in a voltage divider became dislodged, the input would produce either a zero value, or a full system voltage value. This would be avoided if all ID inputs incorporated a voltage divider, neither using purely the ground or system voltage.

The MCUs would then be coded to detect both zero voltage ID inputs, and full system voltage ID inputs. Any instances of detection of this error could be used to self-disqualify the particular MCU from participating in the system's operation.

7.2.2 SPI Refinement

The issue of multiple MCUs initialising the camera and SD modules in their setup functions, is connected with the SPI protocol. In this protocol, the master device sets its card select pin to high, also known as the slave select pin. This identifies the master. However, if multiple MCUs do this at once, they are all identifying as the SPI master. The SD card may not distinguish between them sufficiently well.

Research is required into the fundamental behaviour and requirements of the SPI protocol. Perhaps then, the interface can be enabled as required, then disabled until required next. In this way, their functionality may be able to be controlled. This approach may extend the response time of the camera.

The transfer of data using SPI may also be increased. Further research into the possible data speeds may reduce the transfer time required for each image. This would help to reduce the tradeoff required between speed and image quality.

7.2.3 Selective Powered Module

Another method of controlling the functionality of the SD card and camera modules may be the controlled disablement of their input voltage pins. By restricting the power supply pins of the modules, they could be enabled as required. The connection would then be initialised after they have powered up. When the function has been completed, the modules could then be powered down by the primary controller. This control could be implemented using optocouplers, transistors or similar isolators.

7.2.4 Cellular Integration

Further research is required to make use of the expected email capability of the FONA 3G module. Such development will require custom functions to be added to the library files. The necessary activation sequence first needs to be identified, for establishing the cellular data connection.

7.2.5 Soldered Prototype

A soldered prototype was not completed for the self-supervised camera device. This will have to be developed after the working circuit is successfully configured.

7.2.6 Optimise Timing Values

A future development of the self-supervised redundancy system, is to optimise the required delay times, when the primary role is assigned to one MCU. If the delay times are too short, there is a risk of multiple MCUs assuming simultaneous control, corrupting data. However, if the delay time is too long, unnecessary time is wasted, increasing the gap experienced at the hand over between MCUs.

7.2.7 Error Identification

Further development is also required to identify a suitable method of diagnosing signal faults. A system could respond to errors more accurately if the source of the error was clearly known. This is particularly relevant to the self-supervised system, as it has no signal error detection.

Error detection for the self-supervised system could directly be worked into the code. If an error can be linked to a particular MCU, then that MCU could set a software flag, to prevent taking on the primary role. Such a flag may be best implemented as a delay, so that if no other MCUs are available, then limited control can still be taken on.

A possible path of research for greater fault diagnosis could make use of a shift register. This may be particularly useful for binary signals rather than PWM signals and high speed serial signals. The use of a hardware shift register may improve the process of identifying the primary microcontroller. This, however, may be limited to binary inputs and outputs, excluding time-dependent signals such, as PWM inputs and outputs, and serial data.

Chapter 8

Abbreviations

2G	second generation cellular telecommunications platform
3G	third generation cellular telecommunications platform
ADC	analogue to digital converter
APN	access point name
DIP	dual inline package
GPRS	general packet radio service
IC	integrated circuit
ID	identification
IDE	integrated development environment
ISP	in-system programming
ISR	interrupt subroutine
LED	light emitting diode
LSB	least significant bit
MCU	microcontroller unit
MIMO	multiple input, multiple output
MISO	multiple input, single output
MISO	master input, slave output
MOSI	master output, slave input
MSB	most significant bit
NC	normally closed
NO	normally open
PB	pushbutton
PCB	printed circuit board
pot	potentiometer
PWM	pulse width modulation
Rx	receive
SD	secure digital
SIM	subscriber identity module
SIMO	single input, multiple output
SISO	single input, single output

SMS	short messaging service
SMTP	simple mail transfer protocol
SPI	serial peripheral interface
Tx	transmit
WDT	watchdog timer
WDTCSR	watchdog timer control register

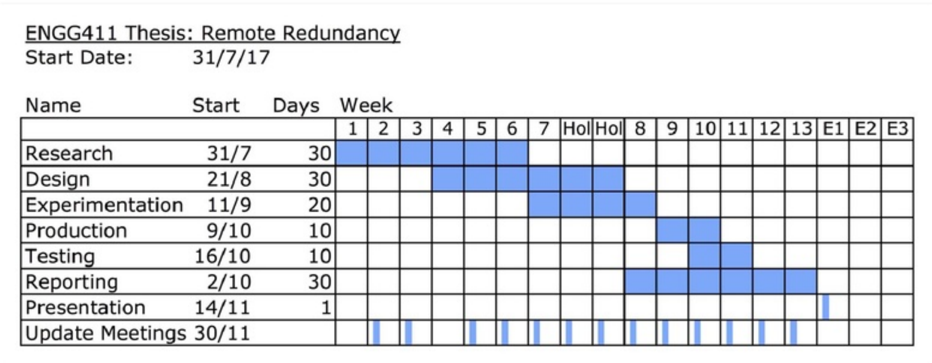
Appendix A

Project Plan and Attendance Form

A.1 Overview

Section A.2 sets out the overall timeline for the project, laid out in a Gantt Chart. The attendance form for consultation meetings is shown in section A.3.

A.2 Project Plan



ENGG411 Thesis: Remote Redundancy

Start Date: 31/7/17

Name	Start	Days	Week																	
			1	2	3	4	5	6	7	Hol	Hol	8	9	10	11	12	13	E1	E2	E3
Research	31/7	30																		
Design	21/8	30																		
Experimentation	11/9	20																		
Production	9/10	10																		
Testing	16/10	10																		
Reporting	2/10	30																		
Presentation	14/11	1																		
Update Meetings	30/11																			

A.3 Consultation Meetings Attendance Form

Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
2	8/8/17	Status update & requirements for next week	[illegible]	[illegible]
3	15/8/17	Discuss required results plots formatting.	[illegible]	[illegible]
5	29/8/17	Discuss ordering parts & reimbursement, & Need for details of progress	[illegible]	[illegible]
7	12/9/17	In person. Status update	[illegible]	[illegible]
Holidays	26/9/17	Group updates presentation	[illegible]	[illegible]
8	3/10/17	Group updates presentation	[illegible]	[illegible]
9	10/10/17	Group updates presentation	[illegible]	[illegible]
10	17/10/17	Group updates presentation	[illegible]	[illegible]
11	25/10/17	Group updates presentation	[illegible]	[illegible]
11	26/10/17	Thesis progression & questions	[illegible]	[illegible]
12	1/11/17	Skype: progress & deadlines	[illegible]	[illegible]
14	14/11/17	Skype: Format & revision	[illegible]	[illegible]

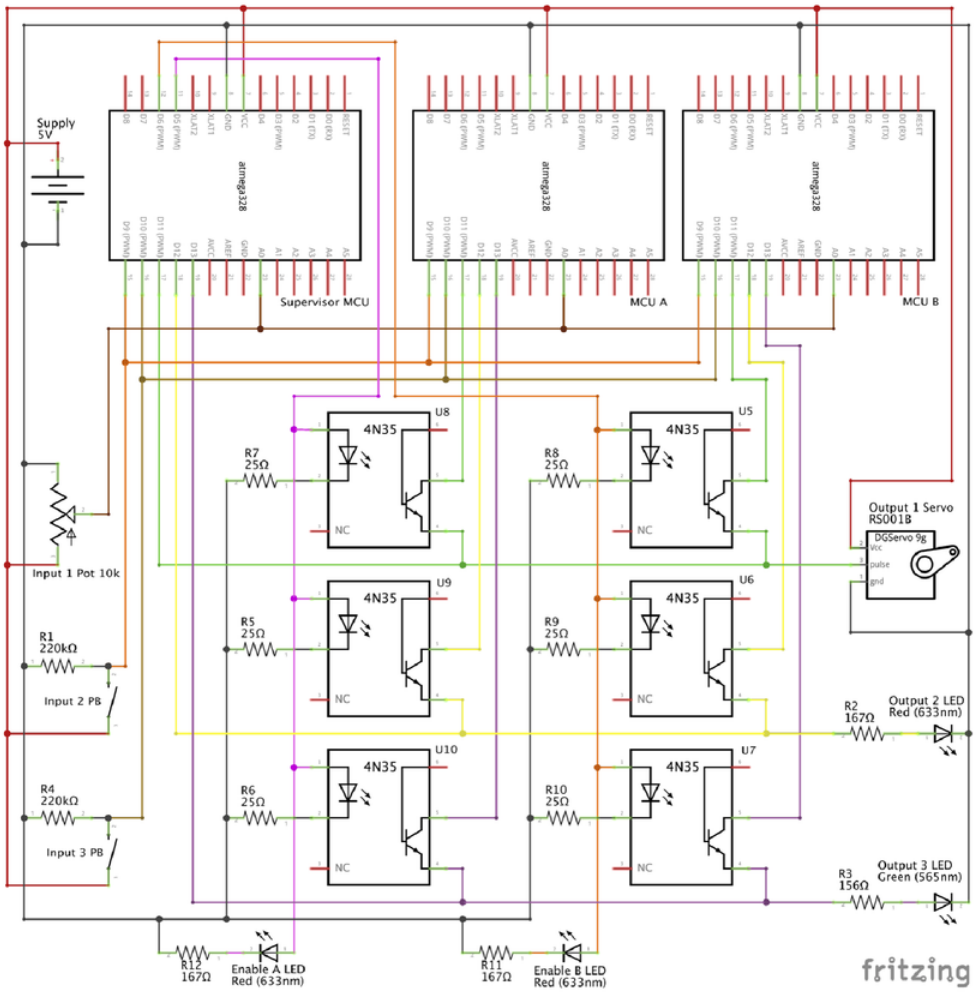
Appendix B

Circuit Diagrams

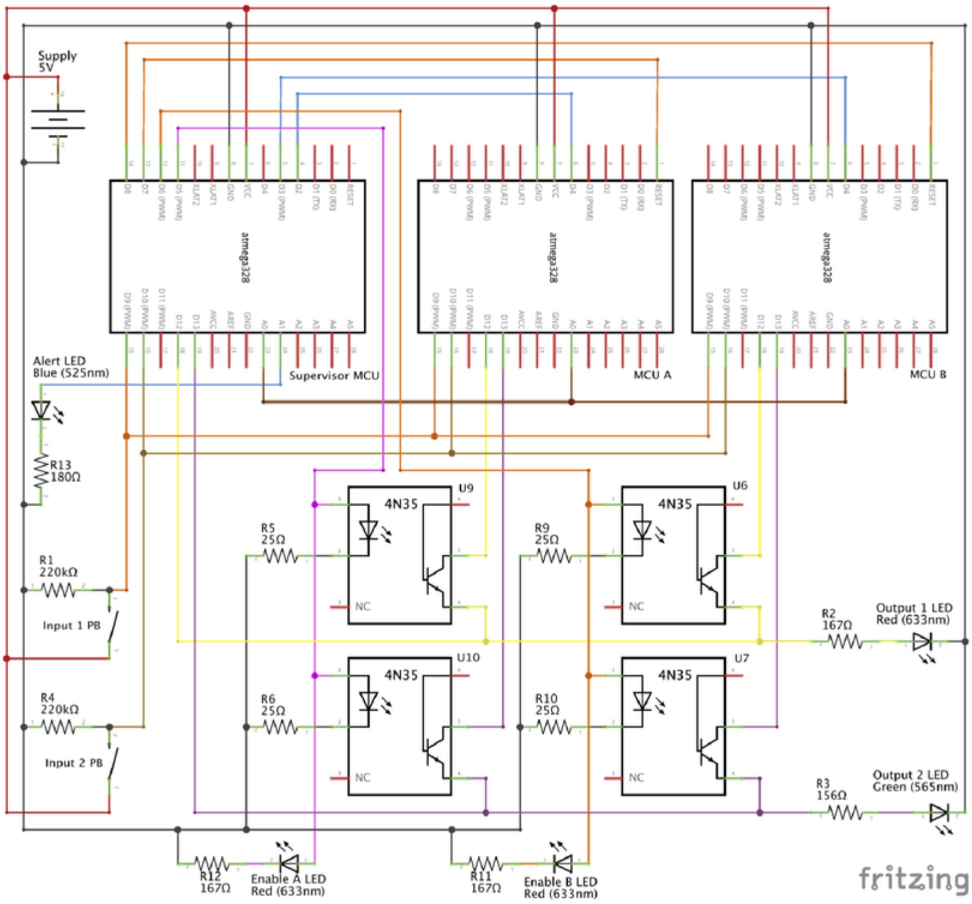
B.1 Overview

This appendix shows the circuit diagrams developed for this project. Section B.2 shows the circuit of a supervised parallel redundancy circuit. Section B.3 shows the diagram of a supervised parallel redundnacy circuit with incorporated resetting capability. The circuit for a self-supervised parallel redundancy circuit is shown in section B.4. A proof of concept circuit for a camera and SD card system is shown in section B.5. The diagram for a cellular module proof of concept is shown in section B.6. Section B.7 shows the circuit designed for an implmentation of the camera and SD modules into the self-supervised redundancy system.

