10-2017

Event and Time-Triggered Control Module Layers for Individual Robot Control Architectures of Unmanned Agricultural Ground Vehicles

Tyler Troyer
University of Nebraska-Lincoln, troyerta@gmail.com

EVENT AND TIME-TRIGGERED CONTROL MODULE LAYERS FOR INDIVIDUAL ROBOT CONTROL ARCHITECTURES OF UNMANNED AGRICULTURAL GROUND VEHICLES

by

Tyler Alan Troyer

A THESIS

Presented to the Faculty of

The Graduate College of the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Agricultural and Biological Systems Engineering

Under the Supervision of Professor Santosh K. Pitla

Lincoln, Nebraska

October, 2017

EVENT AND TIME-TRIGGERED CONTROL MODULE LAYERS FOR INDIVIDUAL ROBOT CONTROL ARCHITECTURES OF UNMANNED AGRICULTURAL GROUND VEHICLES

Tyler Alan Troyer, M.S.

University of Nebraska, 2017

Advisor: Santosh K. Pitla

Automation in the agriculture sector has increased to an extent where the accompanying methods for unmanned field management are becoming more economically viable. This manifests in the industry's recent presentation of conceptual cab-less machines that perform all field operations under the high-level task control of a single remote operator. A dramatic change in the overall workflow for field tasks that historically assumed the presence of a human in the immediate vicinity of the work is predicted. This shift in the entire approach to farm machinery work provides producers increased control and productivity over high-level tasks and less distraction from operating individual machine actuators and implements. The final implication is decreased mechanical complexity of the cab-less field machines from their manned countertypes.

An Unmanned Agricultural Ground Vehicle (UAGV) electric platform received a portable control module layer (CML) which was modular and able to accept higher-level mission commands while returning system states to high-level tasks. The simplicity of this system was shown by its entire implementation running on microcontrollers networked on a Time-Triggered Controller Area Network (TTCAN) bus. A basic form of user input and output was added to the system to demonstrate a simple instance of subsystem integration. In this work, all major levels of design and implementation are examined in detail, revealing the ‘why’ and ‘how’ of each subsystem. System design philosophy is highlighted from the beginning. A state-space feedback steering controller was implemented on the machine utilizing a basic steering model found in literature.

Finally, system performance is evaluated from the perspectives of a number of disciplines including: embedded systems software design, control systems, and robot control architecture. Recommendations for formalized UAGV system modeling, estimation, and control are discussed for the continuation of research in simplified low-cost machines for in-field task automation. Additional recommendations for future time-triggered CML experiments in bus robustness and redundancy are discussed. The work presented is foundational in the shift from event-triggered communications towards time-triggered CMLs for unmanned agricultural machinery and is a front-to-back demonstration of time-triggered design.

Acknowledgments

My advisor originated the contents of this work. He enabled me to explore and wrestle first-hand with the topics herein by his consistent encouragement in things both academic and personal. Without his model leadership talents, my work here would have actually felt something like work – which sounds terrible when compared to all the fun I’ve had. Thank you for everything, Dr. Pitla.

The collective knowledge and wisdom that drips from the minds and hearts of my committee members is a rare combination – one that all men should seek to build up in themselves. I am tremendously privileged to have been continually taught and humbled by the fine corruptors and brainwashers here: Thank you for being my wrestling coaches.

Santosh Pitla, John Boye, Justin Bradley, Yufeng Ge

Speaking of corruption, my lab mates did a fine job of preventing me from working too hard. Thank you. I love you all.

John Evans, Rachel Stevens, Aaron Shearer, Shane Forney,
Jared Werner, Ellen Emanuel, Anna Siebe, James Roeber

A thanks to my family who supported me and have suffered my exploits for no reason more than my whimsical curiosities. While I made this journey longer than I needed to, please know I’ve been having great fun all the while.

Mum & Dad, Joel & Kaylee, Melissa, Grandma & Grandpa

Table of Contents

Figures

Chapter 1 Introduction

Automating agricultural tasks for efficient food, fiber, and fuel production is one of the ways to address the demands of a rapidly increasing world population which is set to reach 9.1 billion people by 2050. Required food production increases could be up to 60 percent of current production rates according to one estimate (Wise, 2013). Most of this increase is expected to occur in densely populated areas, and in still-developing countries of today ("FAO's Director-General on How to Feed the World in 2050," 2009). The Food and Agriculture Organization of the United Nations predicts a 70 percent increase in production demand as newly developed nations gain an increased affordability of meats for regular dietary consumption (U.N. Food and Agriculture Organization, 2009).

Conventional row-crop agriculture has adapted well to its own automation and field management techniques for productivity gains. Commercial equipment manufacturers continue to develop and sell machinery on the basis that the producer's productivity is a function of tractor size, power take off capacity, and auto-steering technologies of the machinery (Klopfenstein, 2016; Zhang and Pierce, 2016). Auto-steer retrofit kits, third-party real-time kinetic global positioning system (RTK-GPS) receivers, and controller area network (CAN) bus monitors enable automation of less-popular machinery products so that many other producers can enjoy similar benefits, but a strong presence of these methods and machines remains to be seen in developing nations.

Moving towards increased production in developing nations means restructuring the crop production automation methods where fields are smaller, hills are steeper, and the fuel is costly (Blackmore et al., 2008; Jensen et al., 2014; Katupitiya et al., 2007). In the push for productivity however, the power output and weight of tractors has grown to where long term effects of soil compaction are negatively affecting yield (Billman et al., 2012; Klopfenstein, 2016). The overall approach to in-field food production may have to move towards a producer’s supervisory role over multiple, smaller machines performing field tasks in a coordinated and cooperative manner (Billman et al., 2012; Zhang et al., 2016).

The cost benefits and productivity increases in using multiple, smaller autonomous machines for field tasks has been identified (Blackmore, Have, et al., 2002; Fountas et al., 2007). The necessary system architectures and requirements have been proposed and presented (Biber et al., 2012; Brooks, 1986; Jensen et al., 2014; Pitla, 2012; V. Silva et al., 2006). These next-generation machines have smaller, lighter frames to reduce soil compaction and treats field tasks as scalable cooperative operations (Blackmore et al., 2004, 2008). This enables the productive execution of any cooperative field-task defined by the number of machines required, and the defined deadlines for task completion.

This evolution of agricultural machinery requires consideration of past agricultural machinery design. Much of the distributed control architecture of modern agricultural machinery was derived from those found in other industries and determining the next generation of agricultural machine design will likely be a matter of adopting methods found in similar places (Zhang et al., 2016). To begin with, the introduction of distributed electronic controllers on tractors rapidly revealed the need to standardize the communication infrastructure. The CAN bus became the communications link layer of choice, and the development of ISO 11783 as an extension to SAE J1939 enabled widespread interconnect usability between differing tractor and implement manufacturers (Zhang et al., 2016, p. 104).

The standardization of on-vehicle communications ushered the beginning of data-driven analysis of agricultural machinery and field tasks. Tractor characterization and testing is performed regularly (Marx et al., 2015; Pitla et al., 2014, 2016). The methods for data collection and interpretation have matured to where inexpensive 3rd party CAN hardware solutions are able to capture on-vehicle datasets (Darr, 2012; Marx et al., 2016).

These distributed-controller architectures are customarily considered in the study of cyber-physical systems – where computer systems are tightly coupled to physical systems, and such control architectures are matured enough to where entire CAN bus networks are easily implemented with inexpensive hardware (Darr et al., 2005; Jadlovska et al., 2016). As both the number of electronic controllers on tractors and system complexity increases along with the necessary added constraints in safety and real-time requirements of autonomous heavy machinery, it becomes more fitting to classify these automated agricultural machines as cyber-physical systems (CPSs) (Herlitzius, 2017; Jacobs et al., 2017; Rad et al., 2015). The disciplines and methods used in the treatment of CPSs therefore, becomes more and more relevant to modern agricultural machinery. Simulation of a CAN-based tractor done by Hofstee and Goense (1999) carries a number of characterization similarities to the CPS architecture developed by Bae et al. (2015) for an aerospace control application. Relatively recent reviews of CPS architecture and applications include considerable mention of applications in agriculture systems (Ahmed et al., 2013; Hu et al., 2012). Center pivot irrigation systems are becoming further instrumented with sensor networks, and modeled as CPSs (Dong et al., 2013; A. Silva and Vuran, 2010). Literature that exposes realistic scenarios where tractors can be characterized as CPSs include steering controllers by Bell et al. and Elkaim et al. (1998, 1997). The second-order system modeling and control by Darr (2004, 2005) shows how small modifications to older, less instrumented machines enables a tight coupling between the physical mechanics of the machine and the computational resources – making a CPS.

The CAN bus is commonly found in general-purpose (non-agriculture related) autonomous robot architectures (V. Silva et al., 2006). This makes modern tractors even stronger candidates for rigorous CPS treatment. Investigations of custom-built autonomous agricultural robots reveals a continued implementation of system backbones as CAN buses as seen in the beans harvester by Saito et al. (2013), and the crop row navigator by Godoy et al. (2010). Both machines are excellent candidates for CPS analysis.

Real-time system safety and reliability of distributed agricultural systems is rarely considered in the context of agricultures' unique operating challenges and conditions - likely since the distributed control methods tend to lag behind other distributed system research fields. However, the methods and considerations for distributed system safety are well published. Fault tolerance was formally outlined by Kopetz (1995) – from an automotive background. Design validation of distributed machines was discussed by Lundin et al. (1996) and a "High Assurance" software engineering workflow for distributed real-time CPSs was presented by Hissam et al. (2015). Finally, an excellent overview a real-time distributed control system is outlined by Thompson et al. (1999), for a large engine. These discussions are not isolated instances of distributed systems design since they each include, or assume the inclusion of a CAN bus, or a CAN bus-like communications medium as the system backbone. These same methods are portable to the increasingly complex autonomous robots being developed in agricultural field robots.

The next step in autonomous agricultural machinery is the establishment of a time-triggered communications medium to increase bus efficiency and determinism. Further, as an effort to reduce message latency and bus design constraints. Likely link-layer protocol candidates for this design include time-triggered CAN (TTCAN), time-triggered flexible date rate CAN (FD-TTCAN), and Flexray (Zhang et al., 2016, pp. 104–105).

Benefits of time-triggered communication architectures in machinery control systems becomes evident after a short overview analyzing event-triggered CAN bus shortcomings and the associated system design problems that result. These are exceptionally outlined by Al Saadi (2013) and Juanole, et al. (2005), and direct comparisons of event-triggered to time-triggered systems are both in-depth and ubiquitous (Albert, 2004; Amir and Pont, 2013; Ataide et al., 2006; Cena et al., 2005; Leen and Heffernan, 2002). The case for message scheduling (which is the basis of time-triggered communications) reveals a great deal of literature that explores the methods which provide the system designer the communications assurances desired in complex systems.

Scheduling analysis in real-time system design carries over from task scheduling in computer systems, to distributed communications systems as the communications bus is treated analogous to a central-processing unit (CPU) - a shared resource that only one system component has access to at any given point in time. Rate-monotonic schedulability analysis was demonstrated by Liu and Layland (1973), and Y.T. Leung and Whitehead (1982), where each task (message) occurrence frequency was treated as an unchanging system requirement. Earliest-deadline-first methods described by Meschi (1996) were applied to CAN bus communications by several outstanding papers that prove the schedulability of a system with defined message deadlines (Fuster et al., 2005; Pedreiras and Almeida, 2002; Shoukry et al., 2011). The violation of priority-based ID-field message arbitration to the CAN bus by some CAN controller hardware implementations was demonstrated by Davis et al. (2011), and discussed in texts by Lawrenze (1997), and Di Natale et al. (2012). Statistical and stochastic CAN message latency measurement methods have also been presented (Di Natale et al., 2012; Nolte et al., 2002; Rodríguez-Navas et al., 2003). Effects of component failure and the “babbling idiot” problem are outlined by Lari et al. (2007), and Kopetz (1995).

A variety of time-triggered message scheduling techniques are described and demonstrated by a set of particularly helpful papers: Schmidt and Schmidt (2007) described the design of message scheduling matrices and their variants. Short et al. (2016) presented a time-triggered messaging technique for enhanced fault tolerance under statistical analysis. Tenruh (2011) presented an optimal scheduling technique that minimizes message latency for both time-triggered and event-triggered messages. Weidong et al. (2006) helpfully demonstrates TTCAN message scheduling on an underwater vehicle in a very practical design example. Analogous to the worst-case execution time of a computational task, Xia et al. (2013) showed the worst-case transmission delay of an event-triggered message that participates in a time-triggered messaging scheme. For very complex systems, Almeida and Fonseca (2000) presented a dynamic scheduling method that treats a large number of system requirements as schedule parameters to produce a dynamic run-time scheduling algorithm that handles both scheduled and sporadic messages.