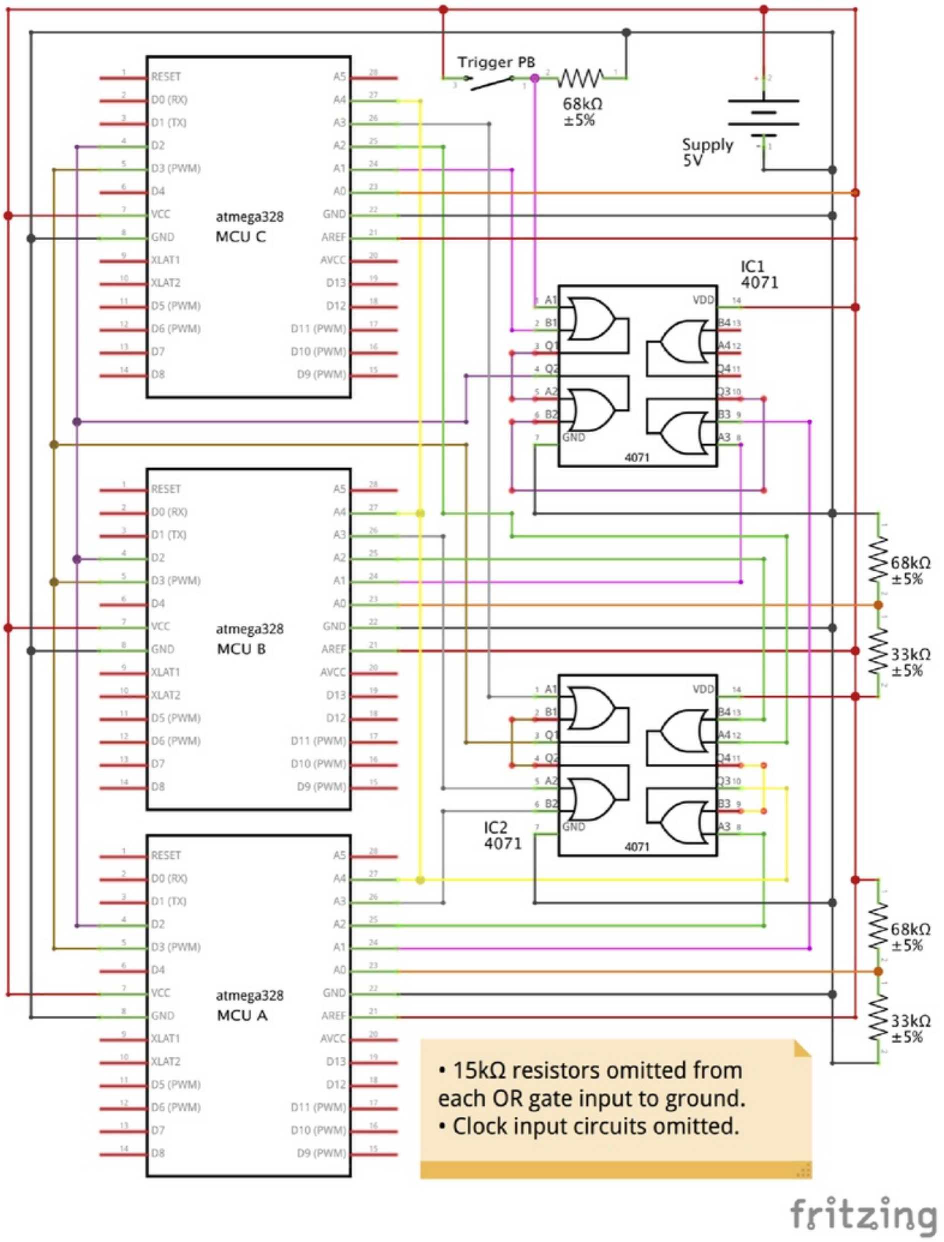
B.2 Supervised Non-Resetting Circuit



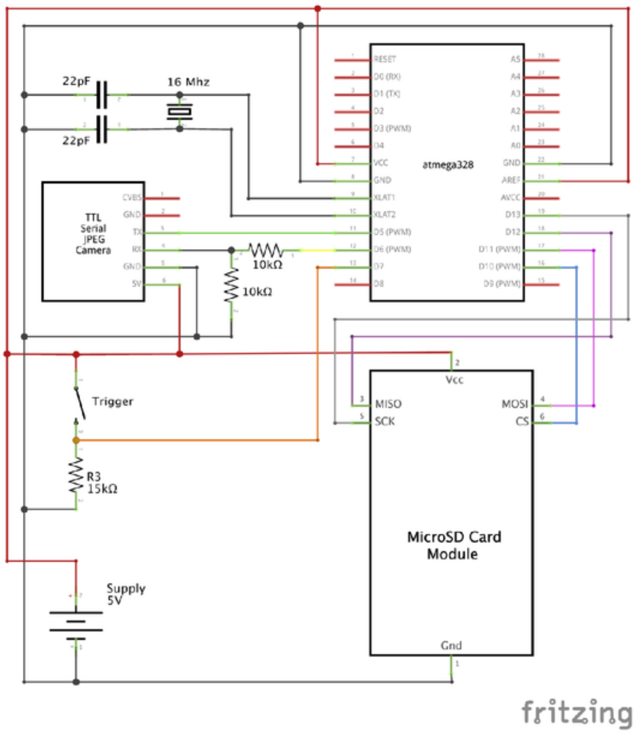
B.3 Supervised Resetting Circuit



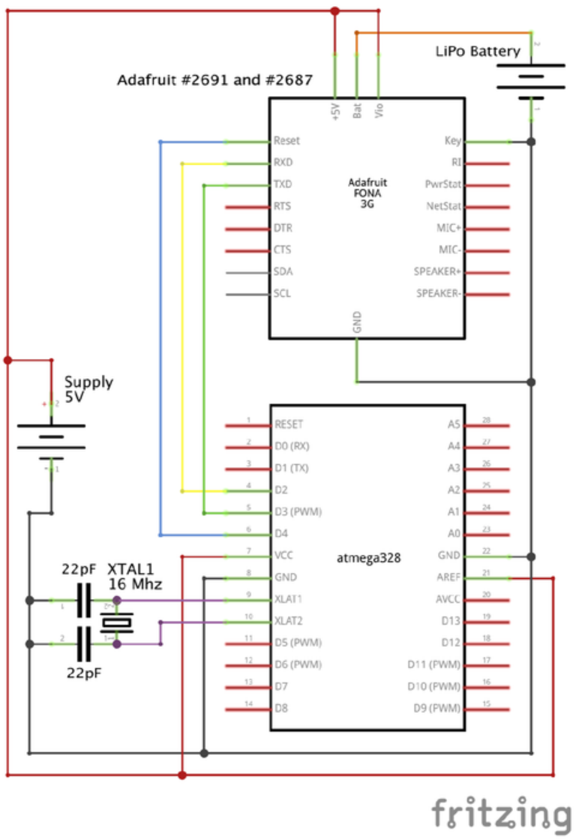
B.4 Self-Supervised Parallel Circuit



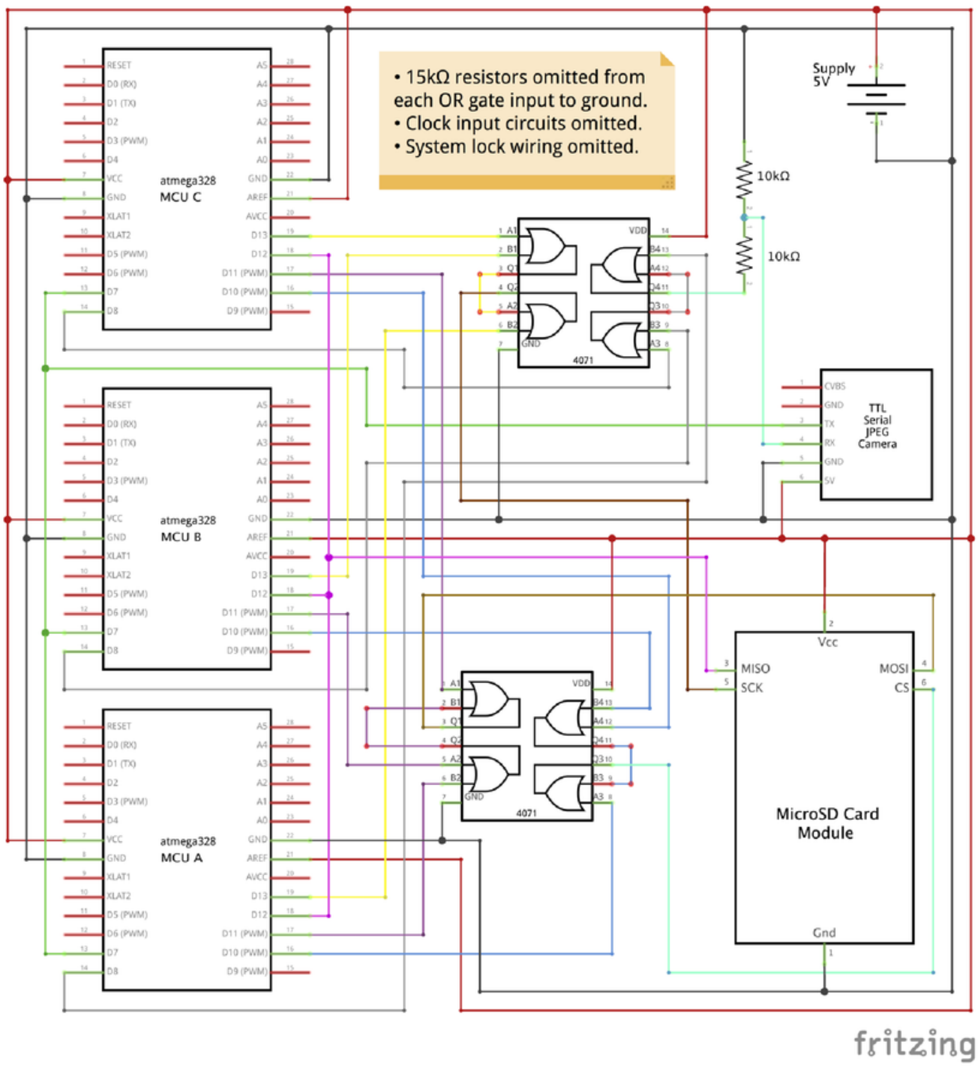
B.5 Proof Of Camera And SD Concept Circuit



B.6 Proof Of Cellular Concept Circuit



B.7 Self-Supervised Camera System Circuit



Appendix C

Arduino Code

C.1 Overview

This appendix contains the Arduino code for the conducted experimentation. Section C.2 shows the Arduino code for the parallel MCUs of the supervised system. Section C.3 lists the Arduino code for the supervisor MCU for the supervised system.

The Arduino code for the supervised MCUs in the resetting supervised system is shown in section C.4. The code for the the supervisor MCUs of the same system is shown in section C.5.

Section C.6 shows the developed Arduion code for a self-supervised redundnacy system. This system incorporated resetting through the use of WDTs.

The developed code for a proof of concept of a camera and SD system is shown in section C.7. The proof of concept for a cellular commmunciation system is shown in section C.8. The modified Arduino library h-file for the cellular system is shown in section C.9. The modified Arduino library cpp-file required for the cellular system is shown in section C.10. The developed code for the ingration of these components into the self-supervised redundancy system is shown in section C.11.

C.2 Non-Resetting Supervised MCU

```
// Declare servo object
#include <Servo.h>
Servo myservo;

// Declare pins
int inPin1 = A0; // Input pin 1
int inPin2 = 9; // Input pin 2
int inPin3 = 10; // Input pin 3
int outPin1 = 11; // Output pin 1
int outPin2 = 12; // Output pin 2
int outPin3 = 13; // Output pin 3

// Declare variables
int in[3]; // Working variables for inputs
int out[3]; // Working variables for outputs

void setup() {
  // Initialise inputs
  pinMode(inPin1, INPUT); // Input 1, potentiometer
  pinMode(inPin2, INPUT); // Input 2, button 1
  pinMode(inPin3, INPUT); // Input 3, button 2

  // Initialise outputs
  // Attach the servo on pin 9 to the servo object
  myservo.attach(outPin1);
  pinMode(outPin2, OUTPUT); // Output 2, red LED.
  pinMode(outPin3, OUTPUT); // Output 3, green LED.
}

void loop() {
  // Read inputs
  in[0] = analogRead(inPin1); // Potentiometer
  in[1] = digitalRead(inPin2); // Button 1
  in[2] = digitalRead(inPin3); // Button 2

  // Calculate outputs
  // Calculate servo level to imitate potentiometer level
  out[0] = map(in[0], 0, 1023, 0, 180);
  // Calculate output 2 as XOR of both button inputs
  out[1] = in[1] * !in[2] + !in[1] * in[2];
  // Calculate output 3 to imitate input 3, button 2
```

```
  out[2] = in[2];

  // Write outputs
  myservo.write(out[0]); // Servo level
  digitalWrite(outPin2, out[1]); // Red LED
  digitalWrite(outPin3, out[2]); // Green LED
}
```

C.3 Non-Resetting Supervisor MCU

```
// Declare supervised pins
int inPin1 = A0; // Input 1 pin
int inPin2 = 9; // Input 2 pin
int inPin3 = 10; // Input 3 pin
int outPin1 = 11; // Output 1 pin
int outPin2 = 12; // Output 2 pin
int outPin3 = 13; // Output 3 pin

// Declare supervision pins
int enPinA = 5; // Enable A pin
int enPinB = 6; // Enable B pin

// Declare variables
int in[6]; // Working variables for inputs
int out[3]; // Working variables for outputs
bool enA = HIGH; // Boolean of A Enable, initialise active
bool enB = LOW; // Boolean of B Enable, initialise inactive
bool allOK; // Boolean status of system

void setup() {
  // Initialise supervisor inputs
  pinMode(inPin1, INPUT); // Input 1, potentiometer
  pinMode(inPin2, INPUT); // Input 2, button 1
  pinMode(inPin3, INPUT); // Input 3, button 2
  pinMode(outPin1, INPUT); // Input 4, servo
  pinMode(outPin2, INPUT); // Input 5, red LED
  pinMode(outPin3, INPUT); // Input 6, green LED

  // Initialise supervisor outputs
  pinMode(enPinA, OUTPUT); // Output 1, Enable A
  pinMode(enPinB, OUTPUT); // Output 2, Enable B
}

void loop() {
  // Read supervised inputs
  in[0] = analogRead(inPin1); // Potentiometer
  in[1] = digitalRead(inPin2); // Button 1
  in[2] = digitalRead(inPin3); // Button 2

  // Read supervised outputs
  in[3] = pulseIn(outPin1, HIGH); // Servo PWM supply
```

```
in[4] = digitalRead(outPin2); // Red LED supply
in[5] = digitalRead(outPin3); // Green LED supply

// Calculate outputs
// Calculate servo level to imitate potentiometer level
out[0] = map(in[0], 0, 1023, 0, 180);
// Calculate output 2 as XOR of both button inputs
out[1] = in[1] * !in[2] + !in[1] * in[2];
// Calculate output 3 to imitate input 3, button 2
out[2] = in[2];

// Translate pulse width into PWM output
in[3] = map(in[3], 480, 2380, 0, 180);

// Compare measured output with calculated outputs
if (abs(in[3] - out[0]) > 10) {
  // Not OK if servo level not within 10   of expectation
  allOK = false;

} else if (in[4] != out[1]) {
  // Not OK if red LED level mismatches expectation
  allOK = false;

} else if (in[5] != out[2]) {
  // Not OK if green LED level mismatches expectation
  allOK = false;

} else {
  // Otherwise reset all OK flag
  allOK = true;
}

// If all is not OK, then toggle enable outputs
if (!allOK) {
  if (enA) {// If A is currently active
    enA = LOW; // Deactivate Enable A
    enB = HIGH; // Activate Enable B

  } else if (enB) { // If B is currently active
    enB = LOW; // Deactivate Enable B
    enA = HIGH; // Activate Enable A
  }
}
```

```

  // Write enable outputs as high active
  digitalWrite(enPinA, enA);
  digitalWrite(enPinB, enB);
}
```

C.4 Resetting Supervised MCU

```
// Declare pins
int inPin1 = 9; // Input pin 1
int inPin2 = 10; // Input pin 2
int outPin1 = 12; // Output pin 1
int outPin2 = 13; // Output pin 2
int lifePin = 4; // Life indicator output pin

// Declare variables
int in[2]; // Working variables for inputs
int out[2]; // Working variables for outputs

void setup() {
  // Initialise inputs
  pinMode(inPin1, INPUT); // Input 1, button 1
  pinMode(inPin2, INPUT); // Input 2, button 2

  // Initialise outputs
  pinMode(outPin1, OUTPUT); // Output 1, red LED
  pinMode(outPin2, OUTPUT); // Output 2, green LED

  //Initialise life indicator pin and turn on
  pinMode(lifePin, OUTPUT);
  digitalWrite(lifePin, HIGH);
}

void loop() {
  // Read inputs
  in[0] = digitalRead(inPin1); // Button 1
  in[1] = digitalRead(inPin2); // Button 2

  // Calculate outputs
  // Calculate output 1 as XOR of both switch inputs
  out[0] = in[0] * !in[1] + !in[0] * in[1];
  // Calculate output 2 to imitate input 3, button 2
  out[1] = in[1];

  // Write outputs
  digitalWrite(outPin2, out[0]); // Red LED
  digitalWrite(outPin3, out[1]); // Green LED
}
```

C.5 Resetting Supervisor MCU

```
// Declare supervised pins
int inPin1 = 9; // Input 1 pin
int inPin2 = 10; // Input 2 pin
int outPin1 = 12; // Output 1 pin
int outPin2 = 13; // Output 2 pin

// Declare supervision pins
int enPinA = 5; // Enable A pin
int enPinB = 6; // Enable B pin
int rstPinA = 7; // Reset A pin
int rstPinB = 8; // Reset B pin
int lifePinA = 2; // Life A pin
int lifePinB = 3; // Life A pin
int alertPin = A1; // Error Alert pin

// Declare variables
int in[6]; // Working variables for inputs
int out[2]; // Working variables for outputs
bool enA = HIGH; // Boolean of A Enable, initialise active
bool enB = LOW; // Boolean of B Enable, initialise unactive
bool allOK; // Boolean status of system

// Declare variables for resetting function
bool rstA = LOW; // Boolean of A Reset, initialise unactive
bool rstB = LOW; // Boolean of B Reset, initialise unactive
int rstCnt = 0; // Count of sequence of resets
int rstCntMax = 3; // Maximum sequence of resets
// Minimum period between sequential resets in milliseconds
int rstDelay = 2000;
long lastRst; // Time of last reset
long currDel; // Current lapsed time since last reset

void setup() {
  // Initialise reset pins and set to high to enable the
  // microcontrollers to start.
  pinMode(rstPinA, OUTPUT);
  pinMode(rstPinB, OUTPUT);
  digitalWrite(rstPinA, HIGH);
  digitalWrite(rstPinB, HIGH);

  // Initialise error alert pin
```

```
  pinMode(alertPin, OUTPUT);

  // Initialise supervisor inputs
  pinMode(inPin1, INPUT); // Input 1, button 1
  pinMode(inPin2, INPUT); // Input 2, button 2
  pinMode(outPin1, INPUT); // Input 3, red LED
  pinMode(outPin2, INPUT); // Input 4, green LED
  pinMode(lifePinA, INPUT); // Input 5, Life A
  pinMode(lifePinB, INPUT); // Input 6, Life B

  // Initialise supervisor outputs
  pinMode(enPinA, OUTPUT); // Output 1, Enable A
  pinMode(enPinB, OUTPUT); // Output 2, Enable B
}

void loop() {
  // Clear reset values
  if (rstA || rstB) {
    rstA = LOW;
    rstB = LOW;
  }

  // Read supervised inputs
  in[0] = digitalRead(inPin1); // Button 1
  in[1] = digitalRead(inPin2); // Button 2

  // Read supervised outputs
  in[2] = digitalRead(outPin1); // Red LED supply
  in[3] = digitalRead(outPin2); // Green LED supply

  // Check supervised controllers for life
  in[4] = digitalRead(lifePinA); // Digital read life of A
  in[5] = digitalRead(lifePinB); // Digital read life of B

  // Calculate outputs
  // Calculate output 2 as XOR of both button inputs
  out[0] = in[0] * !in[1] + !in[0] * in[1];
  // Calculate output 3 to imitate input 3, button 2
  out[1] = in[1];

  // Compare measured output with calculated outputs
  if (in[2] != out[0]) {
    // Not OK if red LED level mismatches expectation
```

```
    allOK = false;

  } else if (in[3] != out[1]) {
    // Not OK if green LED level mismatches expectation
    allOK = false;

  } else {
    // Otherwise reset all OK flag
    allOK = true;
  }

  // Check duration since last reset
  currDel = millis() - lastRst;

  // If the specified length of time has passed, perform
  // any sequential resets
  if (currDel > rstDelay) {
    if (in[4] == 1 && in[5] == 1) {
      // Deactivate alert LED output
      digitalWrite(alertPin, LOW);
      rstCnt = 0; // Clear count of resets

    } else if (rstCnt >= rstCntMax) {
      // Activate alert LED output
      digitalWrite(alertPin, HIGH);

    } else if (enA && in[5] == 0) {
      enB = LOW; // Dectiavte Enable B
      enA = HIGH; // Activate Enable A
      rstB = HIGH; // Activate Reset B
      rstCnt++; // Increment count of resets
      lastRst = millis();// Record time of reset

    } else if (enB && in[4] == 0) {
      enA = LOW; // Dectiavte Enable A
      enB = HIGH; // Activate Enable B
      rstA = HIGH; // Activate Reset A
      rstCnt++; // Increment count of resets
      lastRst = millis();// Record time of reset

    } else if (in[5] == 0) {
      enB = LOW; // Dectiavte Enable B
      enA = HIGH; // Set EnA to high
```

```
      rstB = HIGH; // Activate Reset B
      rstCnt++; // Increment count of resets
      lastRst = millis();// Record time of reset

    } else if (in[4] == 0) {
      enA = LOW; // Dectiavte Enable A
      enB = HIGH; // Activate Enable B
      rstA = HIGH; // Activate Reset A
      rstCnt++; // Increment count of resets
      lastRst = millis();// Record time of reset
    }
  }

  // If not all is OK and the secondary mC is alive, then
  // toggle enable outputs and reset former primary mC
  if (!allOK) {
    if (enA && in[5] == 1) {
      // If A is currently active and B is alive
      enA = LOW; // Dectiavte Enable A
      enB = HIGH; // Activate Enable B
      rstA = HIGH; // Activate Reset A
      rstCnt ++; // Increment count of resets
      lastRst = millis(); // Record time of reset

    } else if (enB && in[4] == 1) {
      // If B is currently active and A is alive
      enB = LOW; // Dectiavte Enable B
      enA = HIGH; // Activate Enable A
      rstB = HIGH; // Activate Reset B
      rstCnt++; // Increment count of resets
      lastRst = millis();// Record time of reset
    }
  }

  // Write enable outputs as high active
  digitalWrite(enPinA, enA);
  digitalWrite(enPinB, enB);

  // Write reset outputs as low active
  digitalWrite(rstPinA, !rstA);
  digitalWrite(rstPinB, !rstB);
}
```

C.6 Self-Supervised Parallel MCU

```
// Declare pins
int pFixedId = A0; // Pin for input of fixed ID
int pClearOut = A1; // Pin for output of clearance flag
int pLockOut = A2; // Pin for output of lock flag
int pProcOut = A3; // Pin for output of processing flag
int pLockIn = A4; // Pin for input of lock flag
int pClearIn = 2; // Pin for input of clearance flag
int pProcIn = 3; // Pin for input of processing flag
int pPrimeLED = 9; // Pin for debugging LED
int pDebug = A5; // Pin for debugging button input

// Declare exclusion lock variables
int fixedId = 0; // Fixed ID
boolean clearance = false; // Clearance flag
boolean currPrime = false; // Current primary MCU flag
long lastClear = 0; // Time of last clearance
int extraDel = 0; // Delay of former primary in ms
boolean debug; // Stored value of debugging button input

// Declare exclusion lock variables as volatile that will
// be modified by the interrupt subroutine
// Current time taken to process
volatile long elapsedProc = 0;
// Flag to start new processing
volatile boolean newProc = false;
// Flag for returning to the start of the main loop
volatile boolean startReturn = false;
volatile boolean locked = false; // Lock flag
volatile boolean processing = false; // Processing flag

// Declare watchdog timer (WDT) components
#include <avr/wdt.h> // Include WDT library
// Flag for disabling main loop after the WDT has timed out
boolean wdtOverRide = false;

// Declare system function variables
int maxProc = 3000; // Maximum processing time in ms

void setup() {
  // Initialise pins
  pinMode(pFixedId, INPUT); // Input of fixed ID voltage
```

```
  pinMode(pClearOut, OUTPUT); // Output of clearance flag
  pinMode(pLockOut, OUTPUT); // Output of lock flag
  pinMode(pProcOut, OUTPUT); // Output of processing flag
  pinMode(pLockIn, INPUT); // Input of lock flag
  pinMode(pClearIn, INPUT); // Input of clearance flag
  pinMode(pProcIn, INPUT); // Input of processing flag
  pinMode(pPrimeLED, OUTPUT); // Output of primary flag LED
  pinMode(pDebug, INPUT); // Input for debugging button
  // Map fixed ID input voltage to integer value
  fixedId = int(round(analogRead(pFixedId) * 3.0 / 1024));
  // Attach clearance/trigger interrupt subroutine (ISR)
  // to digital pin 2, interrupt 0
  attachInterrupt(0, clear_ISR, CHANGE);
  watchdogSetup(); // Initialise WDT
}

// ISR for clearance flag
void clear_ISR() {
  // Calculate time elapsed since last effective clearance
  elapsedProc = millis() - lastClear;
  locked = digitalRead(pLockIn); // Read lock flag
  // Read processing flag
  processing = digitalRead(pProcIn);
  if (currPrime && elapsedProc > maxProc) {
    // If currently processing but longer than expected
    newProc = true; // Set flag of new process
    startReturn = true;
    digitalWrite(pLockOut, false); // Clear lock flag
  } else if (processing && !locked) {
    // Else if currently processing with clear lock flag
    newProc = true; // Set flag of new process
    // Set flag to return to start of main loop
    startReturn = true;
  } else if (processing && elapsedProc < maxProc) {
    // Else if currently processing within expected time
    ; // Don't change
    newProc = false; // Clear flag of new process
  } else if (currPrime) { // Else if current primary
    // Else if currently the primary MCU
    newProc = true; // Set flag of new process
    // Set flag to return to start of main loop
    startReturn = true;
    digitalWrite(pLockOut, false); // Clear lock flag
```

```
  } else {
    newProc = true; // Set flag of new process
    // Set flag to return to start of main loop
    startReturn = true;
  }
  // Perform new process section in loop to avoid having
  // delays in the interrupt subroutine.
}


void loop() {
  // If WDT override flag is set, disable function of
  // main loop
  if (wdtOverRide) {
    delay(10); // Brief delay
    if (currPrime) { // If currently the prime,
      // If currently the prime, prompt a possible primary
      // reassignment
      digitalWrite(pClearOut, true); // Set clearance flag
    }
    if (processing) {
      // If the processing flag is set, maintain flag to
      // hand over to next available MCU
      digitalWrite(pProcOut, true); // Set processing flag
    }
    digitalWrite(pClearOut, false); // Clear clearance flag
    // Return to main loop start, to prevent interference
    // with main system function
    return;
  }
  // Clear flag to return to start of main loop
  startReturn = false;
  if (newProc) {
    // If new process, assign/reassign primary priority
    newProc = false; // Clear new process flag

    if (currPrime) { // If current primary
      // Delay 100ms due to being previous primary
      extraDel = 100;
      currPrime = false; // Clear current primary flag
    } else { // Otherwise
      // Delay 0ms due to not being previous primary
      extraDel = 0;
```

```
    }

    digitalWrite(pProcOut, true); // set processing flag
    delay(extraDel); // Wait for any offset delay
    // Delay by a scalar of fixed ID, reduced by 1 to
    // prevent unnecessary delays
    delay((fixedId - 1) * 30);
    // Record time of start of processing
    lastClear = millis();
    locked = digitalRead(pLockIn); // Read lock flag
    if (!locked) { // If not locked, take on primary role
      digitalWrite(pLockOut, true); // Set lock flag
      locked = true; // Set internal lock flag
      currPrime = true; // Set current primary flag
    } else { // If locked already, remain as non-primary
      // Clear processing flag
      digitalWrite(pProcOut, false);
    }
  }

  debug = digitalRead(pDebug); // Read debugging input
  if (!debug && !startReturn) {
    // If the debugging button is not pressed, reset WDT
    // Also include check of return to start flag
    wdt_reset(); // Reset WDT
  }

  // Update elapsed time since last effective clearance
  elapsedProc = millis() - lastClear;
  locked = digitalRead(pLockIn); // Read lock flag
  // Read processing flag
  processing = digitalRead(pProcIn);

  if (!startReturn) { // If not restarting main loop
    if (currPrime) { // If current primary flagged
      digitalWrite(pPrimeLED, HIGH); // Set debug pin


      // Perform system function here in a single process
      // or multiple cycles include resets for the WDT to
      // ensure activity within the sensitivity timeframe
      // Also include check for return to start flag

```

```

      // If function concluded clear all flags
      // Also include check for return to start flag
      if (elapsedProc > 4000 && !startReturn) {
        digitalWrite(pLockOut, false); // Clear lock flag
        // Clear processing flag
        digitalWrite(pProcOut, false);
        currPrime = false;// Clear current primary flag
      }
    } else { // Else, if current primary not flagged
      digitalWrite(pPrimeLED, LOW); // Clear debug pin
      digitalWrite(pLockOut, false); // Clear lock flag
      // Clear processing flag
      digitalWrite(pProcOut, false);
      // If elapsed time is greater than expected
      // Also include check for return to start flag
      if (elapsedProc > maxProc  && !startReturn) {
        // Read processing flag
        processing = digitalRead(pProcIn);
        if (processing && !startReturn) {
          // If processing flag is still set
          // Also include check for return to start flag
          // Set clearance flag for reassignment of primary
          // priority
          digitalWrite(pClearOut, true);
          // Immediately clear clearance flag
          digitalWrite(pClearOut, false);
        }
      }
    }
  }
}

void watchdogSetup(void) { // Initialise WDT configuration
  cli(); // Disable all interrupts
  wdt_reset(); // Reset the WDT
  WDTCSR |= B00011000; // Enter WDT configuration mode
  // Set WDT settings to activate closing ISR,
  // activate WDT, and set a timeout of 250ms
  WDTCSR = B01001100; // Write configuration to WDT
  sei(); // Enable interrupts
}

```

```
ISR(WDT_vect) { // Watchdog timer ISR
  startReturn = true; // Set return to start flag
  wdtOverRide = true; // Set flag to disable main loop
  digitalWrite(pLockOut, false); // Clear lock flag
}
```

C.7 Proof Of Camera And SD Concept

```
// Declare included libraries
#include <Adafruit_VC0706.h> // Camera
#include <SPI.h> // Serial peripheral interface
#include <SD.h> // SD card
#include <SoftwareSerial.h> // Additional serial port

// Declare new serial ports for camera, (RX, TX)
SoftwareSerial cameraconnection = SoftwareSerial(5, 6);
// Declare camera object
Adafruit_VC0706 cam = Adafruit_VC0706(&cameraconnection);

// Declare other pins
int pChipSelect = 10; // Pin for SD chip select
int pButton = 7; // Pin for camera trigger button
int startTime = 0; // Time of triggering
int duration = 0; // Time taken to record image
// Time for camera to capture after triggering
int responseTime = 0;

// Declare other variables
File imgFile; // Variable for image storage transfer
bool buttonPressed = false; // Variable button result


void setup() {
  // Initialise SD card chip select pin
  pinMode(pChipSelect, OUTPUT); // SS on Uno, etc.

  // Initialise serial connection with computer
  Serial.begin(9600);
  // Notify of start of testing camera system
  Serial.println("VC0706 Camera test");

  // Test SD card connection
  if (SD.begin(pChipSelect)) {
    Serial.println("Card found");
  } else {
    Serial.println("Card not found");
    return; // Abandon attempt
  }

```

```
  // Test camera connection
  if (cam.begin()) {
    Serial.println("Camera found");
  } else {
    Serial.println("Camera not found");
    return; // Abandon attempt
  }

  // Select desired image size
  cam.setImageSize(VC0706_640x480); // Biggest
  //  cam.setImageSize(VC0706_320x240); // Medium
  //  cam.setImageSize(VC0706_160x120); // Small

  // Initialise trigger button input
  pinMode(pButton, INPUT);
}


void loop() {
  // Read trigger button value
  buttonPressed = digitalRead(pButton);
  // If button is pushed, capture an image
  if (buttonPressed) {
    startTime = millis();
    captureImage();
  }
}


void captureImage() {
  if (cam.takePicture()) {
    Serial.print("Image captured on camera in ");
    responseTime = millis() - startTime;
    Serial.print(responseTime);
    Serial.println(" ms!");

  } else {
    Serial.println("Image not captured");
  }

  // Create unique image file name on SD card
  char filename[13];
  strcpy(filename, "IMG0000.JPG");
```

```
  for (int i = 0; i < 1000; i++) {
    filename[4] = '0' + i / 100;
    filename[5] = '0' + (i % 100) / 10;
    filename[6] = '0' + (i % 100) % 10;
    // Create if does not exist, do not open existing,
    // write, sync after write
    if (! SD.exists(filename)) {
      Serial.println(filename);
      break;
    }
  }

  // Initialise new image file on SD card
  imgFile = SD.open(filename, FILE_WRITE);

  // Identify size of image to be saved
  uint16_t jpglen = cam.frameLength();
  // While image still has untransfered data
  while (jpglen > 0) {
    // Read 32 bytes at a time;
    uint8_t bytesToRead = min(64, jpglen);
    // Prepare buffer of received data
    uint8_t *buffer = cam.readPicture(bytesToRead);
    // Write received data from buffer to file on SD card
    imgFile.write(buffer, bytesToRead);
    // Deduct size of written data from remaining quantity
    jpglen -= bytesToRead;
  }
  // Close image file
  imgFile.close();
  // Notify operator of conclusion
  Serial.print("...Done in ");
  duration = millis() - startTime;
  Serial.print(duration);
  Serial.println(" ms!");
  // Set camera back into video mode, to clear captured
  // image from camera memory, in readiness for next image
  cam.resumeVideo();
}
```