Comparing these in-depth CAN bus communication considerations with those found in the agricultural equipment sector is difficult. J1939 and ISO 11783 high-level protocols were developed under the assumption that the CAN bus is just another “link-layer” to the OSI communication model. This is an unfortunate presumption, one which is evident by the missing references to any of the message reliability and determinism concerns in common outlines and implementations of these standards. It is apparent that modern tractors are made with event-triggered messaging techniques only, with the only commonly-cited performance metric as bandwidth percentage – or “bus load” (Zhang et al., 2016, p. 104).

Derivations, descriptions, and implementations of ISO 11783 for instance, entirely omit any mention of communication deadlines, real-time consideration, message latency, fault handling, or safety assurances (McKee et al., 1999; Oksanen et al., 2005; Speckmann and Jahns, 1999). This leaves system design engineers in the agricultural machinery sector with only an ‘intuition’ method for assigning message priorities and payloads in these event-triggered standards. This leaves systems engineers little or no assurances or proofs about the schedulability and worst-case message latencies for systems that can contain hundreds of unique message identifiers and consequent bus priorities.

Simulations of a CAN-based tractor system operating on ISO 11783 done by Hofstee and Goense (1999) reveal message “transfer times” between 6 mS and 70 mS as a function of bus load and CAN bus component arrangement. When system requirements such as worst-case message latency are not defined for the system messages, these results may not appear to be concerning. Though, as hard real-time requirements of tractors manifest as the machines become fully autonomous, these results manifest as high levels of jitter - insufficient for systems safety assurance and the required variability of the tasks and implements for the large variety of autonomous tractor uses. A formalized consideration of system safety is required for autonomous robots and general control systems (Blackmore, Have, et al., 2002; Juanole et al., 2005).

There are a number of robot control architectures described for a subset of a tractor-derived agricultural operations. For instance, some architectures focus on robot navigation, with a subset of them considering the needs of autonomous agricultural steering (Bakker et al., 2010; Blackmore, Have, et al., 2002; Brooks, 1986; Will et al., 1998). Control architectures described by Eaton et al. (2005) and Katupitiya et al. (2007) focus rather on implement and manipulator control, allowing for the inclusion of a wide number of agricultural tasks to have reliable control loops on the CAN bus.

Other control architectures consider broader uses for the general-purpose autonomous agricultural robot. Chan et al. (2014) specified a wide area network communications infrastructure and Blackmore and Fountas et al. (2002) focused on a software-based object-oriented approach to central system architecture. Pitla (2012) developed a scalable and hierarchal architecture, distinguishing hardware tasks, software tasks, robot tasks, and robot group tasks.

When it comes to real-time considerations, many agricultural robot architecture developments do not include the important, yet hidden details. Some architectures make no claims of real-time considerations at all (Biber et al., 2012; Brooks, 1986; Godoy et al., 2010; Jensen et al., 2014; Saito et al., 2013). Other system descriptions include mentions of real-time communication mediums but do not specify messages, schedules, or system deadlines (Chan et al., 2014; V. Silva et al., 2006). There are a few agricultural system descriptions that include the system messages, but use an event-triggered communications scheme – which removes possibility for the inclusion of a messaging schedule (Pitla, 2012; Wei et al., 1998).

Fortunately, within a majority of these architectures and robot descriptions, a common low-level control layer is consistently found. This system abstraction layer assumes closed-loop management of machine actuators and the capacity for high-speed sensor interfacing. This is typically described as a subset of the CAN bus network components that transmit sensor data to the CAN bus, and receive control set-point commands from other components on or off the bus. Beyond this level of description, this layer of system design (perhaps the most important to get 'right') is not commonly described and explained in detail – especially in the descriptions of agricultural robots. A very useful exception to these missing system details is found in the autonomous underwater vehicle by (Weidong et al., 2006) - which, unfortunately, is not a system typically used in the realm of agriculture.

Time-triggered communications in the lowest layer of proposed autonomous agricultural robot architectures have not been detailed and presented to the knowledge of the author. This component will be vital for a safe, and modular set of agricultural automation machines that operate as cyber-physical systems for the efficient and cost-effective increase of global agricultural production.

This thesis is an exploration into the practical implementation of a time-triggered communications networked control system on an unmanned agricultural ground vehicle (UAGV). A number of the chapters are high-level overviews of topics for the agricultural mobile robotics researcher to become familiar with. This is an effort to accelerate the researcher's perspective of a research field which tends to be two or three steps ahead of the current Agricultural and Biological Systems Engineering curriculum.

1.1 Thesis Objectives

1) Develop and demonstrate a control module layer on an inter-row agricultural robot (Chapter 2)
2) Explain problems addressed by real-time system design (Chapter 3)
3) Provide a background of CAN bus system design (Chapter 4)
4) Introduce the UAGV and describe the development of its components (Chapter 5)
5) Develop and demonstrate the hardware and software of the TTCAN communications protocol (Chapter 6)
6) Design and demonstrate a basic state space control system on the UAGV with TTCAN Control Module Layer (Chapter 7)

Chapter 2 Event - Triggered Control Module Layer for Autonomous Inter-row Navigation during GPS Outages

2.1 Introduction and Objectives

Variable-rate precision agriculture techniques can increase the efficiency of material application on crops. Nitrogen, water, and herbicide applications use spatially sampled field data from remote sensing imagers on satellites and unmanned aerial vehicles (Sowers et al., 1994). The limitations of aerial imagery have been identified and remains as a bottle neck to the spatial resolution of field data. Typical pixel resolutions from image-based remote sensors are insufficient for plant-by-plant analysis (Moran et al., 1997). Chemical content index derivation (NDVI, red-edge, etc.) relies on the absence of cloud cover, and a sufficiently developed crop canopy to be useful. Finally, soil-centered indices are difficult to obtain through late-stage crop canopies, requiring instrumentation below the canopy or in direct contact with sub-canopy soil.

These imagery limitations reveal a level of crop sampling that never reaches consideration of the individual plant or of high resolution soil data during late stages of crop development. Consider the issue of plant spacing variation as an example. Corn yield models such as the one developed by Martin et al (2012) utilizes plant-by-plant spacing as input. Plant spacing is important in this model, as experiments done from 1966 to present show that high variation in plant positioning negatively correlates to the yield of the crop and positively correlates to yield variation of the crop’s individual plants (Erbach et al., 1972; Krall et al., 1977; Lauer and Rankin, 2004; Mead, 1966; Soman et al., 1987; Thompson et al., 1999). These experiments used data collected by hand-made measurements, which limited the number of samples taken. The development of the model by Martin et al. (2012) came from data collected from bicycle-mounted optical sensors which were pushed by a researcher through crop rows.

As discussed in Chapter 1, agricultural data collection systems are strong candidates for machine automation. Larger fields could be sampled entirely and more often if the relevant instrumentation is mounted to an autonomous robot. One such instrument for precision spacing measurement developed by Shi et al. (2013) is a strong candidate for automation. In device experiments, only two experiment trials were used for instrument validation on a single 10 meter long corn row of 50 plants. A much stronger case for experiment validation is made by considering the additional number of corn plants that could be sampled by a ground robot that can drive past every plant in the field – The resulting statistical analysis would benefit from the much larger count of plant spacing samples.

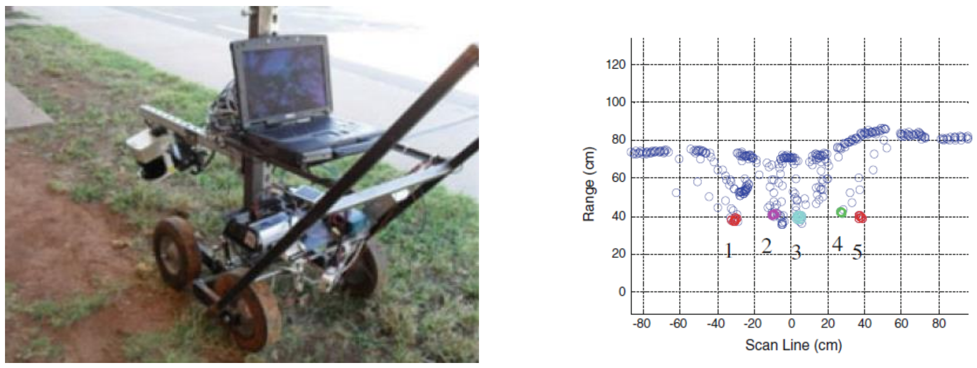


Figure 2.1 – Data collection cart and plant spacing data (Shi et al., 2013)

An autonomous machine capable of carrying sub-canopy crop sensors and instruments should be able to navigate crop rows in the presence of unreliable global positioning system reception. Overhang of the crop canopy in particular can easily be enough to interfere with field navigation instrumentation - preventing GPS-based navigation and the dependent steering algorithms from functioning properly. An additional requirement for such a machine would be the ability of the system to correct itself after unforeseen disturbances from the environment. If the vehicle is relatively lightweight, driving over ordinary dirt clods in its path is enough to suddenly change the robot's pose, which can frustrate a state estimator and controller. This makes modeling such a system very difficult. The physical coupling between the machine and the uneven terrain includes very strong system disturbances that make it difficult to account for them. A row-navigating robot can minimize these effects by using frame stabilization mechanisms such as a suspension system – either passive or active.

The challenging environment of the single crop row represents the second great challenge to reliable robot operation in the field when coupled with considerations for reliability of GPS signaling during sub-canopy navigation. Ultimately, urban, indoor, and agricultural environments present navigation limitations of localization sensors that rely upon GPS satellites or the presence of steady lighting conditions. The addition of supplementary navigation mechanisms for improved navigation reliability of agricultural robots is recommended, and such implementations should be attempted while remaining sensitive to system costs (Tillett et al., 1998).

Objectives:

Develop and demonstrate a control module layer on an inter-row agricultural robot (from Section 1.1).

1) Explore the feasibility of short-term crop row navigation without available GPS input
2) Develop a low-cost supplementary robot steering method for autonomous crop row navigation during GPS outages
3) Present simplified control plant model for navigation control
4) Demonstrate controllability of the controller model in a crop row

2.2 Materials and Methods

2.2.1 Drive Chassis

The drive platform (Figure 2.2) of the robot was a differentially driven set of 6 wheels with an aluminum frame and DC motors for each wheel. The platform dimensions were 12 x 18 inches. Each motor-wheel assembly had a clamp spring for added suspension. The intent was to choose a platform that keeps the on-board crop-row sensing instrumentation stable during row navigation. Keeping the instrumentation steady was found to be important for accurate distance measurements of the row. Dirt clods, puddles, and pivot tracks are among the anticipated obstacles and disturbances to the controller. The platform was purchased from an internet-based vendor and assembled with additional hardware for mounting sensors and controllers.

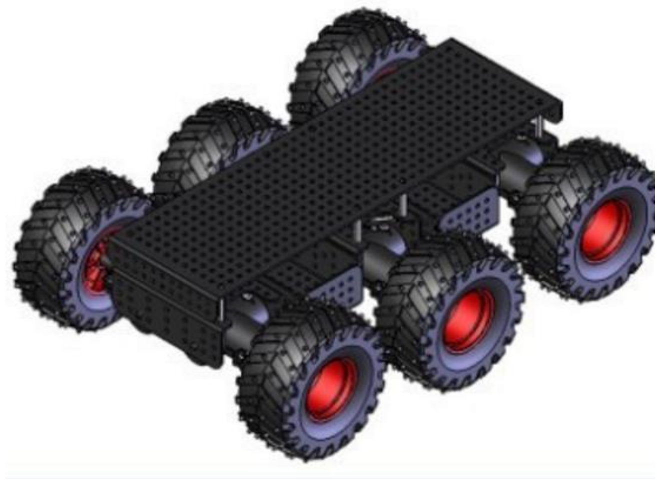


Figure 2.2 - Dagu Wild Thumper Chassis (DAGU Robotics, Zhongshan City, China)

Additional hardware installed on the chassis included a front bumper, a leaf guard ring, and mounting bracket for aiming the distance sensors. These parts were constructed from aluminum stock. The leaf guard (Figure 2.3) was particularly necessary to enforce a lower limit to the distances measured by the sensors, which would otherwise be exceeded if leaves were allowed to hang immediately in front of the sensor apertures. The sensors were pointed 45 degrees forward, across the chassis center such that their respective lines of sight intersected 7 inches in front of the chassis. This alignment was achieved via the visible laser dot generated by the sensors.