C.8 Proof Of Cellular Concept

```
/***************************************************
  This is an example for our Adafruit FONA Cellular Module

  Designed specifically to work with the Adafruit FONA
  ----> http://www.adafruit.com/products/1946
  ----> http://www.adafruit.com/products/1963
  ----> http://www.adafruit.com/products/2468
  ----> http://www.adafruit.com/products/2542

  These cellular modules use TTL Serial to communicate, 2 pins are
  required to interface
  Adafruit invests time and resources providing this open source code,
  please support Adafruit and open-source hardware by purchasing
  products from Adafruit!

  Written by Limor Fried/Ladyada for Adafruit Industries.
  BSD license, all text above must be included in any redistribution
 ****************************************************/

/*
  THIS CODE IS STILL IN PROGRESS!

  Open up the serial console on the Arduino at 115200 baud to interact with FONA

  Note that if you need to set a GPRS APN, username, and password scroll down to
  the commented section below at the end of the setup() function.
*/
#include "Adafruit_FONA.h"

#define FONA_RX 2
#define FONA_TX 3
#define FONA_RST 4

// this is a large buffer for replies
```

```
char replybuffer[255];

// We default to using software serial. If you want to use hardware serial
// (because softserial isnt supported) comment out the following three lines
// and uncomment the HardwareSerial line
#include <SoftwareSerial.h>
SoftwareSerial fonaSS = SoftwareSerial(FONA_TX, FONA_RX);
SoftwareSerial *fonaSerial = &fonaSS;

// Hardware serial is also possible!
//  HardwareSerial *fonaSerial = &Serial1;

// Use this for FONA 800 and 808s
Adafruit_FONA fona = Adafruit_FONA(FONA_RST);
// Use this one for FONA 3G
//Adafruit_FONA_3G fona = Adafruit_FONA_3G(FONA_RST);

uint8_t readline(char *buff, uint8_t maxbuff, uint16_t timeout = 0);

uint8_t type;

void setup() {
  while (!Serial);

  Serial.begin(115200);
  Serial.println(F("FONA basic test"));
  Serial.println(F("Initializing....(May take 3 seconds)"));

  fonaSerial->begin(4800);
  if (! fona.begin(*fonaSerial)) {
    Serial.println(F("Couldn't find FONA"));
    while (1);
  }
  type = fona.type();
  Serial.println(F("FONA is OK"));
  Serial.print(F("Found "));
  switch (type) {
    case FONA3G_A:
      Serial.println(F("FONA 3G (American)")); break;
```

```
    case FONA3G_E:
      Serial.println(F("FONA 3G (European)")); break;
    default:
      Serial.println(F("???")); break;
  }

  // Print module IMEI number.
  char imei[16] = {0}; // MUST use a 16 character buffer for IMEI!
  uint8_t imeiLen = fona.getIMEI(imei);
  if (imeiLen > 0) {
    Serial.print("Module IMEI: "); Serial.println(imei);
  }

  // Optionally configure a GPRS APN, username, and password.
  // You might need to do this to access your network's GPRS/data
  // network.  Contact your provider for the exact APN, username,
  // and password values.  Username and password are optional and
  // can be removed, but APN is required.
  fona.setGPRSNetworkSettings(F("internet"), F(""), F(""));

  // Optionally configure HTTP gets to follow redirects over SSL.
  // Default is not to follow SSL redirects, however if you uncomment
  // the following line then redirects over SSL will be followed.
  //fona.setHTTPSRedirect(true);

  unlockPin();

  printMenu();
}

void printMenu(void) {
  Serial.println(F("-------------------------------------"));
  Serial.println(F("[?] Print this menu"));
```

```
  Serial.println(F("[b] read the Battery V and % charged"));
  Serial.println(F("[C] read the SIM CCID"));
  //  Serial.println(F("[U] Unlock SIM with PIN code"));
  Serial.println(F("[i] read RSSI"));
  Serial.println(F("[n] get Network status"));

  // Phone
  Serial.println(F("[c] make phone Call"));
  Serial.println(F("[A] get call status"));
  Serial.println(F("[h] Hang up phone"));
  Serial.println(F("[p] Pick up phone"));

  // SMS
  Serial.println(F("[N] Number of SMSs"));
  Serial.println(F("[r] Read SMS #"));
  Serial.println(F("[R] Read All SMS"));
  Serial.println(F("[d] Delete SMS #"));
  Serial.println(F("[s] Send SMS"));
  Serial.println(F("[u] Send USSD"));

  // Email
  Serial.println(F("[J] Send email"));

  // GPRS
  Serial.println(F("[G] Enable GPRS"));
  Serial.println(F("[g] Disable GPRS"));
  Serial.println(F("[l] Query GSMLOC (GPRS)"));
  Serial.println(F("[w] Read webpage (GPRS)"));
  Serial.println(F("[W] Post to website (GPRS)"));

  Serial.println(F("[S] create Serial passthru tunnel"));
  Serial.println(F("-------------------------------------"));
  Serial.println(F(""));

}
void loop() {
  Serial.print(F("FONA> "));
  while (! Serial.available() ) {
    if (fona.available()) {
      Serial.write(fona.read());
    }
```

```
  }

  char command = Serial.read();
  Serial.println(command);


  switch (command) {
    case '?': {
        printMenu();
        break;
      }

    case 'J': {
        if (! fona.sendTestMail(true)) {
          Serial.println(F("Failed to send email"));
        } else {
          Serial.print(F("Sent email"));
        }
        break;
      }


    case 'a': {
        // read the ADC
        uint16_t adc;
        if (! fona.getADCVoltage(&adc)) {
          Serial.println(F("Failed to read ADC"));
        } else {
          Serial.print(F("ADC = ")); Serial.print(adc);
             Serial.println(F(" mV"));
        }
        break;
      }

    case 'b': {
        // read the battery voltage and percentage
        uint16_t vbat;
        if (! fona.getBattVoltage(&vbat)) {
          Serial.println(F("Failed to read Batt"));
        } else {
          Serial.print(F("VBat = ")); Serial.print(vbat);
             Serial.println(F(" mV"));
        }
```

```


        if (! fona.getBattPercent(&vbat)) {
          Serial.println(F("Failed to read Batt"));
        } else {
          Serial.print(F("VPct = ")); Serial.print(vbat);
             Serial.println(F("%"));
        }

        break;
      }

    case 'U': {
        // Unlock the SIM with a PIN code
        char PIN[5];
        flushSerial();
        Serial.println(F("Enter 4-digit PIN"));
        readline(PIN, 3);
        Serial.println(PIN);
        Serial.print(F("Unlocking SIM card: "));
        if (! fona.unlockSIM(PIN)) {
          Serial.println(F("Failed"));
        } else {
          Serial.println(F("OK!"));
        }
        break;
      }

    case 'C': {
        // read the CCID
        fona.getSIMCCID(replybuffer);  // make sure
           replybuffer is at least 21 bytes!
        Serial.print(F("SIM CCID = ")); Serial.println(
           replybuffer);
        break;
      }

    case 'i': {
        // read the RSSI
        uint8_t n = fona.getRSSI();
        int8_t r;

```

```
        Serial.print(F("RSSI = ")); Serial.print(n); Serial.print(": ");
        if (n == 0) r = -115;
        if (n == 1) r = -111;
        if (n == 31) r = -52;
        if ((n >= 2) && (n <= 30)) {
          r = map(n, 2, 30, -110, -54);
        }
        Serial.print(r); Serial.println(F(" dBm"));

        break;
      }

    case 'n': {
        // read the network/cellular status
        uint8_t n = fona.getNetworkStatus();
        Serial.print(F("Network status "));
        Serial.print(n);
        Serial.print(F(": "));
        if (n == 0) Serial.println(F("Not registered"));
        if (n == 1) Serial.println(F("Registered (home)"));
        if (n == 2) Serial.println(F("Not registered (searching)"));
        if (n == 3) Serial.println(F("Denied"));
        if (n == 4) Serial.println(F("Unknown"));
        if (n == 5) Serial.println(F("Registered roaming"));
        break;
      }


    /*** Call ***/
    case 'c': {
        // call a phone!
        char number[30];
        flushSerial();
        Serial.print(F("Call #"));
        readline(number, 30);
        Serial.println();
        Serial.print(F("Calling ")); Serial.println(number);
        if (!fona.callPhone(number)) {
          Serial.println(F("Failed"));
```

```
        } else {
          Serial.println(F("Sent!"));
        }

        break;
      }
    case 'A': {
        // get call status
        int8_t callstat = fona.getCallStatus();
        switch (callstat) {
          case 0: Serial.println(F("Ready")); break;
          case 1: Serial.println(F("Could not get status"))
              ; break;
          case 3: Serial.println(F("Ringing (incoming)"));
              break;
          case 4: Serial.println(F("Ringing/in progress (
              outgoing)")); break;
          default: Serial.println(F("Unknown")); break;
        }
        break;
      }

    case 'h': {
        // hang up!
        if (! fona.hangUp()) {
          Serial.println(F("Failed"));
        } else {
          Serial.println(F("OK!"));
        }
        break;
      }

    case 'p': {
        // pick up!
        if (! fona.pickUp()) {
          Serial.println(F("Failed"));
        } else {
          Serial.println(F("OK!"));
        }
        break;
      }

    /*** SMS ***/
```

```

    case 'N': {
        // read the number of SMS's!
        int8_t smsnum = fona.getNumSMS();
        if (smsnum < 0) {
          Serial.println(F("Could not read # SMS"));
        } else {
          Serial.print(smsnum);
          Serial.println(F(" SMS's on SIM card!"));
        }
        break;
      }
    case 'r': {
        // read an SMS
        flushSerial();
        Serial.print(F("Read #"));
        uint8_t smsn = readnumber();
        Serial.print(F("\n\rReading SMS #")); Serial.println(smsn);

        // Retrieve SMS sender address/phone number.
        if (! fona.getSMSSender(smsn, replybuffer, 250)) {
          Serial.println("Failed!");
          break;
        }
        Serial.print(F("FROM: ")); Serial.println(replybuffer);

        // Retrieve SMS value.
        uint16_t smslen;
        if (! fona.readSMS(smsn, replybuffer, 250, &smslen)) { // pass in buffer and max len!
          Serial.println("Failed!");
          break;
        }
        Serial.print(F("***** SMS #")); Serial.print(smsn);
        Serial.print(" ("); Serial.print(smslen); Serial.println(F(") bytes *****"));
        Serial.println(replybuffer);
        Serial.println(F("*****"));

        break;
      }
```

```
    case 'R': {
        // read all SMS
        int8_t smsnum = fona.getNumSMS();
        uint16_t smslen;
        int8_t smsn;

        if ( (type == FONA3G_A) || (type == FONA3G_E) ) {
          smsn = 0; // zero indexed
          smsnum--;
        } else {
          smsn = 1;  // 1 indexed
        }

        for ( ; smsn <= smsnum; smsn++) {
          Serial.print(F("\n\rReading SMS #")); Serial.println(smsn);
          if (!fona.readSMS(smsn, replybuffer, 250, &smslen)) {  // pass in buffer and max len!
            Serial.println(F("Failed!"));
            break;
          }
          // if the length is zero, its a special case where the index number is higher
          // so increase the max we'll look at!
          if (smslen == 0) {
            Serial.println(F("[empty slot]"));
            smsnum++;
            continue;
          }

          Serial.print(F("***** SMS #")); Serial.print(smsn);
          Serial.print(" ("); Serial.print(smslen); Serial.println(F(") bytes *****"));
          Serial.println(replybuffer);
          Serial.println(F("*****"));
        }
        break;
      }

    case 'd': {
        // delete an SMS
        flushSerial();
```

```
        Serial.print(F("Delete #"));
        uint8_t smsn = readnumber();

        Serial.print(F("\n\rDeleting SMS #")); Serial.
           println(smsn);
        if (fona.deleteSMS(smsn)) {
          Serial.println(F("OK!"));
        } else {
          Serial.println(F("Couldn't delete"));
        }
        break;
      }

    case 's': {
        // send an SMS!
        char sendto[21], message[141];
        flushSerial();
        Serial.print(F("Send to #"));
        readline(sendto, 20);
        Serial.println(sendto);
        Serial.print(F("Type out one-line message (140 char
           ): "));
        readline(message, 140);
        Serial.println(message);
        if (!fona.sendSMS(sendto, message)) {
          Serial.println(F("Failed"));
        } else {
          Serial.println(F("Sent!"));
        }

        break;
      }

    case 'u': {
        // send a USSD!
        char message[141];
        flushSerial();
        Serial.print(F("Type out one-line message (140 char
           ): "));
        readline(message, 140);
        Serial.println(message);

        uint16_t ussdlen;
```

```
        if (!fona.sendUSSD(message, replybuffer, 250, &
           ussdlen)) { // pass in buffer and max len!
          Serial.println(F("Failed"));
        } else {
          Serial.println(F("Sent!"));
          Serial.print(F("***** USSD Reply"));
          Serial.print(" ("); Serial.print(ussdlen); Serial
             .println(F(") bytes *****"));
          Serial.println(replybuffer);
          Serial.println(F("*****"));
        }
      }


    /********************************** GPRS */

    case 'g': {
        // turn GPRS off
        if (!fona.enableGPRS(false))
          Serial.println(F("Failed to turn off"));
        break;
      }
    case 'G': {
        // turn GPRS on
        if (!fona.enableGPRS(true))
          Serial.println(F("Failed to turn on"));
        break;
      }
    case 'l': {
        // check for GSMLOC (requires GPRS)
        uint16_t returncode;

        if (!fona.getGSMLoc(&returncode, replybuffer, 250))
          Serial.println(F("Failed!"));
        if (returncode == 0) {
          Serial.println(replybuffer);
        } else {
          Serial.print(F("Fail code #")); Serial.println(
             returncode);
        }

        break;
      }
```

```
    case 'w': {
        // read website URL
        uint16_t statuscode;
        int16_t length;
        char url[80];

        flushSerial();
        Serial.println(F("NOTE: in beta! Use small webpages to read!"));
        Serial.println(F("URL to read (e.g. www.adafruit.com/testwifi/index.html):"));
        Serial.print(F("http://")); readline(url, 79);
        Serial.println(url);

        Serial.println(F("****"));
        if (!fona.HTTP_GET_start(url, &statuscode, (uint16_t *)&length)) {
          Serial.println("Failed!");
          break;
        }
        while (length > 0) {
          while (fona.available()) {
            char c = fona.read();

            // Serial.write is too slow, we'll write directly to Serial register!
#if defined(__AVR_ATmega328P__) || defined(__AVR_ATmega168__)
            loop_until_bit_is_set(UCSR0A, UDRE0); /* Wait until data register empty. */
            UDR0 = c;
#else
            Serial.write(c);
#endif
            length--;
            if (! length) break;
          }
        }
        Serial.println(F("\n****"));
        fona.HTTP_GET_end();
        break;
      }

```

```
    case 'W': {
        // Post data to website
        uint16_t statuscode;
        int16_t length;
        char url[80];
        char data[80];

        flushSerial();
        Serial.println(F("NOTE: in beta! Use simple websites to post!"));
        Serial.println(F("URL to post (e.g. httpbin.org/post):"));
        Serial.print(F("http://")); readline(url, 79);
        Serial.println(url);
        Serial.println(F("Data to post (e.g. \"foo\" or \"{\"simple\":\"json\"}\"):"));
        readline(data, 79);
        Serial.println(data);

        Serial.println(F("****"));
        if (!fona.HTTP_POST_start(url, F("text/plain"), (uint8_t *) data, strlen(data), &statuscode, (uint16_t *)&length)) {
          Serial.println("Failed!");
          break;
        }
        while (length > 0) {
          while (fona.available()) {
            char c = fona.read();

#if defined(__AVR_ATmega328P__) || defined(__AVR_ATmega168__)
            loop_until_bit_is_set(UCSR0A, UDRE0); /* Wait until data register empty. */
            UDR0 = c;
#else
            Serial.write(c);
#endif

            length--;
            if (! length) break;
          }
        }
```

```
        Serial.println(F("\n****"));
        fona.HTTP_POST_end();
        break;
      }
    /*****************************************/

    case 'S': {
        Serial.println(F("Creating SERIAL TUBE"));
        while (1) {
          while (Serial.available()) {
            delay(1);
            fona.write(Serial.read());
          }
          if (fona.available()) {
            Serial.write(fona.read());
          }
        }
        break;
      }

    default: {
        Serial.println(F("Unknown command"));
        printMenu();
        break;
      }
  }
  // flush input
  flushSerial();
  while (fona.available()) {
    Serial.write(fona.read());
  }

}

void flushSerial() {
  while (Serial.available())
    Serial.read();
}

char readBlocking() {
  while (!Serial.available());
  return Serial.read();
}
```

```
uint16_t readnumber() {
  uint16_t x = 0;
  char c;
  while (! isdigit(c = readBlocking())) {
    //Serial.print(c);
  }
  Serial.print(c);
  x = c - '0';
  while (isdigit(c = readBlocking())) {
    Serial.print(c);
    x *= 10;
    x += c - '0';
  }
  return x;
}

uint8_t readline(char *buff, uint8_t maxbuff, uint16_t
    timeout) {
  uint16_t buffidx = 0;
  boolean timeoutvalid = true;
  if (timeout == 0) timeoutvalid = false;

  while (true) {
    if (buffidx > maxbuff) {
      //Serial.println(F("SPACE"));
      break;
    }

    while (Serial.available()) {
      char c =  Serial.read();

      //Serial.print(c, HEX); Serial.print("#"); Serial.
         println(c);

      if (c == '\r') continue;
      if (c == 0xA) {
        if (buffidx == 0)   // the first 0x0A is ignored
          continue;

        timeout = 0;         // the second 0x0A is the end
           of the line
        timeoutvalid = true;
        break;
```

```
      }
      buff[buffidx] = c;
      buffidx++;
    }

    if (timeoutvalid && timeout == 0) {
      //Serial.println(F("TIMEOUT"));
      break;
    }
    delay(1);
  }
  buff[buffidx] = 0;  // null term
  return buffidx;
}

void unlockPin() {
  //void unlockPin(String input) {
  char PIN[5] = {'0', '0', '0', '0'};
  if (! fona.unlockSIM(PIN)) {
    Serial.println(F("Sim Unlocking Failed"));
  } else {
    Serial.println(F("Sim Unlocked!"));
  }
}
```

C.9 Modified Adafruit FONA Library H-File

```
/***************************************************
  This is a library for our Adafruit FONA Cellular Module

  Designed specifically to work with the Adafruit FONA
  ----> http://www.adafruit.com/products/1946
  ----> http://www.adafruit.com/products/1963

  These displays use TTL Serial to communicate, 2 pins are
     required to
  interface
  Adafruit invests time and resources providing this open
     source code,
  please support Adafruit and open-source hardware by
     purchasing
  products from Adafruit!

  Written by Limor Fried/Ladyada for Adafruit Industries.
  BSD license, all text above must be included in any
     redistribution
 ****************************************************/
#ifndef ADAFRUIT_FONA_H
#define ADAFRUIT_FONA_H

#include "includes/FONAConfig.h"
#include "includes/FONAExtIncludes.h"
#include "includes/platform/FONAPlatform.h"



#define FONA800L 1
#define FONA800H 6

#define FONA808_V1 2
#define FONA808_V2 3

#define FONA3G_A 4
#define FONA3G_E 5

// Set the preferred SMS storage.
//   Use "SM" for storage on the SIM.
//   Use "ME" for internal storage on the FONA chip
```

```
#define FONA_PREF_SMS_STORAGE "\"SM\""
//#define FONA_PREF_SMS_STORAGE "\"ME\""

#define FONA_HEADSETAUDIO 0
#define FONA_EXTAUDIO 1

#define FONA_STTONE_DIALTONE 1
#define FONA_STTONE_BUSY 2
#define FONA_STTONE_CONGESTION 3
#define FONA_STTONE_PATHACK 4
#define FONA_STTONE_DROPPED 5
#define FONA_STTONE_ERROR 6
#define FONA_STTONE_CALLWAIT 7
#define FONA_STTONE_RINGING 8
#define FONA_STTONE_BEEP 16
#define FONA_STTONE_POSTONE 17
#define FONA_STTONE_ERRTONE 18
#define FONA_STTONE_INDIANDIALTONE 19
#define FONA_STTONE_USADIALTONE 20

#define FONA_DEFAULT_TIMEOUT_MS 500

#define FONA_HTTP_GET   0
#define FONA_HTTP_POST  1
#define FONA_HTTP_HEAD  2

#define FONA_CALL_READY 0
#define FONA_CALL_FAILED 1
#define FONA_CALL_UNKNOWN 2
#define FONA_CALL_RINGING 3
#define FONA_CALL_INPROGRESS 4

class Adafruit_FONA : public FONAStreamType {
 public:
  Adafruit_FONA(int8_t r);
  boolean begin(FONAStreamType &port);
  uint8_t type();

  // Stream
  int available(void);
  size_t write(uint8_t x);
  int read(void);
  int peek(void);
```

```
void flush();

// FONA 3G requirements
boolean setBaudrate(uint16_t baud);


// Battery and ADC
boolean getADCVoltage(uint16_t *v);
boolean getBattPercent(uint16_t *p);
boolean getBattVoltage(uint16_t *v);

// SIM query
uint8_t unlockSIM(char *pin);
uint8_t getSIMCCID(char *ccid);
uint8_t getNetworkStatus(void);
uint8_t getRSSI(void);

// IMEI
uint8_t getIMEI(char *imei);

// SMS handling
boolean setSMSInterrupt(uint8_t i);
uint8_t getSMSInterrupt(void);
int8_t getNumSMS(void);
boolean readSMS(uint8_t i, char *smsbuff, uint16_t max,
   uint16_t *readsize);
boolean sendSMS(char *smsaddr, char *smsmsg);
boolean deleteSMS(uint8_t i);
boolean getSMSSender(uint8_t i, char *sender, int
   senderlen);
boolean sendUSSD(char *ussdmsg, char *ussdbuff, uint16_t
   maxlen, uint16_t *readlen);


// GPRS handling
boolean enableGPRS(boolean onoff);
uint8_t GPRSstate(void);
boolean getGSMLoc(uint16_t *replycode, char *buff,
   uint16_t maxlen);
boolean getGSMLoc(float *lat, float *lon);
void setGPRSNetworkSettings(FONAFlashStringPtr apn,
   FONAFlashStringPtr username=0, FONAFlashStringPtr
   password=0);
```

```


  // TCP raw connections
  boolean TCPconnect(char *server, uint16_t port);
  boolean TCPclose(void);
  boolean TCPconnected(void);
  boolean TCPsend(char *packet, uint8_t len);
  uint16_t TCPavailable(void);
  uint16_t TCPread(uint8_t *buff, uint8_t len);

  // Phone calls
  boolean callPhone(char *phonenum);
  uint8_t getCallStatus(void);
  boolean hangUp(void);
  boolean pickUp(void);
  boolean callerIdNotification(boolean enable, uint8_t
     interrupt = 0);
  boolean incomingCallNumber(char* phonenum);

  // SMTP Mail
  boolean sendTestMail(void);

  // Helper functions to verify responses.
  boolean expectReply(FONAFlashStringPtr reply, uint16_t
     timeout = 10000);
  boolean sendCheckReply(char *send, char *reply, uint16_t
     timeout = FONA_DEFAULT_TIMEOUT_MS);
  boolean sendCheckReply(FONAFlashStringPtr send,
     FONAFlashStringPtr reply, uint16_t timeout =
     FONA_DEFAULT_TIMEOUT_MS);
  boolean sendCheckReply(char* send, FONAFlashStringPtr
     reply, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);


 protected:
  int8_t _rstpin;
  uint8_t _type;

  char replybuffer[255];
  FONAFlashStringPtr apn;
  FONAFlashStringPtr apnusername;
  FONAFlashStringPtr apnpassword;
  boolean httpsredirect;
```

```
FONAFlashStringPtr useragent;
FONAFlashStringPtr ok_reply;

// HTTP helpers
boolean HTTP_setup(char *url);

void flushInput();
uint16_t readRaw(uint16_t b);
uint8_t readline(uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS, boolean multiline = false);
uint8_t getReply(char *send, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);
uint8_t getReply(FONAFlashStringPtr send, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);
uint8_t getReply(FONAFlashStringPtr prefix, char *suffix, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);
uint8_t getReply(FONAFlashStringPtr prefix, int32_t suffix, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);
uint8_t getReply(FONAFlashStringPtr prefix, int32_t suffix1, int32_t suffix2, uint16_t timeout); // Don't set default value or else function call is ambiguous.
uint8_t getReplyQuoted(FONAFlashStringPtr prefix, FONAFlashStringPtr suffix, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);

boolean sendCheckReply(FONAFlashStringPtr prefix, char *suffix, FONAFlashStringPtr reply, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);
boolean sendCheckReply(FONAFlashStringPtr prefix, int32_t suffix, FONAFlashStringPtr reply, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);
boolean sendCheckReply(FONAFlashStringPtr prefix, int32_t suffix, int32_t suffix2, FONAFlashStringPtr reply, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);
boolean sendCheckReplyQuoted(FONAFlashStringPtr prefix, FONAFlashStringPtr suffix, FONAFlashStringPtr reply, uint16_t timeout = FONA_DEFAULT_TIMEOUT_MS);


boolean parseReply(FONAFlashStringPtr toreply,
        uint16_t *v, char divider  = ',', uint8_t index =0);
boolean parseReply(FONAFlashStringPtr toreply,
```

```
                char *v, char divider  = ',', uint8_t index=0);
    boolean parseReplyQuoted(FONAFlashStringPtr toreply,
                char *v, int maxlen, char divider, uint8_t index);

    boolean sendParseReply(FONAFlashStringPtr tosend,
          FONAFlashStringPtr toreply,
          uint16_t *v, char divider = ',', uint8_t index=0);

    static boolean _incomingCall;
    static void onIncomingCall();

    FONAStreamType *mySerial;
};

class Adafruit_FONA_3G : public Adafruit_FONA {

 public:
    Adafruit_FONA_3G (int8_t r) : Adafruit_FONA(r) { _type = FONA3G_A; }

      boolean getBattVoltage(uint16_t *v);
      boolean playToolkitTone(uint8_t t, uint16_t len);
      boolean hangUp(void);
      boolean pickUp(void);
      boolean enableGPRS(boolean onoff);
      boolean enableGPS(boolean onoff);

 protected:
      boolean parseReply(FONAFlashStringPtr toreply,
             float *f, char divider, uint8_t index);

      boolean sendParseReply(FONAFlashStringPtr tosend,
           FONAFlashStringPtr toreply,
           float *f, char divider = ',', uint8_t index=0);
};

#endif
```

C.10 Modified Adafruit FONA Library CPP-File

```
/***************************************************
  This is a library for our Adafruit FONA Cellular Module

  Designed specifically to work with the Adafruit FONA
  ----> http://www.adafruit.com/products/1946
  ----> http://www.adafruit.com/products/1963

  These displays use TTL Serial to communicate, 2 pins are required to
  interface
  Adafruit invests time and resources providing this open source code,
  please support Adafruit and open-source hardware by purchasing
  products from Adafruit!

  Written by Limor Fried/Ladyada for Adafruit Industries.
  BSD license, all text above must be included in any redistribution
 ****************************************************/
    // next line per http://postwarrior.com/arduino-ethershield-error-prog_char-does-not-name-a-type/

#include "Adafruit_FONA.h"




Adafruit_FONA::Adafruit_FONA(int8_t rst)
{
  _rstpin = rst;

  apn = F("FONAnet");
  apnusername = 0;
  apnpassword = 0;
  mySerial = 0;
  httpsredirect = false;
  useragent = F("FONA");
  ok_reply = F("OK");
}

```

```
uint8_t Adafruit_FONA::type(void) {
  return _type;
}

boolean Adafruit_FONA::begin(Stream &port) {
  mySerial = &port;

  pinMode(_rstpin, OUTPUT);
  digitalWrite(_rstpin, HIGH);
  delay(10);
  digitalWrite(_rstpin, LOW);
  delay(100);
  digitalWrite(_rstpin, HIGH);

  DEBUG_PRINTLN(F("Attempting to open comm with ATs"));
  // give 7 seconds to reboot
  int16_t timeout = 7000;

  while (timeout > 0) {
    while (mySerial->available()) mySerial->read();
    if (sendCheckReply(F("AT"), ok_reply))
      break;
    while (mySerial->available()) mySerial->read();
    if (sendCheckReply(F("AT"), F("AT")))
      break;
    delay(500);
    timeout-=500;
  }

  if (timeout <= 0) {
#ifdef ADAFRUIT_FONA_DEBUG
    DEBUG_PRINTLN(F("Timeout: No response to AT... last
       ditch attempt."));
#endif
    sendCheckReply(F("AT"), ok_reply);
    delay(100);
    sendCheckReply(F("AT"), ok_reply);
    delay(100);
    sendCheckReply(F("AT"), ok_reply);
    delay(100);
  }

  // turn off Echo!
```

```
sendCheckReply(F("ATE0"), ok_reply);
delay(100);

if (! sendCheckReply(F("ATE0"), ok_reply)) {
  return false;
}

// turn on hangupitude
sendCheckReply(F("AT+CVHU=0"), ok_reply);

delay(100);
flushInput();


DEBUG_PRINT(F("\t---> ")); DEBUG_PRINTLN("ATI");

mySerial->println("ATI");
readline(500, true);

DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);



if (prog_char_strstr(replybuffer, (prog_char *)F("SIM808 R14")) != 0) {
  _type = FONA808_V2;
} else if (prog_char_strstr(replybuffer, (prog_char *)F("SIM808 R13")) != 0) {
  _type = FONA808_V1;
} else if (prog_char_strstr(replybuffer, (prog_char *)F("SIM800 R13")) != 0) {
  _type = FONA800L;
} else if (prog_char_strstr(replybuffer, (prog_char *)F("SIMCOM_SIM5320A")) != 0) {
  _type = FONA3G_A;
} else if (prog_char_strstr(replybuffer, (prog_char *)F("SIMCOM_SIM5320E")) != 0) {
  _type = FONA3G_E;
}

if (_type == FONA800L) {
  // determine if L or H

```

```
  DEBUG_PRINT(F("\t---> ")); DEBUG_PRINTLN("AT+GMM");

    mySerial->println("AT+GMM");
    readline(500, true);

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);


    if (prog_char_strstr(replybuffer, (prog_char *)F("
       SIM800H")) != 0) {
      _type = FONA800H;
    }
  }

#if defined(FONA_PREF_SMS_STORAGE)
    sendCheckReply(F("AT+CPMS=" FONA_PREF_SMS_STORAGE ","
       FONA_PREF_SMS_STORAGE "," FONA_PREF_SMS_STORAGE),
       ok_reply);
#endif

  return true;
}


/********* BATTERY & ADC
   ********************************************/

/* returns value in mV (uint16_t) */
boolean Adafruit_FONA::getBattVoltage(uint16_t *v) {
  return sendParseReply(F("AT+CBC"), F("+CBC: "), v, ',',
     2);
}

/* returns value in mV (uint16_t) */
boolean Adafruit_FONA_3G::getBattVoltage(uint16_t *v) {
  float f;
  boolean b = sendParseReply(F("AT+CBC"), F("+CBC: "), &f,
     ',', 2);
  *v = f*1000;
  return b;
}