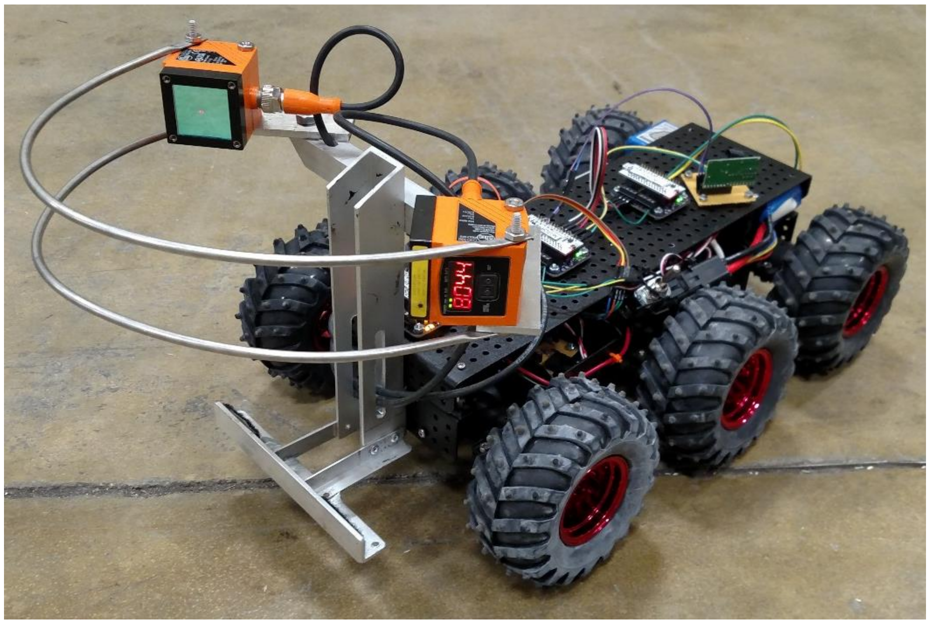


Figure 2.3 - Positioning and aiming of distance sensors behind leaf guard

The top surface (Figure 2.4) of the chassis held mounts for the CAN controller circuit boards and a 433 MHz wireless general purpose input/output (GPIO) receiver (Adafruit Industries, NYC, NY). Compartments below the top surface housed a 13V 3-cell lithium battery (5000mAh 3S, Turnigy, Hong Kong, CN), a power unit, two DC motor drivers (Victor SP, Vex Robotics, Greenville, TX), and a voltage divider to reduce the sensor output voltages to safe levels for the CAN node analog input pins. The power unit supplied a regulated 24VDC to the distance sensors and 5V to the CAN nodes. Figure 2.4 also depicts a CAN bus / power pigtail at the rear of the chassis for connecting a CAN message logging device (Memorator, Kvaser Inc., Mission Viejo, California).

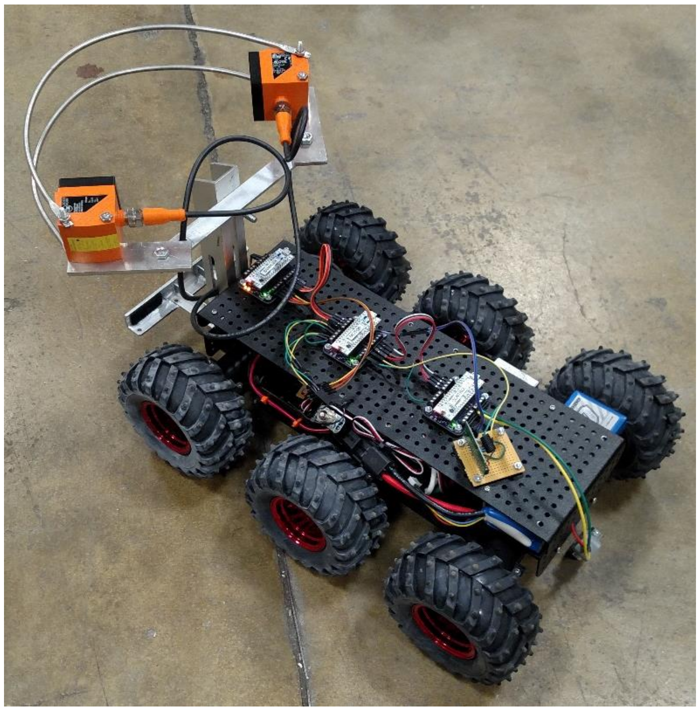


Figure 2.4 – Completed CAN bus system mounted on chassis

2.2.2 Control Model

Differential steering is the heading adjustment method of a vehicle by commanding a difference in wheel rotation rates between the left and right side wheels. Small mobile robots benefit from such simplified steering mechanisms that reduce the moving parts count, which in turn reduces the system complexity and cost. By simply commanding a speed difference, the robot can be steered left or right. This design benefit comes at the cost of complex system dynamics for vehicles with four or more wheels.

Kinematic models assume a constant center of rotation to the chassis, which in reality is highly variable depending on a number of system states that change quickly, including the coefficient of friction between each tire and the ground, the incline angle of the chassis, and the looseness of the soil under the machine. The center of rotation is further complicated by the design of the chassis wheels (Figure 2.2), where the middle axle of the machine is mounted lower – further away from the bottom of the chassis. This is an effort to guarantee that the middle axle is always in contact with the ground. This also causes only one of the two remaining axles (front or rear) to be able to strongly contact the ground at any time. This means the chassis behaves more similarly to two separate 4-wheel differential drive systems (one towards the front of the chassis, the other towards the rear), where the shift between one set of 4 wheels to the other set can occur both frequently and randomly – depending on the distribution of weight on the robot and the robot's angle of contact with the terrain.

Dynamic models tend to rely on a constant center of mass of the vehicle, which is likely to change as the instrumentation payload of the robot changes. This would require a change to the system model and controller for every change of instrumentation on board, yielding a rather tedious workflow for the end operator of the robot. A more universal robot model could benefit the design of the steering controller and the end user.

The model developed for inter-row navigation (Figure 2.5) is a kinematic model that assumes an operating environment of a crop stalk matured to at least 6 inches of stalk height - which is a superset of the use cases for a machine that initially operates underneath a crop canopy. Crop rows with short plants can benefit a robot with reliable GPS signal reception. The model treats the crop row as a pair of solid barriers to navigate the robot between – using two measured distances on either side of, and in front of the drive direction. The measured distances are 45 degrees forward from the perpendicular distance of the robot to the nearest plant in the row. This is a 1:1 coupling of row heading state information and center-positioning state information. This dual-state coupling is possible because of the feedback direction required for each of the two states' contribution to the error signal where:

$$error = Dist_{Right} - Dist_{Left} \tag{2.1}$$

This effect is demonstrated with the step responses depicted in Figure 2.6, showing how an error signal derived from only heading error is of the same sign as the error signal resulting from only center positioning error. The error signal is then a superposition of the heading error and centering error. This is in effect, a system reduction from a two-input, single output system to a single-input, single-output system. This simple single-input, single-output model was paired with a proportional controller, and proportional-integral controller to demonstrate controllability of the robot in the presence of large measurement disturbances.

The inevitability of measurement noise due to inter-plant gaps and overhanging leaves was modeled as a strong disturbance with upper and lower bounds to its effect. That is, each measured distance presented to the feedback control algorithm was constrained between some minimum and maximum bounds. The "true" state is the sensor measurement of the barrier distance - when it is accurately aimed at the plant stalk. This instance is shown as L_{est} in Figure 2.5. A disturbance to this measurement is shown as R_{est}, where the sensor misreads the stalk location by reading overhanging leaves and objects beyond the barrier (crop row).

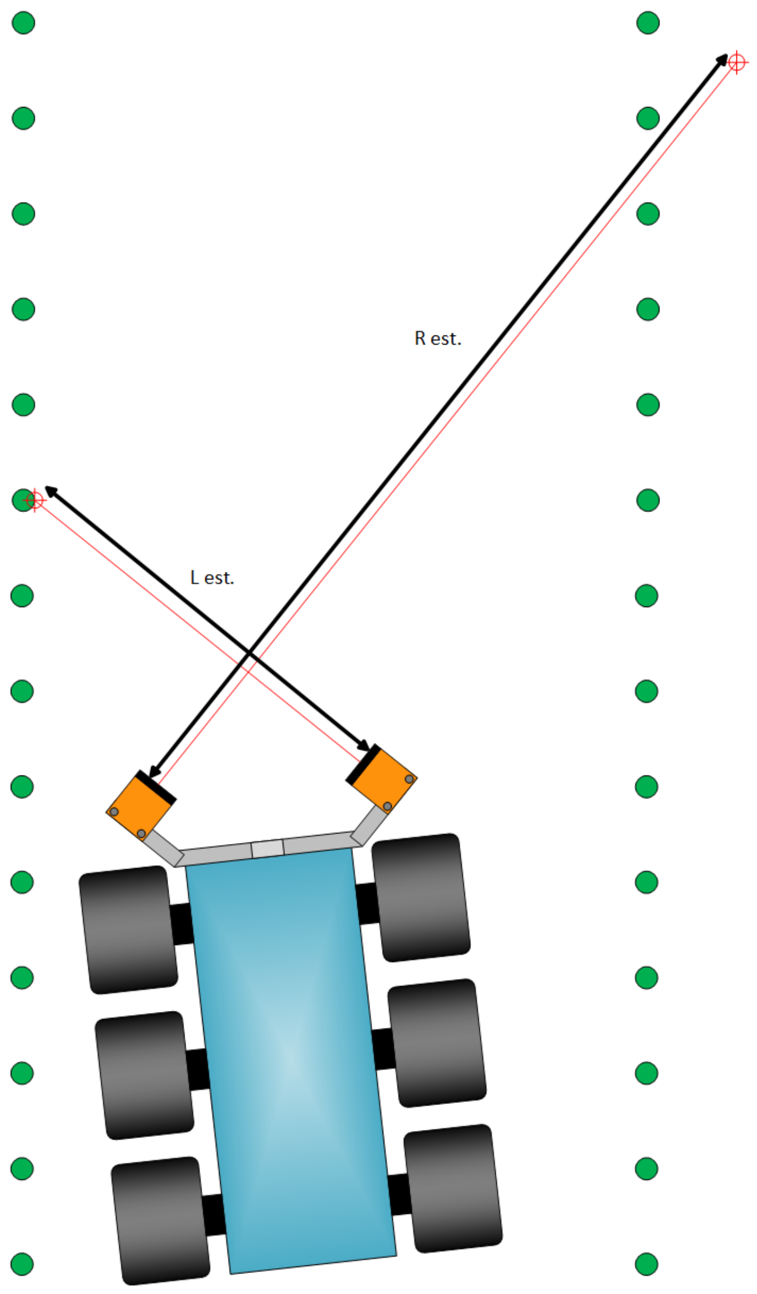


Figure 2.5 - Control Model including noisy crop row distance inputs

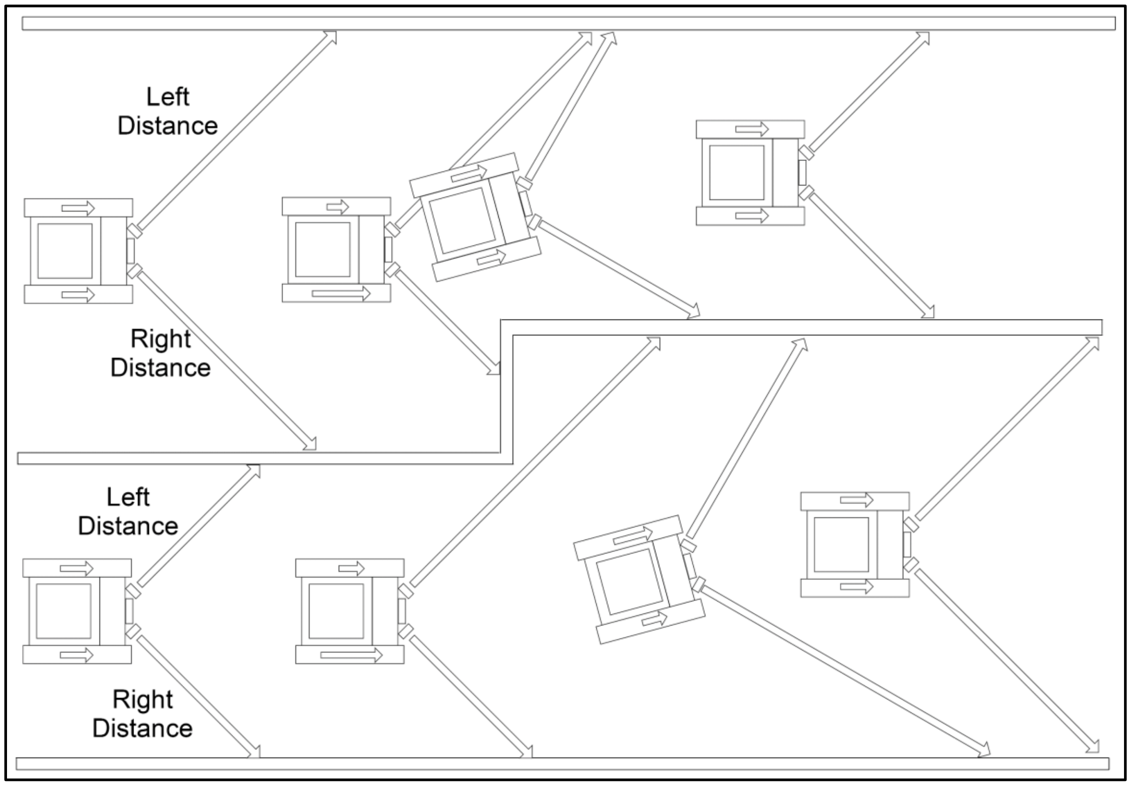


Figure 2.6 - Centering and heading error contributions to error signal generation

2.2.3 System Architecture

A distributed system architecture was selected for the scalability it lends to system modification and maintainability, and for the simplification of the individual firmware tasks across the system. Figure 2.7 depicts the system's organization across a CAN bus. The component interfaces are also shown. Interfacing all system components to the CAN bus also simplified access to central system variables being transmitted periodically to the bus, for both debugging and logging purposes.

Left and right-side motor drivers were individually controlled by pulse-width modulation (PWM) on one CAN node. Distance sensors were interfaced to another CAN node with an analog signal proportional to the measured distance. A CAN message logger device was connected to the bus for system analysis. Wireless control of the system was achieved with a pin toggle switch receiver (Adafruit Industries, NYC, NY) connected to the feedback node, which performed the feedback control calculations. The entire system performed control cycles at 200 Hz, sampling the distance sensors, generating a control error signal, calculating controller commands - actuating the motors in each cycle.

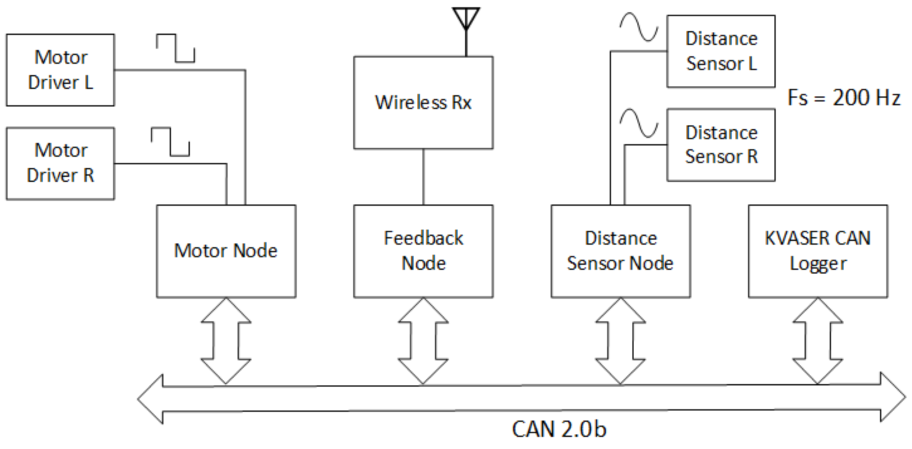


Figure 2.7 - Row follower system architecture

2.2.4 System Components

An overview of the system components is presented in an effort to describe implementation details that often get overlooked in the literature.

2.2.4.1 Microcontroller-based Controller Area Network (CAN) nodes

To interface the system sensors and actuators to the centralized system, a microcontroller development board coupled to a CAN bus interfacing board was used (Figure 2.8). The development board is a low-cost microcontroller coupled with a target debugger device that works with computer-based software development environments to download and debug target firmware. This ST-Link debugger (ST Micro, Geneva, Switzerland) enabled a USB interface for each CAN node, and simplified firmware development for each controller. The development boards were installed on a customized CAN interface board (CAN “sled”). The sled included screw terminals that connected microcontroller GPIO pins to external wiring for power, analog input, PWM output, and differential CAN bus signaling.

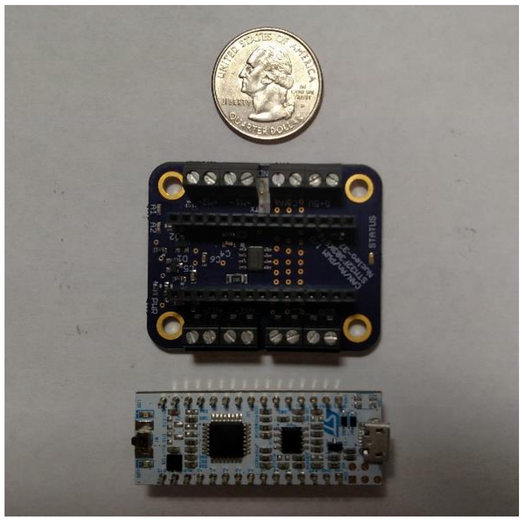
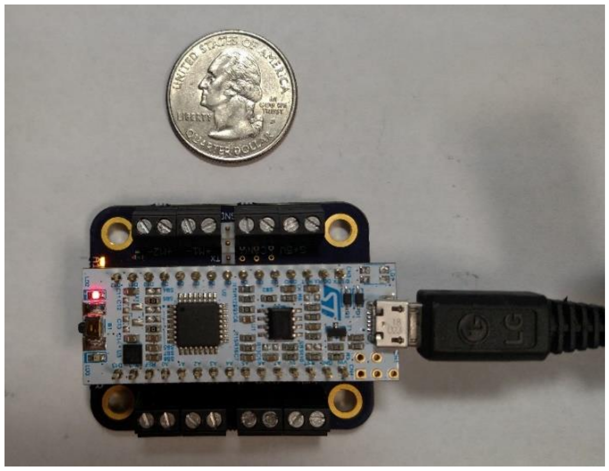



Figure 2.8 – Board for interfacing to 1 x CAN, 2 x PWM outputs, and 2 x analog inputs

Circuitry on the sleds included a CAN bus transceiver (Texas Instruments, Dallas, TX, USA) to translate microcontroller pin logic voltage levels (0V / 3.3V) to the differential signaling of the CAN bus (0V / 1.65V / 3.3V), two LED indicators for visual system feedback, a serial USART connector for terminal messaging to a computer, and input pin protection circuitry for two analog input pins on the microcontroller (Figure 2.9). The protection circuitry limited the voltages presented to the microcontroller pin to prevent damage. The circuit included a low pass filter, a current limiting resistor, and a Schottky diode pair for re-routing negative voltages and voltages above 3.3V.

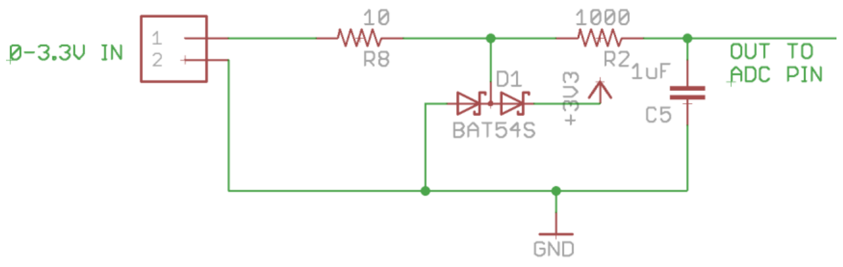


Figure 2.9 - The clamping filter for protection of the microcontroller ADC pins.

Wiring for power and CAN bus connectivity on the robot chassis was simplified by routing the traces straight across the board, enabling a daisy-chain connection for each node. Since CAN bus signaling is normally a twisted-wire pair to reduce the effect of additive signaling noise, the CAN traces through the board were bordered with vias to reduce the traces' susceptibility to noise generation and collection. Figure 2.10 highlights the routing paths through the CAN sled.

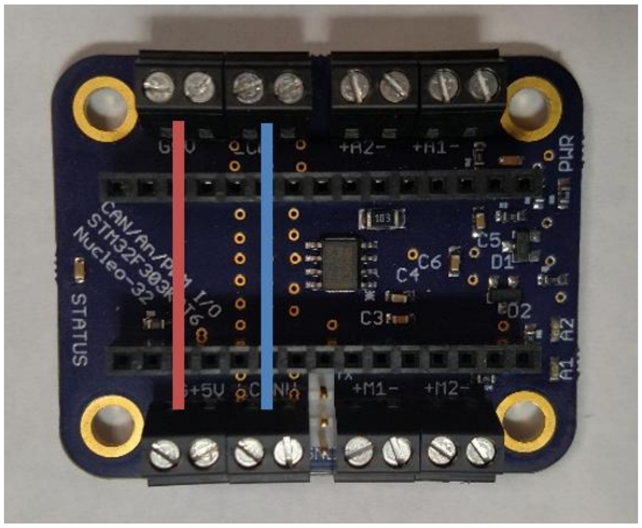


Figure 2.10 - Pass-through buses for power and CAN bus "daisy-chaining"

2.2.4.2 Sensors

The distance sensors selected (Figure 2.11) operated on 24VDC, and provided an analog voltage output proportional to the distance to nearest object in the sensor's narrow line of sight. The output voltage ranged from 0-10VDC, but was divided down to 0-3.3V to maximize the usable range of the microcontroller's analog-to-digital converter. The 2:1 ratio of the resistor values made the division of 3 simple to implement with 3 resistors of equal value. These sensors operated on optical wavelengths appropriate for outdoor use as demonstrated by Pitla et al (2008). The sensor CAN node sampled both analog signals of the sensors autonomously with a 12-bit analog-to-digital converter operating in line with a direct memory access controller. The sampling process and the sample transfer process were both in continuous operation at 250 Hz, triggered by a hardware timer. After the transfer of both analog samples was completed, an interrupt handler converted the samples to voltage units, and then to centimeter distance units with a calibration equation before transmitting the distances to the CAN bus in a single, two-byte message. The distance values transmitted to the CAN bus were saturated to 150 cm to simplify signal processing down the signal chain.

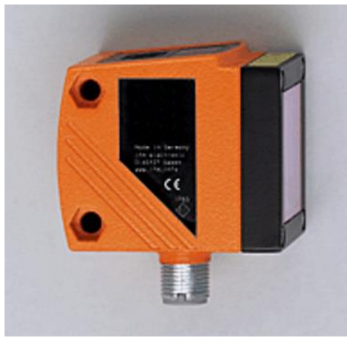


Figure 2.11 - O1D100 by Ifm-efector, Inc. (Exton, PA) 20 to 1000 cm range

2.2.4.3 Actuators

The DC motor drivers were selected for their flexible compatibility with a large range of DC motors, as well as for their compact size. The DC motors included with the chassis kit were characterized by their 34:1 gearboxes and the 12VDC operation by the manufacturer, but the remaining motor characteristics were not provided. It was also found that the set of motors included with the chassis were highly variable in their behavior steady-state response to spin commands - responding to identical 12V PWM inputs with high variability of resulting rotation speeds under both minimum and maximum load. The three motors of each left and right side of the chassis were connected to a motor driver in parallel in an attempt to minimize the deviance effects of any single motor to the sum of the robot’s resulting left and right side speeds.