```

```
/* returns the percentage charge of battery as reported by
    sim800 */
boolean Adafruit_FONA::getBattPercent(uint16_t *p) {
  return sendParseReply(F("AT+CBC"), F("+CBC: "), p, ',',
     1);
}

boolean Adafruit_FONA::getADCVoltage(uint16_t *v) {
  return sendParseReply(F("AT+CADC?"), F("+CADC: 1,"), v);
}

/********* SIM
    *********************************************************
    */

uint8_t Adafruit_FONA::unlockSIM(char *pin)
{
  char sendbuff[14] = "AT+CPIN=";
  sendbuff[8] = pin[0];
  sendbuff[9] = pin[1];
  sendbuff[10] = pin[2];
  sendbuff[11] = pin[3];
  sendbuff[12] = '\0';

  return sendCheckReply(sendbuff, ok_reply);
}

uint8_t Adafruit_FONA::getSIMCCID(char *ccid) {
  getReply(F("AT+CCID"));
  // up to 28 chars for reply, 20 char total ccid
  if (replybuffer[0] == '+') {
    // fona 3g?
    strncpy(ccid, replybuffer+8, 20);
  } else {
    // fona 800 or 800
    strncpy(ccid, replybuffer, 20);
  }
  ccid[20] = 0;

  readline(); // eat 'OK'

  return strlen(ccid);
}
```

```
/********* IMEI *********************************************************
*/

uint8_t Adafruit_FONA::getIMEI(char *imei) {
  getReply(F("AT+GSN"));

  // up to 15 chars
  strncpy(imei, replybuffer, 15);
  imei[15] = 0;

  readline(); // eat 'OK'

  return strlen(imei);
}

/********* NETWORK *******************************************************/

uint8_t Adafruit_FONA::getNetworkStatus(void) {
  uint16_t status;

  if (! sendParseReply(F("AT+CREG?"), F("+CREG: "), &status, ',', 1)) return 0;

  return status;
}


uint8_t Adafruit_FONA::getRSSI(void) {
  uint16_t reply;

  if (! sendParseReply(F("AT+CSQ"), F("+CSQ: "), &reply) ) return 0;

  return reply;
}




```

```
/********* CALL PHONES **************************************************/
boolean Adafruit_FONA::callPhone(char *number) {
  char sendbuff[35] = "ATD";
  strncpy(sendbuff+3, number, min(30, strlen(number)));
  uint8_t x = strlen(sendbuff);
  sendbuff[x] = ';';
  sendbuff[x+1] = 0;
  //DEBUG_PRINTLN(sendbuff);

  return sendCheckReply(sendbuff, ok_reply);
}


uint8_t Adafruit_FONA::getCallStatus(void) {
  uint16_t phoneStatus;

  if (! sendParseReply(F("AT+CPAS"), F("+CPAS: "), &phoneStatus))
    return FONA_CALL_FAILED; // 1, since 0 is actually a known, good reply

  return phoneStatus;  // 0 ready, 2 unkown, 3 ringing, 4 call in progress
}

boolean Adafruit_FONA::hangUp(void) {
  return sendCheckReply(F("ATH0"), ok_reply);
}

boolean Adafruit_FONA_3G::hangUp(void) {
  getReply(F("ATH"));

  return (prog_char_strstr(replybuffer, (prog_char *)F("VOICE CALL: END")) != 0);
}

boolean Adafruit_FONA::pickUp(void) {
  return sendCheckReply(F("ATA"), ok_reply);
}

boolean Adafruit_FONA_3G::pickUp(void) {
  return sendCheckReply(F("ATA"), F("VOICE CALL: BEGIN"));
```

```
}


void Adafruit_FONA::onIncomingCall() {

  DEBUG_PRINT(F("> ")); DEBUG_PRINTLN(F("Incoming call..."));

  Adafruit_FONA::_incomingCall = true;
}

boolean Adafruit_FONA::_incomingCall = false;

boolean Adafruit_FONA::callerIdNotification(boolean enable, uint8_t interrupt) {
  if(enable){
    attachInterrupt(interrupt, onIncomingCall, FALLING);
    return sendCheckReply(F("AT+CLIP=1"), ok_reply);
  }

  detachInterrupt(interrupt);
  return sendCheckReply(F("AT+CLIP=0"), ok_reply);
}

boolean Adafruit_FONA::incomingCallNumber(char* phonenum) {
  //+CLIP: "<incoming phone number>",145,"",0,"",0
  if(!Adafruit_FONA::_incomingCall)
    return false;

  readline();
  while(!prog_char_strcmp(replybuffer, (prog_char*)F("RING")) == 0) {
    flushInput();
    readline();
  }

  readline(); //reads incoming phone number line

  parseReply(F("+CLIP: \""), phonenum, '"');


  DEBUG_PRINT(F("Phone Number: "));
  DEBUG_PRINTLN(replybuffer);
```

```


  Adafruit_FONA::_incomingCall = false;
  return true;
}

/********* SMS ****************************************************
   */

uint8_t Adafruit_FONA::getSMSInterrupt(void) {
  uint16_t reply;

  if (! sendParseReply(F("AT+CFGRI?"), F("+CFGRI: "), &
     reply) ) return 0;

  return reply;
}

boolean Adafruit_FONA::setSMSInterrupt(uint8_t i) {
  return sendCheckReply(F("AT+CFGRI="), i, ok_reply);
}

int8_t Adafruit_FONA::getNumSMS(void) {
  uint16_t numsms;

  // get into text mode
  if (! sendCheckReply(F("AT+CMGF=1"), ok_reply)) return
     -1;

  // ask how many sms are stored
  if (sendParseReply(F("AT+CPMS?"), F(FONA_PREF_SMS_STORAGE
     ","), &numsms))
    return numsms;
  if (sendParseReply(F("AT+CPMS?"), F("\"SM\","), &numsms))
    return numsms;
  if (sendParseReply(F("AT+CPMS?"), F("\"SM_P\","), &numsms
     ))
    return numsms;
  return -1;
}

```

```
// Reading SMS's is a bit involved so we don't use helpers
    that may cause delays or debug
// printouts!
boolean Adafruit_FONA::readSMS(uint8_t i, char *smsbuff,
                         uint16_t maxlen, uint16_t *readlen) {
  // text mode
  if (! sendCheckReply(F("AT+CMGF=1"), ok_reply)) return
     false;

  // show all text mode parameters
  if (! sendCheckReply(F("AT+CSDH=1"), ok_reply)) return
     false;

  // parse out the SMS len
  uint16_t thesmslen = 0;


  DEBUG_PRINT(F("AT+CMGR="));
  DEBUG_PRINTLN(i);


  //getReply(F("AT+CMGR="), i, 1000);  //  do not print
     debug!
  mySerial->print(F("AT+CMGR="));
  mySerial->println(i);
  readline(1000); // timeout

  //DEBUG_PRINT(F("Reply: ")); DEBUG_PRINTLN(replybuffer);
  // parse it out...


  DEBUG_PRINTLN(replybuffer);


  if (! parseReply(F("+CMGR:"), &thesmslen, ',', 11)) {
    *readlen = 0;
    return false;
  }

  readRaw(thesmslen);

  flushInput();

```

```
  uint16_t thelen = min(maxlen, strlen(replybuffer));
  strncpy(smsbuff, replybuffer, thelen);
  smsbuff[thelen] = 0; // end the string


  DEBUG_PRINTLN(replybuffer);

  *readlen = thelen;
  return true;
}

// Retrieve the sender of the specified SMS message and copy it as a string to
// the sender buffer.  Up to senderlen characters of the sender will be copied
// and a null terminator will be added if less than senderlen charactesr are
// copied to the result.  Returns true if a result was successfully retrieved,
// otherwise false.
boolean Adafruit_FONA::getSMSSender(uint8_t i, char *sender, int senderlen) {
  // Ensure text mode and all text mode parameters are sent.
  if (! sendCheckReply(F("AT+CMGF=1"), ok_reply)) return false;
  if (! sendCheckReply(F("AT+CSDH=1"), ok_reply)) return false;


  DEBUG_PRINT(F("AT+CMGR="));
  DEBUG_PRINTLN(i);


  // Send command to retrieve SMS message and parse a line of response.
  mySerial->print(F("AT+CMGR="));
  mySerial->println(i);
  readline(1000);


  DEBUG_PRINTLN(replybuffer);

```

```
  // Parse the second field in the response.
  boolean result = parseReplyQuoted(F("+CMGR:"), sender,
     senderlen, ',', 1);
  // Drop any remaining data from the response.
  flushInput();
  return result;
}

boolean Adafruit_FONA::sendSMS(char *smsaddr, char *smsmsg)
    {
  if (! sendCheckReply(F("AT+CMGF=1"), ok_reply)) return
     false;

  char sendcmd[30] = "AT+CMGS=\"";
  strncpy(sendcmd+9, smsaddr, 30-9-2);  // 9 bytes
     beginning, 2 bytes for close quote + null
  sendcmd[strlen(sendcmd)] = '\"';

  if (! sendCheckReply(sendcmd, F("> "))) return false;

  DEBUG_PRINT(F("> ")); DEBUG_PRINTLN(smsmsg);

  mySerial->println(smsmsg);
  mySerial->println();
  mySerial->write(0x1A);

  DEBUG_PRINTLN("^Z");

  if ( (_type == FONA3G_A) || (_type == FONA3G_E) ) {
    // Eat two sets of CRLF
    readline(200);
    //DEBUG_PRINT("Line 1: "); DEBUG_PRINTLN(strlen(
       replybuffer));
    readline(200);
    //DEBUG_PRINT("Line 2: "); DEBUG_PRINTLN(strlen(
       replybuffer));
  }
  readline(10000); // read the +CMGS reply, wait up to 10
     seconds!!!
  //DEBUG_PRINT("Line 3: "); DEBUG_PRINTLN(strlen(
     replybuffer));
  if (strstr(replybuffer, "+CMGS") == 0) {
```

```
    return false;
  }
  readline(1000); // read OK
  //DEBUG_PRINT("* "); DEBUG_PRINTLN(replybuffer);

  if (strcmp(replybuffer, "OK") != 0) {
    return false;
  }

  return true;
}


boolean Adafruit_FONA::deleteSMS(uint8_t i) {
    if (! sendCheckReply(F("AT+CMGF=1"), ok_reply)) return
       false;
  // read an sms
  char sendbuff[12] = "AT+CMGD=000";
  sendbuff[8] = (i / 100) + '0';
  i %= 100;
  sendbuff[9] = (i / 10) + '0';
  i %= 10;
  sendbuff[10] = i + '0';

  return sendCheckReply(sendbuff, ok_reply, 2000);
}


/********* GPRS
   **********************************************************
   */


boolean Adafruit_FONA::enableGPRS(boolean onoff) {

  if (onoff) {
    // disconnect all sockets
    // sendCheckReply(F("AT+CIPSHUT"), F("SHUT OK"), 20000)
       ;

    if (! sendCheckReply(F("AT+CGATT=1"), ok_reply, 10000))
      return false;

```

```
    // set bearer profile! connection type GPRS
    if (! sendCheckReply(F("AT+SAPBR=3,1,\"CONTYPE\",\"GPRS
       \""),
          ok_reply, 10000))
      return false;

    // set bearer profile access point name
    if (apn) {
      // Send command AT+SAPBR=3,1,"APN","<apn value>"
         where <apn value> is the configured APN value.
      if (! sendCheckReplyQuoted(F("AT+SAPBR=3,1,\"APN\","
         , apn, ok_reply, 10000))
        return false;

      // send AT+CSTT,"apn","user","pass"
      flushInput();

      mySerial->print(F("AT+CSTT=\""));
      mySerial->print(apn);
      if (apnusername) {
  mySerial->print("\",\"");
  mySerial->print(apnusername);
      }
      if (apnpassword) {
  mySerial->print("\",\"");
  mySerial->print(apnpassword);
      }
      mySerial->println("\"");

      DEBUG_PRINT(F("\t---> ")); DEBUG_PRINT(F("AT+CSTT=\""
         ));
      DEBUG_PRINT(apn);

      if (apnusername) {
  DEBUG_PRINT("\",\"");
  DEBUG_PRINT(apnusername);
      }
      if (apnpassword) {
  DEBUG_PRINT("\",\"");
  DEBUG_PRINT(apnpassword);
      }
      DEBUG_PRINTLN("\"");

```

```
      if (! expectReply(ok_reply)) return false;

      // set username/password
      if (apnusername) {
        // Send command AT+SAPBR=3,1,"USER","<user>" where <user> is the configured APN username.
        if (! sendCheckReplyQuoted(F("AT+SAPBR=3,1,\"USER\","), apnusername, ok_reply, 10000))
          return false;
      }
      if (apnpassword) {
        // Send command AT+SAPBR=3,1,"PWD","<password>" where <password> is the configured APN password.
        if (! sendCheckReplyQuoted(F("AT+SAPBR=3,1,\"PWD\","), apnpassword, ok_reply, 10000))
          return false;
      }
    }

    // open GPRS context
    if (! sendCheckReply(F("AT+SAPBR=1,1"), ok_reply, 30000))
      return false;

    // bring up wireless connection
    if (! sendCheckReply(F("AT+CIICR"), ok_reply, 10000))
      return false;

  } else {
    // disconnect all sockets
    if (! sendCheckReply(F("AT+CIPSHUT"), F("SHUT OK"), 20000))
      return false;

    // close GPRS context
    if (! sendCheckReply(F("AT+SAPBR=0,1"), ok_reply, 10000))
      return false;

    if (! sendCheckReply(F("AT+CGATT=0"), ok_reply, 10000))
      return false;

  }
```

```
  return true;
}

boolean Adafruit_FONA_3G::enableGPRS(boolean onoff) {

  if (onoff) {
    // disconnect all sockets
    //sendCheckReply(F("AT+CIPSHUT"), F("SHUT OK"), 5000);

    if (! sendCheckReply(F("AT+CGATT=1"), ok_reply, 10000))
      return false;


    // set bearer profile access point name
    if (apn) {
      // Send command AT+CGSOCKCONT=1,"IP","<apn value>"
         where <apn value> is the configured APN name.
      if (! sendCheckReplyQuoted(F("AT+CGSOCKCONT=1,\"IP\",
         "), apn, ok_reply, 10000))
        return false;

      // set username/password
      if (apnusername) {
  char authstring[100] = "AT+CGAUTH=1,1,\"";
  char *strp = authstring + strlen(authstring);
  prog_char_strcpy(strp, (prog_char *)apnusername);
  strp+=prog_char_strlen((prog_char *)apnusername);
  strp[0] = '\"';
  strp++;
  strp[0] = 0;

  if (apnpassword) {
    strp[0] = ','; strp++;
    strp[0] = '\"'; strp++;
    prog_char_strcpy(strp, (prog_char *)apnpassword);
    strp+=prog_char_strlen((prog_char *)apnpassword);
    strp[0] = '\"';
    strp++;
    strp[0] = 0;
  }

  if (! sendCheckReply(authstring, ok_reply, 10000))
    return false;
```

```
      }
    }

    // connect in transparent
    if (! sendCheckReply(F("AT+CIPMODE=1"), ok_reply, 10000))
      return false;
    // open network (?)
    if (! sendCheckReply(F("AT+NETOPEN=,,1"), F("Network opened"), 10000))
      return false;

    readline(); // eat 'OK'
  } else {
    // close GPRS context
    if (! sendCheckReply(F("AT+NETCLOSE"), F("Network closed"), 10000))
      return false;

    readline(); // eat 'OK'
  }

  return true;
}

uint8_t Adafruit_FONA::GPRSstate(void) {
  uint16_t state;

  if (! sendParseReply(F("AT+CGATT?"), F("+CGATT: "), &state) )
    return -1;

  return state;
}

void Adafruit_FONA::setGPRSNetworkSettings(FONAFlashStringPtr apn,
              FONAFlashStringPtr username, FONAFlashStringPtr password) {
  this->apn = apn;
  this->apnusername = username;
  this->apnpassword = password;
}
```

```

boolean Adafruit_FONA::getGSMLoc(uint16_t *errorcode, char
    *buff, uint16_t maxlen) {

  getReply(F("AT+CIPGSMLOC=1,1"), (uint16_t)10000);

  if (! parseReply(F("+CIPGSMLOC: "), errorcode))
    return false;

  char *p = replybuffer+14;
  uint16_t lentocopy = min(maxlen-1, strlen(p));
  strncpy(buff, p, lentocopy+1);

  readline(); // eat OK

  return true;
}

boolean Adafruit_FONA::getGSMLoc(float *lat, float *lon) {

  uint16_t returncode;
  char gpsbuffer[120];

  // make sure we could get a response
  if (! getGSMLoc(&returncode, gpsbuffer, 120))
    return false;

  // make sure we have a valid return code
  if (returncode != 0)
    return false;

  // +CIPGSMLOC: 0,-74.007729,40.730160,2015/10/15,19:24:55
  // tokenize the gps buffer to locate the lat & long
  char *longp = strtok(gpsbuffer, ",");
  if (! longp) return false;

  char *latp = strtok(NULL, ",");
  if (! latp) return false;

  *lat = atof(latp);
  *lon = atof(longp);

  return true;
```

```

}
/********* TCP FUNCTIONS ************************************/


boolean Adafruit_FONA::TCPconnect(char *server, uint16_t port) {
  flushInput();

  // close all old connections
  if (! sendCheckReply(F("AT+CIPSHUT"), F("SHUT OK"), 20000) ) return false;

  // single connection at a time
  if (! sendCheckReply(F("AT+CIPMUX=0"), ok_reply) ) return false;

  // manually read data
  if (! sendCheckReply(F("AT+CIPRXGET=1"), ok_reply) ) return false;


  DEBUG_PRINT(F("AT+CIPSTART=\"TCP\",\""));
  DEBUG_PRINT(server);
  DEBUG_PRINT(F("\",\""));
  DEBUG_PRINT(port);
  DEBUG_PRINTLN(F("\""));


  mySerial->print(F("AT+CIPSTART=\"TCP\",\""));
  mySerial->print(server);
  mySerial->print(F("\",\""));
  mySerial->print(port);
  mySerial->println(F("\""));

  if (! expectReply(ok_reply)) return false;
  if (! expectReply(F("CONNECT OK"))) return false;

  // looks like it was a success (?)
  return true;
}