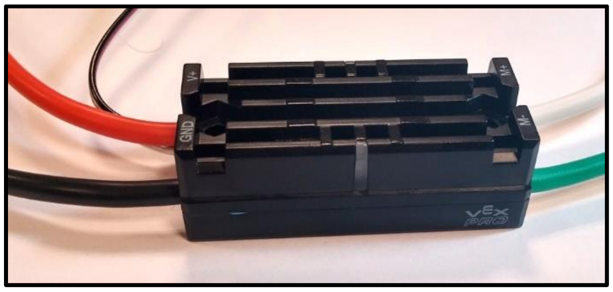


Figure 2.12 - Victor SP, Vex Robotics, Greenville, TX, 60A Continuous, PWM input

2.2.5 Communications

2.2.5.1 Protocol

An event-driven message passing protocol was selected for data transfer over the CAN bus. The protocol was highly procedural, where each of the steps of system operation were dependent on the step previous to it. Beginning at the distance sensor CAN node, an internal microcontroller timer triggered an analog-to-digital (ADC) converter peripheral at a rate of 200 Hz. At each trigger, the ADC began a fast sampling process on the sensor output voltage which repeated 50 times. After each sample was collected, an internal direct memory access peripheral on the microcontroller transferred each ADC conversion to a buffer. After all 50 samples were transferred, an interrupt generated for the processor calculated the mean of the contents of the buffer. After the mean result was converted to distance units of centimeters, the interrupt concluded by packing the results to a CAN message and then finally sending the message to the CAN peripheral for transmission on the bus. This was done for each sensor channel, resulting in two CAN messages being transmitted from the sensor node. This conversion and transfer of data was selected to simplify the microcontroller’s software, which only configured the internal peripherals and then handled a single interrupt. This kept the microcontroller very responsive and deterministic in execution of its tasks.

The feedback CAN node received the distance measurements from the CAN bus and began a filtering process to determine the system state and to calculate the corresponding control signal. This process happens in a low-latency interrupt that responds to the arrival of the distance CAN message itself. The results of the controller computation in turn immediately get transmitted to the CAN bus.

The final step of the protocol is the reception of the control message by the motor CAN node, and the resulting actuation of the motors via modification of PWM pin duty cycles.

This event-triggered mechanism allowed for low-latency, high frequency system operation. This topology resulted in a small amount of message jitter visible on the bus via oscilloscope. The transmission of the control message to the motor node was dependent on an internal computation of state performed by the feedback node. The response of the feedback node was then directly proportional to the time it took to compute the results. The system performance in this mode was sufficient given the low number of system messages and simplicity of the controller firmware. A system cycle is shown in Figure 2.13, depicting the two distance sensor CAN messages followed by the feedback node's control message. The repetition of this cycle is shown in Figure 2.14 at 100 Hz, though 200 Hz was used for the final tests.

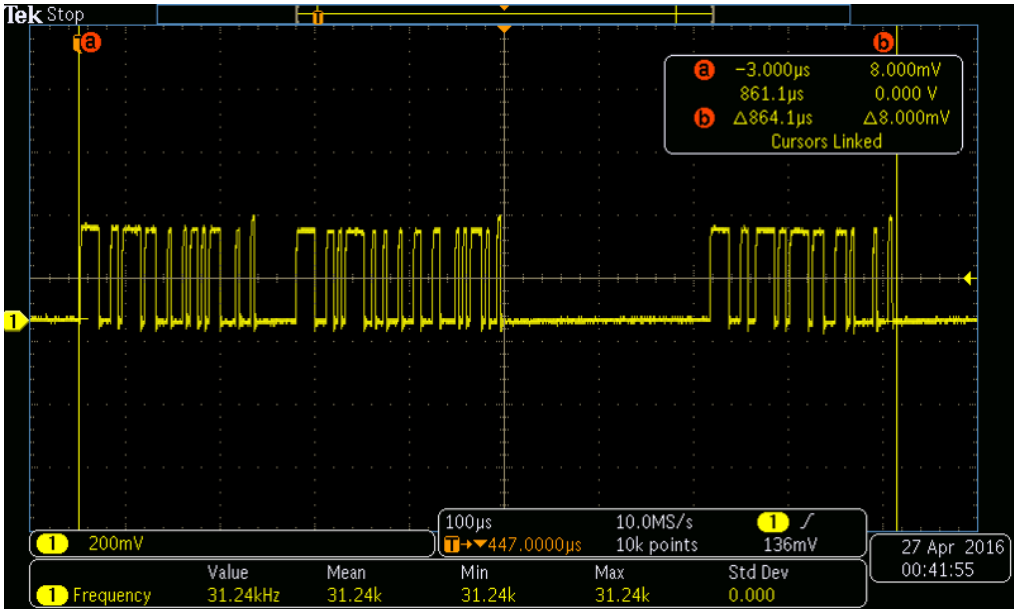


Figure 2.13 - View of CAN bus communications protocol - microscale

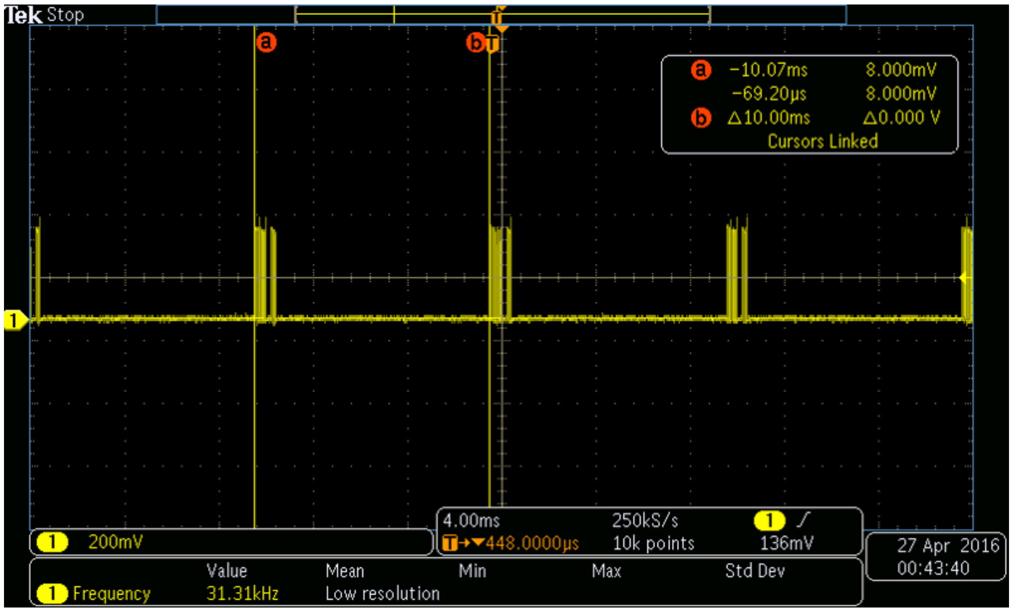


Figure 2.14 - View of CAN bus communications protocol - macroscale

2.2.5.2 System messages

The defined system messages carried sensor and control data between system controllers, and also provided access to central system variables useful for characterizing system behavior and performance measures. Only two messages between the three CAN controllers were required for corn row navigation (Figure 2.15, Figure 2.16). Since the three CAN nodes represent only a subsystem to a larger robot, minimal use of the CAN bus was sought to demonstrate the lightweight requirements of the system. A third message was defined as a re-transmitted combination of the first two in order to simplify the use of the CAN message logging device. The control model and therefore the controller CAN node required only two system variables for state estimation, the left and right distances measured by the sensor CAN node.

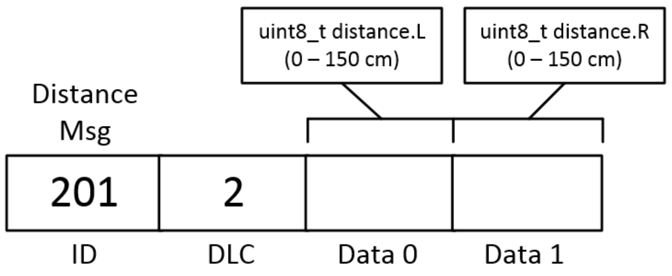


Figure 2.15 - Unsigned distance samples were transmitted in centimeters.

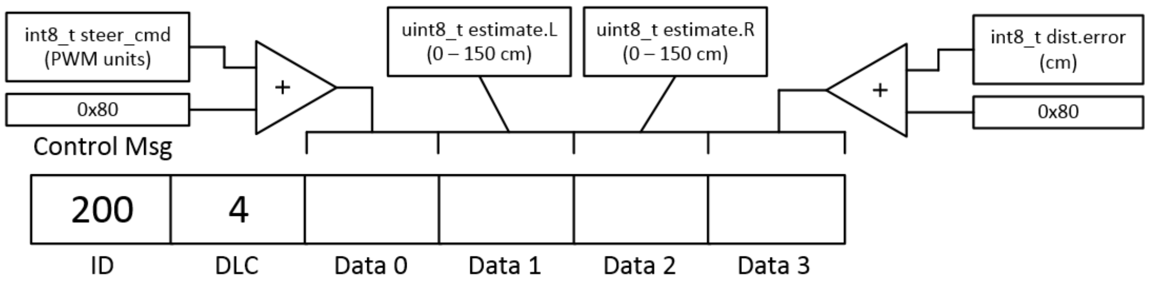


Figure 2.16 - Estimator results and feedback commands control message.

2.2.5.3 Signal Processing Chain

Distance estimation occurred for both distance samples to enable possible differences and biases between the two sensor responses. After the initial reception of the distance samples, they were processed through a conditional zero-order hold filter that passed the previous valid sample to its output, if the current sample was not deemed valid, by exceeding a maximum threshold, or by falling below a minimum threshold (Figure 2.17, Figure 2.18). This filtering technique relied on the assumption that the robot's position could not deviate very far from the center of the crop row between 250 Hz distance samples. This allowed the controller to ignore abrupt changes to the distance signals caused by overhanging leaves and air gaps between plants. After providing samples that fell into an acceptable range of values, the samples were filtered with a singledimensional Kalman exponential filter to smooth the distance samples. The filter was chosen for its computational simplicity and tuning methods, as well as the signal's lack of identifiable frequency ranges to block or pass that conventional digital filters are designed around.

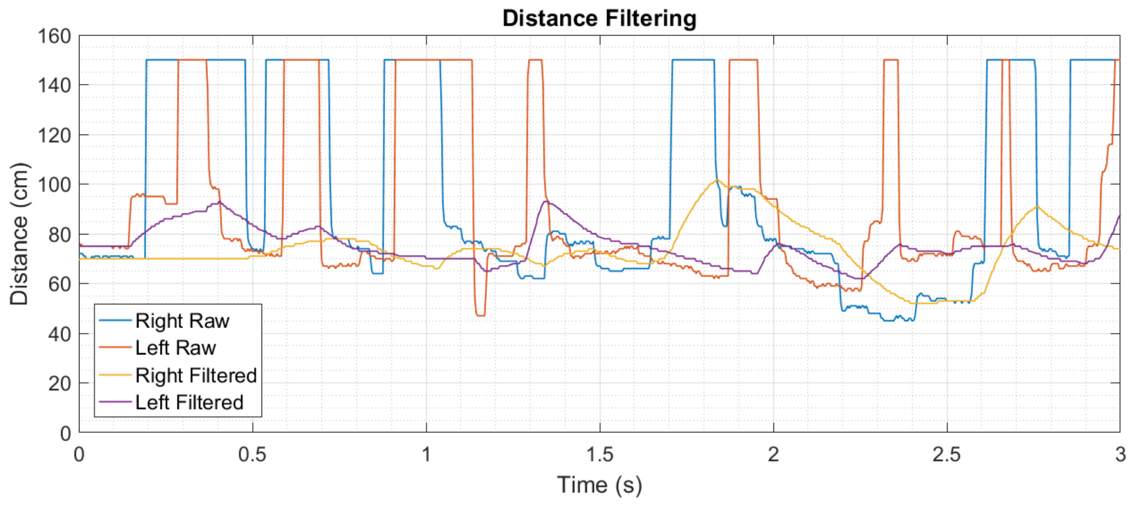


Figure 2.17 - Filtered signals better represent left and right side distances to the crop

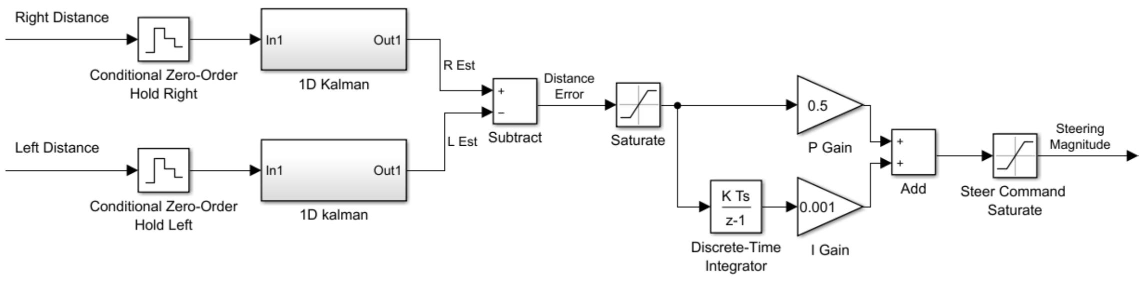


Figure 2.18 – Signal flow for distance estimator and PI controller

2.2.6 Feedback control system

2.2.6.1 Differences between controllers

The filtering mechanism provided a "true" estimate of the crop stalk distances, which enabled the easy inclusion of a proportional and proportional-integral feedback controller for steering down a crop row. Both the input and output side of the controller was saturated to install a hard limit to the still-present effects of error spikes from gaps in the crop row. The gains of the controller were gaining empirically, as an accurate plant dynamic model was unavailable in mathematic terms of the distances as measured. Tuning the controller was performed assuming perfectly smooth rows to isolate effects of the filter from the performance of the controller.

Smooth parallel barriers were used to guide the robot toward a step input for the controller to respond to, as shown in Figure 2.6. The row width started as 100 cm and stepped up to 130 cm. A proportional gain was derived from halving the gain from instances of controller oscillation. The proportional-only response is shown in Figure 2.19, and the final proportional-integral controller with a reduced steady state input shown in Figure 2.20.

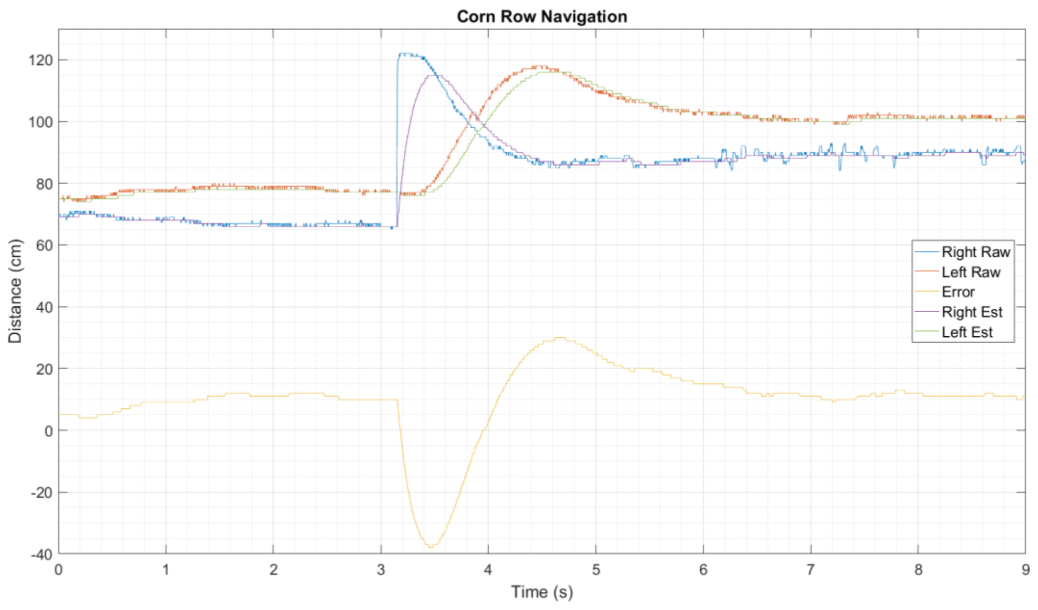


Figure 2.19 - Proportional controller response to step input

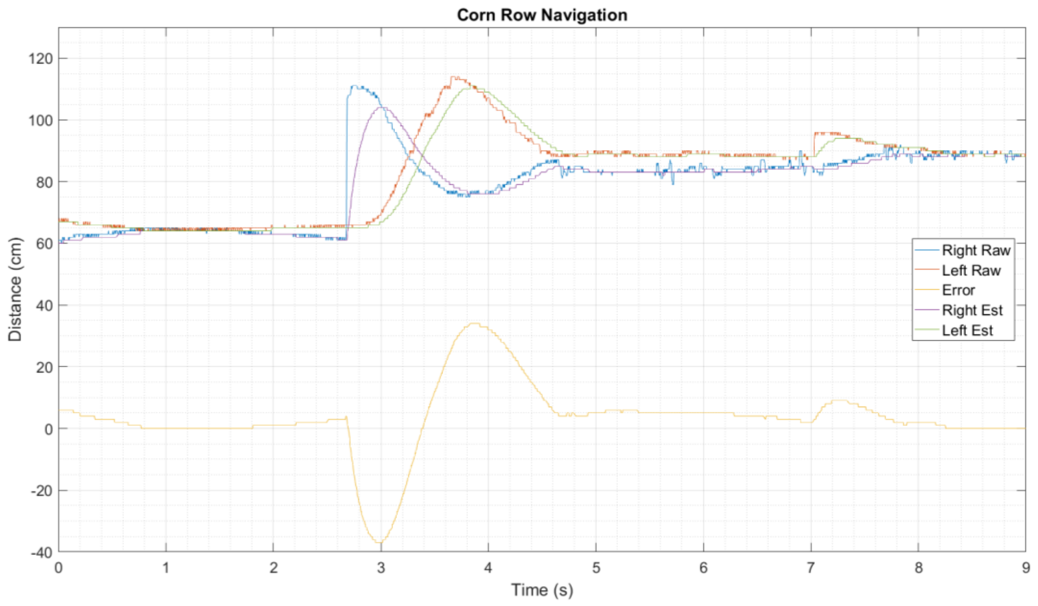


Figure 2.20 - PI response to step input, showing decreased steady-state error

2.2.7 Experiment Design

The conclusion of controller tuning led to the formal observation of navigation performance in both simulated and actual rows of corn stalks. Navigation runs on simulated corn rows occurred indoors and tested the distance filtering techniques on level terrain, and the outdoor tests characterized controller error under maximum environmental disturbances including plant gaps, overhanging leaves, and uneven terrain. Simulated corn rows were constructed with PVC poles mounted with semi-regular 8 inch spacing, where the row width was 30 inches and row length was 15 feet. The outdoor corn row was 30 inches wide, 75 feet long at 8 inch spacing.

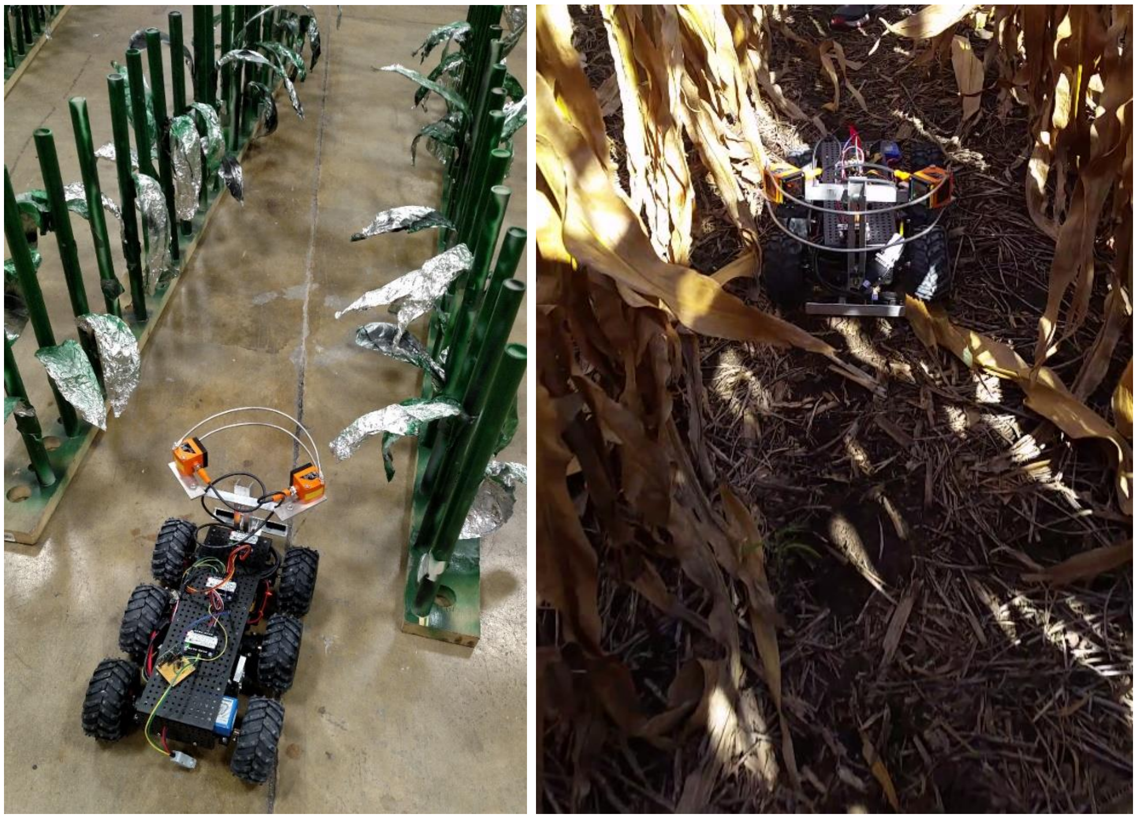


Figure 2.21 - Experiment layout for noisy input tests

2.3 Results and Discussion

Testing the filtering on the distance samples occurred in the simulated corn rows, where the robot started with smooth barriers on both sides, then transitioned to the corn row and back into smooth barriers again. The tests were an indication of mean controller stability and mean error. Error was simply taken as the difference between the two filtered distance samples since there was no other positioning reference. Around 20% of the test runs in simulated corn drove the robot into the simulated corn plants, indicating navigation failure. The failures did not appear to come from controller instabilities however, but rather from long row sections with overhanging leafs on only one side of the robot. The failed runs only occurred at slower navigation speeds of 0.4 m/s (Figure

2.22, Figure 2.23). Faster runs at 0.8 m/s displayed no improvement to the error mean, but produced no failed tests (Figure 2.24). Mean errors were gathered from only the samples contained in the corn row, see 5 – 16 second interval of test in Figure 2.22.