```

```
boolean Adafruit_FONA::TCPclose(void) {
  return sendCheckReply(F("AT+CIPCLOSE"), ok_reply);
}

boolean Adafruit_FONA::TCPconnected(void) {
  if (! sendCheckReply(F("AT+CIPSTATUS"), ok_reply, 100) ) return false;
  readline(100);

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  return (strcmp(replybuffer, "STATE: CONNECT OK") == 0);
}

boolean Adafruit_FONA::TCPsend(char *packet, uint8_t len) {

  DEBUG_PRINT(F("AT+CIPSEND="));
  DEBUG_PRINTLN(len);
#ifdef ADAFRUIT_FONA_DEBUG
  for (uint16_t i=0; i<len; i++) {
  DEBUG_PRINT(F(" 0x"));
  DEBUG_PRINT(packet[i], HEX);
  }
#endif
  DEBUG_PRINTLN();


  mySerial->print(F("AT+CIPSEND="));
  mySerial->println(len);
  readline();

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  if (replybuffer[0] != '>') return false;

  mySerial->write(packet, len);
  readline(3000); // wait up to 3 seconds to send the data

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);


  return (strcmp(replybuffer, "SEND OK") == 0);
}
```

```

uint16_t Adafruit_FONA::TCPavailable(void) {
  uint16_t avail;

  if (! sendParseReply(F("AT+CIPRXGET=4"), F("+CIPRXGET: 4,"), &avail, ',', 0) ) return false;


  DEBUG_PRINT (avail); DEBUG_PRINTLN(F(" bytes available"));


  return avail;
}


uint16_t Adafruit_FONA::TCPread(uint8_t *buff, uint8_t len) {
  uint16_t avail;

  mySerial->print(F("AT+CIPRXGET=2,"));
  mySerial->println(len);
  readline();
  if (! parseReply(F("+CIPRXGET: 2,"), &avail, ',', 0)) return false;

  readRaw(avail);

#ifdef ADAFRUIT_FONA_DEBUG
  DEBUG_PRINT (avail); DEBUG_PRINTLN(F(" bytes read"));
  for (uint8_t i=0;i<avail;i++) {
  DEBUG_PRINT(F(" 0x")); DEBUG_PRINT(replybuffer[i], HEX);
  }
  DEBUG_PRINTLN();
#endif

  memcpy(buff, replybuffer, avail);

  return avail;
}


```

```
/********* SMTP Mail ********************************************/

boolean Adafruit_FONA::sendTestMail() {
  if (!sendCheckReply(F("AT+CSMTPSSRV=\"smtp.mail.yahoo.com\",587,2"), ok_reply)) // ==> returns OK
    return false;

    if (!sendCheckReply(F("AT+CSMTPSAUTH=1,\"joshuapidgeon@yahoo.com.au\   ,\ password\      ), ok_reply)) // ==> returns OK
    return false;

    if (!sendCheckReply(F("AT+CSMTPSFROM=\"joshuapidgeon@yahoo.com.au\",\"Joshua Pidgeon\""), ok_reply)) // ==> returns OK
    return false;

    if (!sendCheckReply(F("AT+CSMTPSRCPT=0,0,\ joshua.pidgeon@students.mq.edu.au\   ,\ Josh MQ\      ), ok_reply)) // ==> returns OK
    return false;

    if (!sendCheckReply(F("AT+CSMTPSSUB=5,\"utf-8\"\r\n\"FONA email\""), ">")) // ==> returns OK
    return false;

    if (!sendCheckReply(F("AT+CSMTPSBODY=16\r\n\"Test email from Fona for Josh Pidgeon\""), ok_reply)) // ==> returns OK
    return false;

    if (!sendCheckReply(F("AT+CSMTPSSEND"), ok_reply)) // ==> returns ok
    return false;

    return true;
}


/********* HELPERS ***********************************************/

```

```
boolean Adafruit_FONA::expectReply(FONAFlashStringPtr reply
    ,
                                    uint16_t timeout) {
  readline(timeout);

  DEBUG_PRINT(F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  return (prog_char_strcmp(replybuffer, (prog_char*)reply)
     == 0);
}

/********* LOW LEVEL
    *******************************************/

inline int Adafruit_FONA::available(void) {
  return mySerial->available();
}

inline size_t Adafruit_FONA::write(uint8_t x) {
  return mySerial->write(x);
}

inline int Adafruit_FONA::read(void) {
  return mySerial->read();
}

inline int Adafruit_FONA::peek(void) {
  return mySerial->peek();
}

inline void Adafruit_FONA::flush() {
  mySerial->flush();
}

void Adafruit_FONA::flushInput() {
    // Read all available serial input to flush pending
       data.
    uint16_t timeoutloop = 0;
    while (timeoutloop++ < 40) {
        while(available()) {
            read();
            timeoutloop = 0;  // If char was received reset
                the timer
```

```
        }
        delay(1);
    }
}

uint16_t Adafruit_FONA::readRaw(uint16_t b) {
  uint16_t idx = 0;

  while (b && (idx < sizeof(replybuffer)-1)) {
    if (mySerial->available()) {
      replybuffer[idx] = mySerial->read();
      idx++;
      b--;
    }
  }
  replybuffer[idx] = 0;

  return idx;
}

uint8_t Adafruit_FONA::readline(uint16_t timeout, boolean
    multiline) {
  uint16_t replyidx = 0;

  while (timeout--) {
    if (replyidx >= 254) {
      //DEBUG_PRINTLN(F("SPACE"));
      break;
    }

    while(mySerial->available()) {
      char c =  mySerial->read();
      if (c == '\r') continue;
      if (c == 0xA) {
        if (replyidx == 0)   // the first 0x0A is ignored
          continue;

        if (!multiline) {
          timeout = 0;         // the second 0x0A is the
             end of the line
          break;
        }
      }
```

```
      replybuffer[replyidx] = c;
      //DEBUG_PRINT(c, HEX); DEBUG_PRINT("#"); DEBUG_PRINTLN(c);
      replyidx++;
    }

    if (timeout == 0) {
      //DEBUG_PRINTLN(F("TIMEOUT"));
      break;
    }
    delay(1);
  }
  replybuffer[replyidx] = 0;  // null term
  return replyidx;
}

uint8_t Adafruit_FONA::getReply(char *send, uint16_t timeout) {
  flushInput();


  DEBUG_PRINT(F("\t---> ")); DEBUG_PRINTLN(send);


  mySerial->println(send);

  uint8_t l = readline(timeout);

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  return l;
}

uint8_t Adafruit_FONA::getReply(FONAFlashStringPtr send, uint16_t timeout) {
  flushInput();


  DEBUG_PRINT(F("\t---> ")); DEBUG_PRINTLN(send);


  mySerial->println(send);

```

```
  uint8_t l = readline(timeout);

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  return l;
}

// Send prefix, suffix, and newline. Return response (and also set replybuffer with response).
uint8_t Adafruit_FONA::getReply(FONAFlashStringPtr prefix, char *suffix, uint16_t timeout) {
  flushInput();


  DEBUG_PRINT(F("\t---> ")); DEBUG_PRINT(prefix); DEBUG_PRINTLN(suffix);


  mySerial->print(prefix);
  mySerial->println(suffix);

  uint8_t l = readline(timeout);

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  return l;
}

// Send prefix, suffix, and newline. Return response (and also set replybuffer with response).
uint8_t Adafruit_FONA::getReply(FONAFlashStringPtr prefix, int32_t suffix, uint16_t timeout) {
  flushInput();


  DEBUG_PRINT(F("\t---> ")); DEBUG_PRINT(prefix); DEBUG_PRINTLN(suffix, DEC);


  mySerial->print(prefix);
  mySerial->println(suffix, DEC);

  uint8_t l = readline(timeout);
```

```

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  return l;
}

// Send prefix, suffix, suffix2, and newline. Return
    response (and also set replybuffer with response).
uint8_t Adafruit_FONA::getReply(FONAFlashStringPtr prefix,
    int32_t suffix1, int32_t suffix2, uint16_t timeout) {
  flushInput();


  DEBUG_PRINT(F("\t---> ")); DEBUG_PRINT(prefix);
  DEBUG_PRINT(suffix1, DEC); DEBUG_PRINT(',');
     DEBUG_PRINTLN(suffix2, DEC);


  mySerial->print(prefix);
  mySerial->print(suffix1);
  mySerial->print(',');
  mySerial->println(suffix2, DEC);

  uint8_t l = readline(timeout);

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  return l;
}

// Send prefix, ", suffix, ", and newline. Return response
    (and also set replybuffer with response).
uint8_t Adafruit_FONA::getReplyQuoted(FONAFlashStringPtr
    prefix, FONAFlashStringPtr suffix, uint16_t timeout) {
  flushInput();


  DEBUG_PRINT(F("\t---> ")); DEBUG_PRINT(prefix);
  DEBUG_PRINT('"'); DEBUG_PRINT(suffix); DEBUG_PRINTLN('"')
     ;


  mySerial->print(prefix);
```

```
  mySerial->print('"');
  mySerial->print(suffix);
  mySerial->println('"');

  uint8_t l = readline(timeout);

  DEBUG_PRINT (F("\t<--- ")); DEBUG_PRINTLN(replybuffer);

  return l;
}

boolean Adafruit_FONA::sendCheckReply(char *send, char *
    reply, uint16_t timeout) {
  if (! getReply(send, timeout) )
    return false;
/*
  for (uint8_t i=0; i<strlen(replybuffer); i++) {
  DEBUG_PRINT(replybuffer[i], HEX); DEBUG_PRINT(" ");
  }
  DEBUG_PRINTLN();
  for (uint8_t i=0; i<strlen(reply); i++) {
    DEBUG_PRINT(reply[i], HEX); DEBUG_PRINT(" ");
  }
  DEBUG_PRINTLN();
  */
  return (strcmp(replybuffer, reply) == 0);
}

boolean Adafruit_FONA::sendCheckReply(FONAFlashStringPtr
    send, FONAFlashStringPtr reply, uint16_t timeout) {
  if (! getReply(send, timeout) )
    return false;

  return (prog_char_strcmp(replybuffer, (prog_char*)reply)
      == 0);
}

boolean Adafruit_FONA::sendCheckReply(char* send,
    FONAFlashStringPtr reply, uint16_t timeout) {
  if (! getReply(send, timeout) )
    return false;
  return (prog_char_strcmp(replybuffer, (prog_char*)reply)
      == 0);
```

```
}


// Send prefix, suffix, and newline.  Verify FONA response matches reply parameter.
boolean Adafruit_FONA::sendCheckReply(FONAFlashStringPtr prefix, char *suffix, FONAFlashStringPtr reply, uint16_t timeout) {
  getReply(prefix, suffix, timeout);
  return (prog_char_strcmp(replybuffer, (prog_char*)reply) == 0);
}

// Send prefix, suffix, and newline.  Verify FONA response matches reply parameter.
boolean Adafruit_FONA::sendCheckReply(FONAFlashStringPtr prefix, int32_t suffix, FONAFlashStringPtr reply, uint16_t timeout) {
  getReply(prefix, suffix, timeout);
  return (prog_char_strcmp(replybuffer, (prog_char*)reply) == 0);
}

// Send prefix, suffix, suffix2, and newline.  Verify FONA response matches reply parameter.
boolean Adafruit_FONA::sendCheckReply(FONAFlashStringPtr prefix, int32_t suffix1, int32_t suffix2, FONAFlashStringPtr reply, uint16_t timeout) {
  getReply(prefix, suffix1, suffix2, timeout);
  return (prog_char_strcmp(replybuffer, (prog_char*)reply) == 0);
}

// Send prefix, ", suffix, ", and newline.  Verify FONA response matches reply parameter.
boolean Adafruit_FONA::sendCheckReplyQuoted(FONAFlashStringPtr prefix, FONAFlashStringPtr suffix, FONAFlashStringPtr reply, uint16_t timeout) {
  getReplyQuoted(prefix, suffix, timeout);
  return (prog_char_strcmp(replybuffer, (prog_char*)reply) == 0);
}