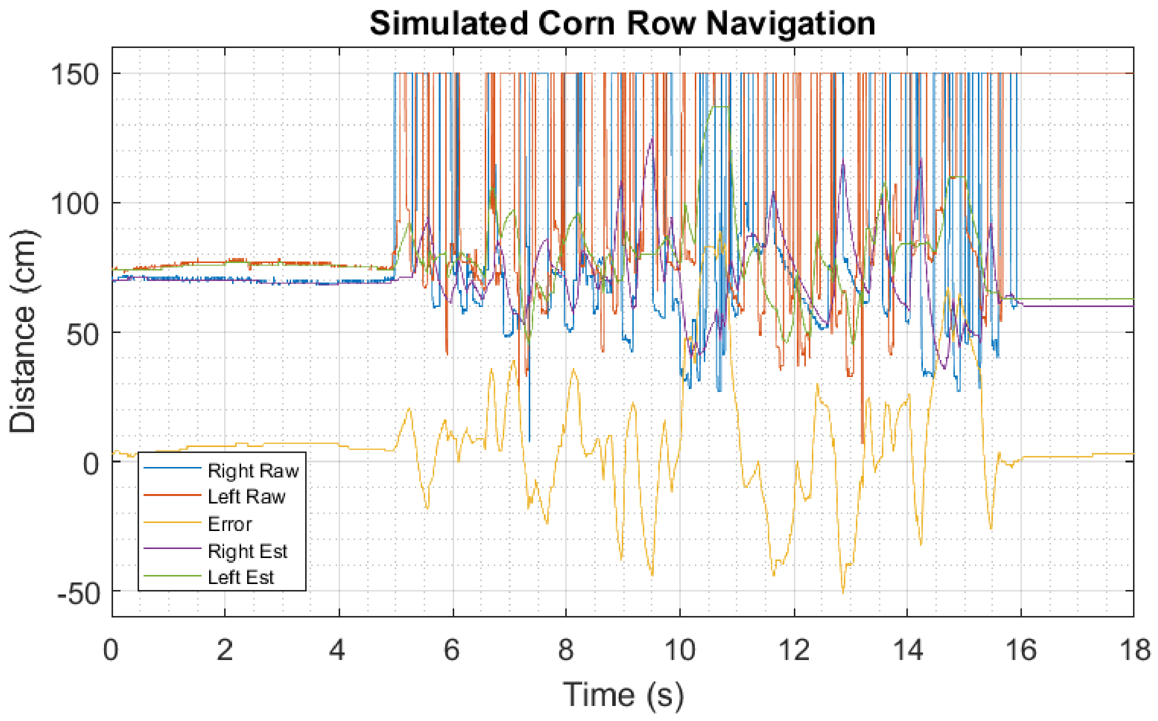


Figure 2.22 – Test at 0.8 m/s with 9.1 cm mean error

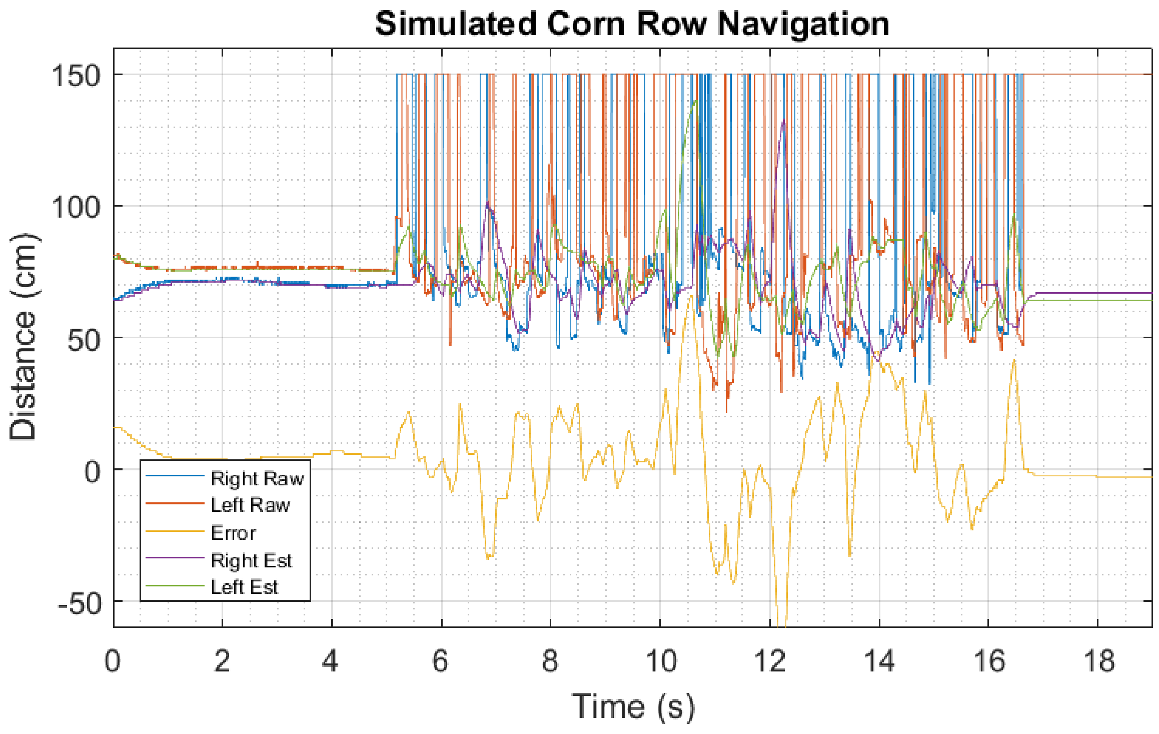


Figure 2.23 – Test at 0.8 m/s with 4.0 cm mean error

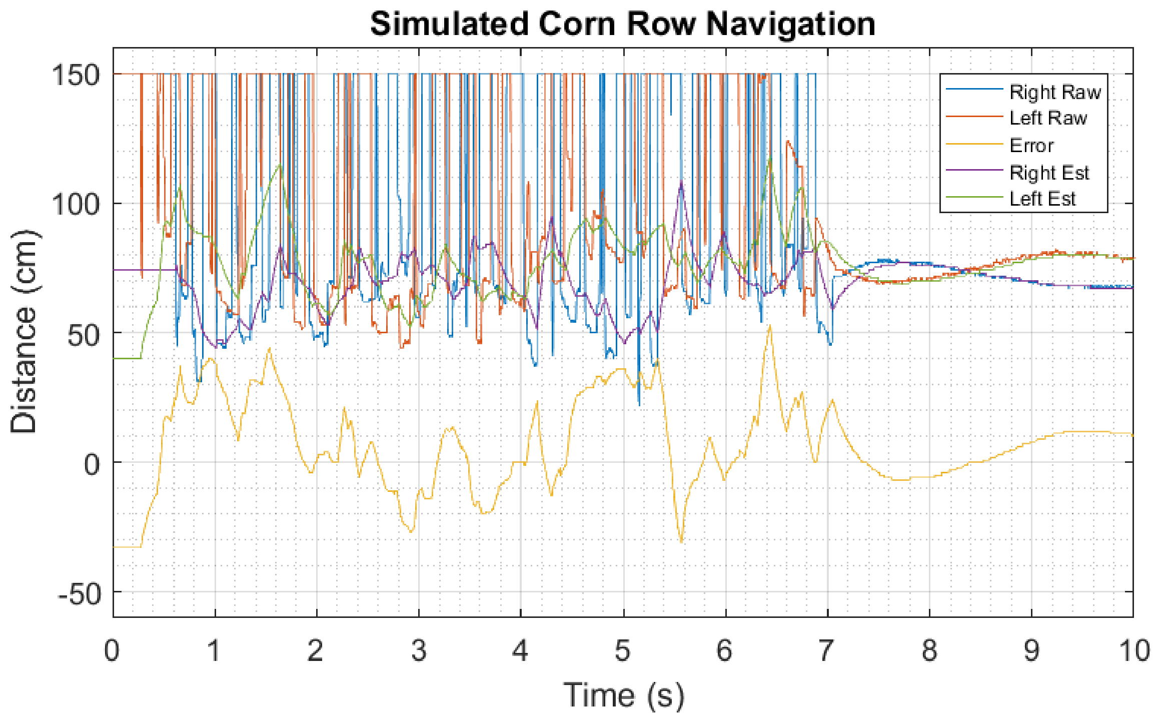


Figure 2.24 – Test at 0.8 m/s with 7.9 cm mean error

All test runs performed in the corn field were done at 0.4 m/s, and allowed for much longer tests (Figure 2.25, Figure 2.26, and Figure 2.27). The mean successful navigation time in the field was 39 seconds before some unrecoverable state was reached. The addition of uneven ground and in-row weeds larger than the robot were causes of failed tests. The mean errors from field tests were not significantly larger than in those of the simulated corn rows. This is likely an effect of the longer lengths of the tests.