```

```

boolean Adafruit_FONA::parseReply(FONAFlashStringPtr
    toreply,
          uint16_t *v, char divider, uint8_t index) {
  char *p = prog_char_strstr(replybuffer, (prog_char*)
     toreply);  // get the pointer to the voltage
  if (p == 0) return false;
  p+=prog_char_strlen((prog_char*)toreply);
  //DEBUG_PRINTLN(p);
  for (uint8_t i=0; i<index;i++) {
    // increment dividers
    p = strchr(p, divider);
    if (!p) return false;
    p++;
    //DEBUG_PRINTLN(p);

  }
  *v = atoi(p);

  return true;
}

boolean Adafruit_FONA::parseReply(FONAFlashStringPtr
    toreply,
          char *v, char divider, uint8_t index) {
  uint8_t i=0;
  char *p = prog_char_strstr(replybuffer, (prog_char*)
     toreply);
  if (p == 0) return false;
  p+=prog_char_strlen((prog_char*)toreply);

  for (i=0; i<index;i++) {
    // increment dividers
    p = strchr(p, divider);
    if (!p) return false;
    p++;
  }

  for(i=0; i<strlen(p);i++) {
    if(p[i] == divider)
      break;
    v[i] = p[i];
  }
```

```
  v[i] = '\0';

  return true;
}

// Parse a quoted string in the response fields and copy its value (without quotes)
// to the specified character array (v).  Only up to maxlen characters are copied
// into the result buffer, so make sure to pass a large enough buffer to handle the
// response.
boolean Adafruit_FONA::parseReplyQuoted(FONAFlashStringPtr toreply,
          char *v, int maxlen, char divider, uint8_t index) {
  uint8_t i=0, j;
  // Verify response starts with toreply.
  char *p = prog_char_strstr(replybuffer, (prog_char*)toreply);
  if (p == 0) return false;
  p+=prog_char_strlen((prog_char*)toreply);

  // Find location of desired response field.
  for (i=0; i<index;i++) {
    // increment dividers
    p = strchr(p, divider);
    if (!p) return false;
    p++;
  }

  // Copy characters from response field into result string.
  for(i=0, j=0; j<maxlen && i<strlen(p); ++i) {
    // Stop if a divier is found.
    if(p[i] == divider)
      break;
    // Skip any quotation marks.
    else if(p[i] == '"')
      continue;
    v[j++] = p[i];
  }
```

```

  // Add a null terminator if result string buffer was not filled.
  if (j < maxlen)
    v[j] = '\0';

  return true;
}

boolean Adafruit_FONA::sendParseReply(FONAFlashStringPtr tosend,
                  FONAFlashStringPtr toreply,
                  uint16_t *v, char divider, uint8_t index) {
  getReply(tosend);

  if (! parseReply(toreply, v, divider, index)) return false;

  readline(); // eat 'OK'

  return true;
}


// needed for CBC and others

boolean Adafruit_FONA_3G::sendParseReply(FONAFlashStringPtr tosend,
                  FONAFlashStringPtr toreply,
                  float *f, char divider, uint8_t index) {
  getReply(tosend);

  if (! parseReply(toreply, f, divider, index)) return false;

  readline(); // eat 'OK'

  return true;
}


boolean Adafruit_FONA_3G::parseReply(FONAFlashStringPtr toreply,
```

```
                float *f, char divider, uint8_t index) {
  char *p = prog_char_strstr(replybuffer, (prog_char*)
     toreply);  // get the pointer to the voltage
  if (p == 0) return false;
  p+=prog_char_strlen((prog_char*)toreply);
  //DEBUG_PRINTLN(p);
  for (uint8_t i=0; i<index;i++) {
    // increment dividers
    p = strchr(p, divider);
    if (!p) return false;
    p++;
    //DEBUG_PRINTLN(p);

  }
  *f = atof(p);

  return true;
}
```

C.11 Redundant Camera And SD

```
// Declare pins
int pFixedId = A0; // Pin for input of fixed ID
int pClearOut = A1; // Pin for output of clearance flag
int pLockOut = A2; // Pin for output of lock flag
int pProcOut = A3; // Pin for output of processing flag
int pLockIn = A4; // Pin for input of lock flag
int pClearIn = 2; // Pin for input of clearance flag
int pProcIn = 3; // Pin for input of processing flag
int pPrimeLED = 9; // Pin for debugging LED

// Declare exclusion lock variables
int fixedId = 0; // Fixed ID
boolean clearance = false; // Clearance flag
boolean currPrime = false; // Current primary MCU flag
long lastClear = 0; // Time of last clearance
int extraDel = 0; // Delay of former primary in ms
boolean debug; // Stored value of debugging button input

// Declare exclusion lock variables as volatile that will
// be modified by the interrupt subroutine
// Current time taken to process
volatile long elapsedProc = 0;
// Flag to start new processing
volatile boolean newProc = false;
// Flag for returning to the start of the main loop
volatile boolean startReturn = false;
volatile boolean locked = false; // Lock flag
volatile boolean processing = false; // Processing flag

// Declare watchdog timer (WDT) components
#include <avr/wdt.h> // Include WDT library
// Flag for disabling main loop after the WDT has timed out
boolean wdtOverRide = false;

// Declare included libraries
#include <Adafruit_VC0706.h> // Camera
#include <SPI.h> // Serial peripheral interface
#include <SD.h> // SD card
#include <SoftwareSerial.h> // Additional serial port

// Declare new serial ports for camera, (RX, TX)
```

```
SoftwareSerial cameraconnection = SoftwareSerial(7, 8);
// Declare camera object
Adafruit_VC0706 cam = Adafruit_VC0706(&cameraconnection);


// Declare other pins
int pChipSelect = 10; // Pin for SD chip select
int pButton = 7; // Pin for camera trigger button

// Declare system function variables
File imgFile; // Variable for image storage transfer
bool buttonPressed = false; // Variable button result
int maxProc = 8000; // Maximum processing time in ms

void setup() {
  // Initialise pins
  pinMode(pFixedId, INPUT); // Input of fixed ID voltage
  pinMode(pClearOut, OUTPUT); // Output of clearance flag
  pinMode(pLockOut, OUTPUT); // Output of lock flag
  pinMode(pProcOut, OUTPUT); // Output of processing flag
  pinMode(pLockIn, INPUT); // Input of lock flag
  pinMode(pClearIn, INPUT); // Input of clearance flag
  pinMode(pProcIn, INPUT); // Input of processing flag
  pinMode(pPrimeLED, OUTPUT); // Output of primary flag LED
  // Map fixed ID input voltage to integer value
  fixedId = int(round(analogRead(pFixedId) * 3.0 / 1024));
  // Attach clearance/trigger interrupt subroutine (ISR)
  // to digital pin 2, interrupt 0
  attachInterrupt(0, clear_ISR, CHANGE);
  watchdogSetup(); // Initialise WDT

  // Initialise camera & SD pins
  pinMode(pChipSelect, OUTPUT); // SS on Uno, etc.

  // Initialise serial connection with computer
  Serial.begin(9600);
  // Notify of start of testing camera system
  Serial.println("VC0706 Camera test");

  // Test SD card connection
  if (SD.begin(pChipSelect)) {
    Serial.println("Card found");
  } else {
```

```
    Serial.println("Card not found");
    return; // Abandon attempt
  }

  wdt_reset(); // Reset WDT

  // Test camera connection
  if (cam.begin()) {
    Serial.println("Camera found");
  } else {
    Serial.println("Camera not found");
    return; // Abandon attempt
  }
  wdt_reset(); // Reset WDT

  // Select desired image size
  //cam.setImageSize(VC0706_640x480); // Biggest
  cam.setImageSize(VC0706_320x240); // Medium
  //cam.setImageSize(VC0706_160x120); // Small

  // Initialise trigger button input
  pinMode(pButton, INPUT);
}

// ISR for clearance flag
void clear_ISR() {
  // Calculate time elapsed since last effective clearance
  elapsedProc = millis() - lastClear;
  locked = digitalRead(pLockIn); // Read lock flag
  // Read processing flag
  processing = digitalRead(pProcIn);
  if (currPrime && elapsedProc > maxProc) {
    // If currently processing but longer than expected
    newProc = true; // Set flag of new process
    startReturn = true;
    digitalWrite(pLockOut, false); // Clear lock flag
  } else if (processing && !locked) {
    // Else if currently processing with clear lock flag
    newProc = true; // Set flag of new process
    // Set flag to return to start of main loop
    startReturn = true;
  } else if (processing && elapsedProc < maxProc) {
    // Else if currently processing within expected time
```

```
      ; // Don't change
      newProc = false; // Clear flag of new process
    } else if (currPrime) { // Else if current primary
      // Else if currently the primary MCU
      newProc = true; // Set flag of new process
      // Set flag to return to start of main loop
      startReturn = true;
      digitalWrite(pLockOut, false); // Clear lock flag
    } else {
      newProc = true; // Set flag of new process
      // Set flag to return to start of main loop
      startReturn = true;
    }
    // Perform new process section in loop to avoid having
    // delays in the interrupt subroutine.
  }


  void loop() {
    // If WDT override flag is set, disable function of
    // main loop
    if (wdtOverRide) {
      delay(10); // Brief delay
      if (currPrime) { // If currently the prime,
        // If currently the prime, prompt a possible primary
        // reassignment
        digitalWrite(pClearOut, true); // Set clearance flag
      }
      if (processing) {
        // If the processing flag is set, maintain flag to
        // hand over to next available MCU
        digitalWrite(pProcOut, true); // Set processing flag
      }
      digitalWrite(pClearOut, false); // Clear clearance flag
      // Return to main loop start, to prevent interference
      // with main system function
      return;
    }
    // Clear flag to return to start of main loop
    startReturn = false;
    if (newProc) {
      // If new process, assign/reassign primary priority
      newProc = false; // Clear new process flag
```

```

    if (currPrime) { // If current primary
      // Delay 100ms due to being previous primary
      extraDel = 100;
      currPrime = false; // Clear current primary flag
    } else { // Otherwise
      // Delay 0ms due to not being previous primary
      extraDel = 0;
    }

    digitalWrite(pProcOut, true); // set processing flag
    delay(extraDel); // Wait for any offset delay
    // Delay by a scalar of fixed ID, reduced by 1 to
    // prevent unnecessary delays
    delay((fixedId - 1) * 30);
    // Record time of start of processing
    lastClear = millis();
    locked = digitalRead(pLockIn); // Read lock flag
    if (!locked) { // If not locked, take on primary role
      digitalWrite(pLockOut, true); // Set lock flag
      locked = true; // Set internal lock flag
      currPrime = true; // Set current primary flag
    } else { // If locked already, remain as non-primary
      // Clear processing flag
      digitalWrite(pProcOut, false);
    }
  }


  // Update elapsed time since last effective clearance
  elapsedProc = millis() - lastClear;
  locked = digitalRead(pLockIn); // Read lock flag
  // Read processing flag
  processing = digitalRead(pProcIn);

  //  Serial.print(fixedId);
  //  Serial.print('\t');
  //  Serial.print(locked);
  //  Serial.print(processing);
  //  Serial.print(currPrime);
  //  Serial.print('\t');
  //  Serial.print(lapsedProc);
  //  Serial.print('\t');
```

```
//  Serial.println(startReturn);
//  delay(10);
wdt_reset(); // Reset WDT


if (!startReturn) { // If not restarting main loop
  if (currPrime) { // If current primary flagged
    digitalWrite(pPrimeLED, HIGH); // Set debug pin


    if (cam.takePicture()) {
      Serial.println("Image captured on camera");
    } else {
      Serial.println("Image not captured");
    }
    wdt_reset(); // Reset WDT

    // Create unique image file name on SD card
    char filename[13];
    strcpy(filename, "IMG0000.JPG");
    for (int i = 0; i < 1000; i++) {
      filename[4] = '0' + i / 100;
      filename[5] = '0' + (i % 100) / 10;
      filename[6] = '0' + (i % 100) % 10;
      // Create if does not exist, do not open existing,
      // write, sync after write
      if (! SD.exists(filename)) {
        break;
      }
    }
    Serial.println(filename);
    wdt_reset(); // Reset WDT

    // Initialise new image file on SD card
    imgFile = SD.open(filename, FILE_WRITE);

    // Identify size of image to be saved
    uint16_t jpglen = cam.frameLength();
    // While image still has untransfered data
    while (jpglen > 0) {
      // Read 32 bytes at a time;
      uint8_t bytesToRead = min(32, jpglen);
      // Prepare buffer of received data
```

```
        uint8_t *buffer = cam.readPicture(bytesToRead);
        // Write received data from buffer to file on card
        imgFile.write(buffer, bytesToRead);
        // Deduct size of written data from data remainder
        jpglen -= bytesToRead;
        wdt_reset(); // Reset WDT
      }
      // Close image file
      imgFile.close();
      // Notify operator of conclusion
      Serial.println("...Done!");
      // Set camera back into video mode, to clear captured
      // image from camera memory, ready for next image
      cam.resumeVideo();

      // If function concluded later than half the expected
      // time, clear all flags
      // Also include check for return to start flag
      if (elapsedProc > int(.5 * maxProc) && !startReturn){
        digitalWrite(pLockOut, false); // Clear lock flag
        // Clear processing flag
        digitalWrite(pProcOut, false);
        currPrime = false; // Clear current primary flag
      } else if (!startReturn) {
        // Check for return to start flag
        digitalWrite(pLockOut, false); // Clear lock flag
        // Set clearance flag
        digitalWrite(pClearOut, true);
        // Clear clearance flag
        digitalWrite(pClearOut, false);
      }
    } else { // Else, if current primary not flagged
      digitalWrite(pPrimeLED, LOW); // Clear debug pin
      digitalWrite(pLockOut, false); // Clear lock flag
      // Clear processing flag
      digitalWrite(pProcOut, false);
      // If elapsed time is greater than expected
      // Also include check for return to start flag
      if (elapsedProc > maxProc  && !startReturn) {
        // Read processing flag
        processing = digitalRead(pProcIn);
        if (processing && !startReturn) {
          // If processing flag is still set
```

```
          // Also include check for return to start flag
          // Set clearance flag for reassignment of primary
          // priority
          digitalWrite(pClearOut, true);
          // Immediately clear clearance flag
          digitalWrite(pClearOut, false);
        }
      }
    }
  }
}

void watchdogSetup(void) { // Initialise WDT configuration
  cli(); // Disable all interrupts
  wdt_reset(); // Reset the WDT
  WDTCSR |= B00011000; // Enter WDT configuration mode
  // Set WDT settings to activate closing ISR,
  // activate WDT, and set a timeout of 250ms
  WDTCSR = B01001100; // Write configuration to WDT
  sei(); // Enable interrupts
}

ISR(WDT_vect) { // Watchdog timer ISR
  startReturn = true; // Set return to start flag
  wdtOverRide = true; // Set flag to disable main loop
  digitalWrite(pLockOut, false); // Clear lock flag
}
```

Bibliography

[1] M. H. Kim, S. Lee, and K. C. Lee, "Experimental performance evaluation of smoothing predictive redundancy using embedded microcontroller unit," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 3, pp. 784–791, March 2011.

[2] S. Tavallaei, J. Autor, A. Vu, and J. Lacombe, "Computer system comprising a method and apparatus for periodic testing of redundant devices," Nov. 10 1998, uS Patent 5,834,856. [Online]. Available: https://www.google.com/patents/US5834856

[3] C.-S. Yoon and W.-P. Hong, "Design of network-based induction motors fault diagnosis system using redundant dsp microcontroller with integrated can module," *Journal of the Korean Institute of Illuminating and Electrical Installation Engineers*, vol. 19, no. 5, pp. 80–86, July 2005.

[4] T. Kifuku, K. Tsutsumi, and C. Fujimoto, "Electric power steering apparatus," Feb. 3 2004, uS Patent 6,687,590. [Online]. Available: https://www.google.com/patents/US6687590

[5] D. A. Rennels, D. W. Caldwell, R. Hwang, and M. Mesarina, "A fault-tolerant embedded microcontroller testbed," in *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, Dec 1997, pp. 7–14.

[6] K. Dickson and W. Stonehouse, "Method for operating redundant master i/o controllers," Jul. 1 1997, uS Patent 5,644,700. [Online]. Available: https://www.google.com/patents/US5644700

[7] D. A. Rennels and R. Hwang, "Recovery in fault-tolerant distributed microcontrollers," in *2001 International Conference on Dependable Systems and Networks*, July 2001, pp. 475–480.

[8] Atmel, "Atmega328/P 8-bit AVR microcontrollers." Datasheet, Nov. 2016.

[9] SIMCom, "A T Command Set: SIM5320." Datasheet, Feb. 2016.