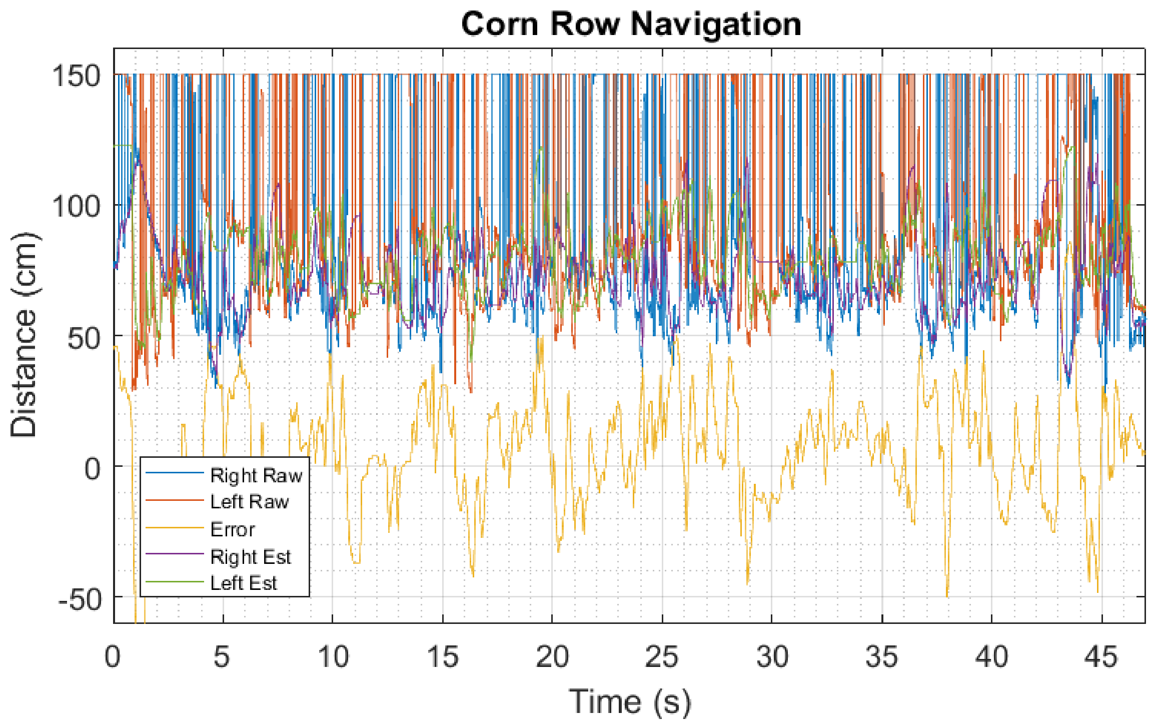


Figure 2.25 – Field test at 0.4 m/s with 5.8 cm mean error

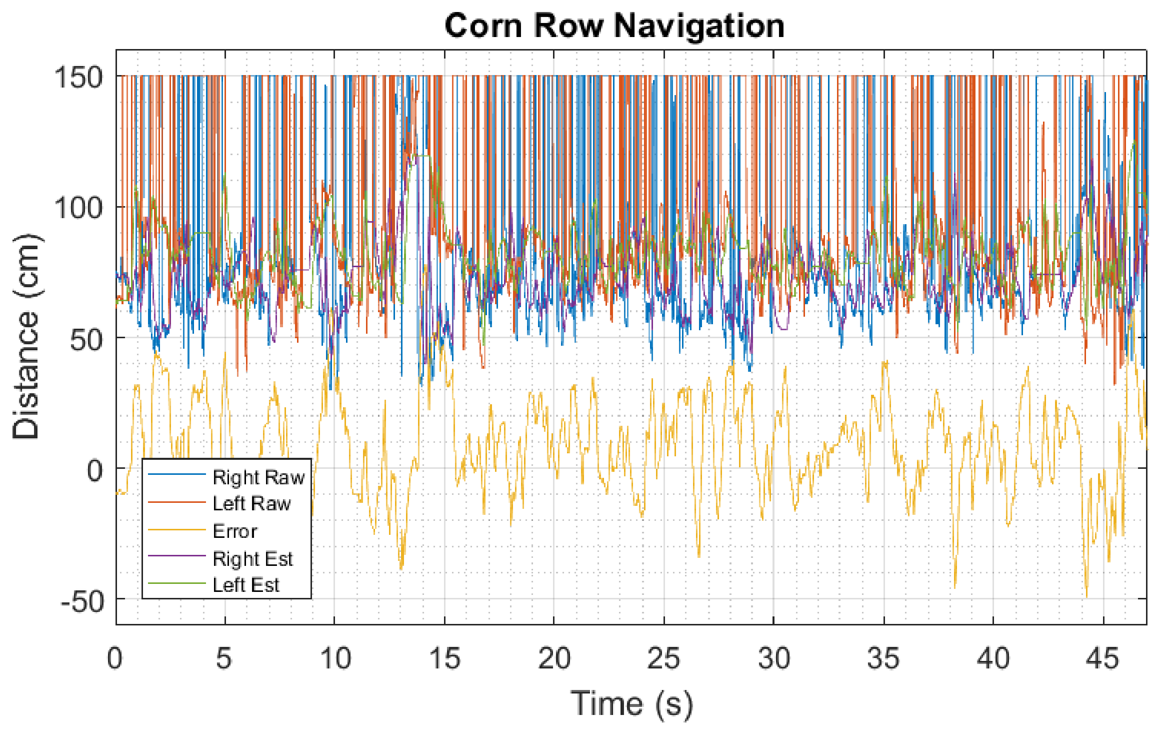


Figure 2.26 – Field test at 0.4 m/s with 8.4 cm mean error

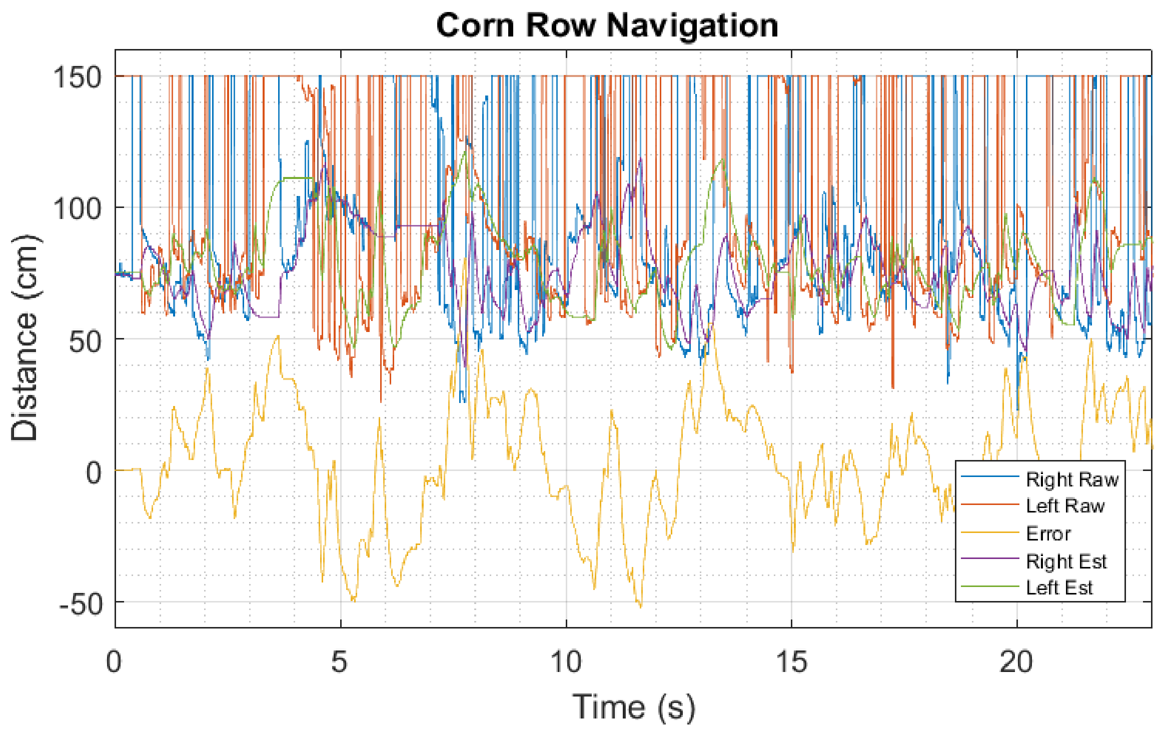


Figure 2.27 – Field test at 0.4 m/s with 5.0 cm mean error

2.4 Conclusions

The sub-canopy crop row continues to be an incredibly challenging navigation and localization environment for robots sized for the task. The recommendation from Tillet et al (1998) for supplementary sensing mechanisms for the crop row stands as a navigation approach that demands further exploration into weights of cost, computational effort, and performance between differing supplemental sensing technologies. This experiment showed that a low-cost sensor installation with low computational resources can provide short-term row navigation control for a large variety of row-following machines during GPS signal outages. The value in this low-cost, low computation GPS supplement is in the ease of integration for any row-following machine, and the low impact on system computation and communication resources. Supplementary sensing mechanisms to explore further might include low-cost cameras for machine vision row sensing (low-cost, high computational effort), 2D LIDAR sensors (high cost, high computational effort), and sensor platform stabilization gimbals for each of the sensors evaluated.

The transition from research machines to commercially viable row-following robots will be marked by the successful integration of cost-sensitive components that provide robust localization supplements to GPS signaling in the challenging environment of the crop row. The benefits of the successful implementation of these technologies will propagate from yield management of crops to the advantage of the producer and then finally consumers everywhere.

Chapter 3 Real-Time and Cyber-Physical Systems

3.1 Introduction

Consideration of the following topics is required for a meaningful understanding of the materials and methods sections of the document. These outlining sections are a complimentary guide of topics rarely included in an agricultural and biological systems curriculum. Engineering is the study and development of systems. Biological and agricultural systems are a little unique however, in that they tend to be complicated combinations of multiple, distinguishable subsystems – usually characterized by the other engineering fields (mechanical, electrical, chemical, etc.). Most research in biological systems even requires use of electronic sensing mechanisms and computational tools for processing data. This makes the field uniquely interdisciplinary – requiring the researcher to possess knowledge from a number of other engineering disciplines to further the research.

The research outlined in this thesis work is a part of the *beginnings* of an intersection of cyber-physical systems with the unique automation challenges in agriculture. The automation of agricultural machinery will continue to require more and more computing and signals knowledge as the demand for system integration, connectivity, and autonomy increases.

A cyber-physical system is the marriage of a computer system with a physical one, where each closely influences the behavior of the other. An understanding of real-time systems, the CAN bus, and feedback control systems will make the methods in this thesis work meaningful to the reader. A majority of treatment will be given to real-time systems and time-triggered CAN communications, since those are the topics least covered by the agricultural automation research literature.

Objective:

2) Elucidate problems addressed by real-time system design (from Section 1.1)

3.2 Real-Time Systems

Ordinarily, a system (some translation of input to output) can be characterized solely by the correctness of the system's output. In computing systems, only the low abstraction layers (capacitances and transistors) are characterized by physical limitations. For instance, the switching time characteristic of a transistor is dependent on the parasitic gate capacitances determined by the device's chemistry, geometry, and operating conditions. As long as this physical layer of a computing system is used within these switching time constraints (e.g. accounting for signal propagation delays), the abstraction layer is considered "correct", and all of the resulting upper abstraction layers of the system can be characterized by 'correctness factors' that are not centered on physical constants or the passage of time. The combinational logic, the finite state machines, and higher-level digital devices built on the physical layer can be characterized by logical response only, independent of their physical implementation (Figure 3.1).

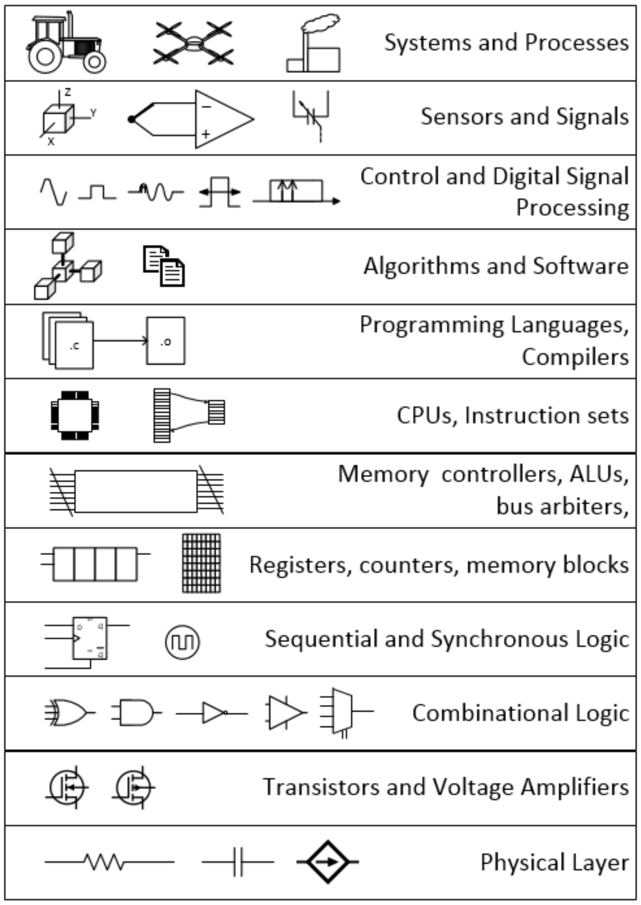

Systems and Processes
Sensors and Signals
Control and Digital Signal Processing
Algorithms and Software
Programming Languages, Compilers
CPUs, Instruction sets
Memory controllers, ALUs, bus arbiters,
Registers, counters, memory blocks
Sequential and Synchronous Logic
Combinational Logic
Transistors and Voltage Amplifiers
Physical Layer

Figure 3.1 - Abstraction Layers of a Controlled System

For example, the user of a word processing application pays no mind to the underlying digital systems of the machine, and does not much care if the response of the system to a keystroke is 2uS or 2mS. Even if the machine took a full 2 seconds to display the character, (which indeed, sometimes happens) the system is still considered ‘correct’ in its behavior. Some manner of input was given (‘J’ key pressed), and the correct corresponding letter ‘J’ was still printed on the screen for the user.

This sort of system response *variation* is unacceptable in many other systems. How quick to respond should the ABS brakes on a vehicle be? How reliable should the emergency-stop switch on a surgical endoscope be? What if the actuators on an assembly line took too long to stop operation during an emergency stop scenario? Some systems need to be made with the passage of real-time (RT) in mind. A real-time system (RTS) is a system that needs to meet its deadline requirements, or else suffer serious system breakdown or cause other serious consequences. Designing an RTS is ensuring the system meets its goals in the time domain, usually in the form of deadlines. This is the re-acquaintance of the overall system with the passage of time, something otherwise only considered on the low-level, physical layers of the system. A real-time system is evaluated on the correctness of its outputs and on the timeliness of the results. The output of an RTS should be correct, though it should also be “guaranteed punctual” to the degree of the system requirements or deadlines. Real-time systems are known to be difficult to design, and just as difficult to verify and test.

In the context of agricultural machinery automation, “correct” behavior can also be defined as more than just correct agricultural task operation. Considerations for system safety, reliability, responsiveness, and failure modes introduces a set of deadlines to be met for the system to be considered “correct” in its behavior.

Deadlines are sometimes also imposed onto a system from the physical dynamics and constraints surrounding the system. The deadlines are therefore not necessarily easy to define, and validation of an RTS can be extremely difficult. System design tools are not always configured to enforce hard deadlines of a system under design. For example, the programming language of C does not even have any construct or inclination of the progression of time. A C program can execute perfectly, producing correct output, and yet fail to produce the needs of the system's overall intent as a result.

Consider a handheld calculator that is always correct in the mathematical results it produces, but requires 4 seconds to compute a single addition operation, 8 seconds to produce multiplication results, and 2 hours to generate a sin(x) approximation. While the output of the device is indeed correct, the output is hardly "on time" from the user's perspective. The entire purpose of using the device is nullified by its failure to be *useful*. The point of using a handheld calculator is to get correct answers, but also to get them at a speed that saves the user from having to put in the time to compute the same results by hand. A calculator needs to be engineered to contain the appropriate computational resources for the correct behavior of the entire system – including the user's needs.

There is an important distinction to be made: optimization of computation time is not equivalent to real-time awareness. One is an end, the other is a means. If systems engineers wanted to improve the handheld calculator described above, they might want to improve the execution time of the computational tasks. This could mean changing computation algorithms, branching instruction count reduction, or increasing system clock frequencies. The continuation of this process eventually convinces the engineer to simply replace the hardware with a 2.0 GHz central processing unit (CPU), and call the device finished - albeit the device might now cost $200 and be a vast waste of computational potential and electrical power. RTS design is not simply a matter of "throwing" better hardware at the problem; it is about meeting output deadlines for the usefulness and safety of the entire system.

3.3 Cyber-physical Systems

A computing system classification that attempts to include “real time” as a parameter for correctness is a cyber-physical System (CPS). CPS study formalizes for the coupling between a computer and a physical process. Usually, a CPS has a physical process in a feedback loop (Figure 3.2). The computer and the physical process have an interdependence on the other for correct operation.

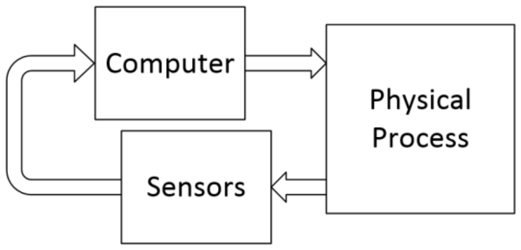


Figure 3.2 - Very Simple Cyber-Physical System Architecture

The physical process here could be a simple system of thermal dynamics, such as a boiler tank interfaced to a microcontroller with a temperature sensor and a heating element, or an amplifier with a well understood frequency response. Many physical systems however, are much more complex. Consider an ammonia manufacturing plant - a vast machine sitting on a square-mile of concrete, delicately wrapped in aluminum walls of a building. The entire plant might consist of several hundred sensors for temperature, flow, and pressure - and there might be just as many valves, switches, and motors used to act on the system. Without even considering the RT needs of the system, consider how a controller / computer might be able to interact with, sense, and control the machine’s manufacturing process. Should all the sensors and actuators be under the control of one centralized controller?

Surely it is clear that some physical systems are complex enough to be broken down into a number of subsystems – each with their own system dynamics. The ammonia plant as described, is not only physically spread out, but actually contains a number of sub-processes connected together. This is a good reason to consider the use of a *distributed* cyber-physical system architecture.

In many industrial applications, a distributed system architecture is used to handle many concurrent sub-processes in real-time. This allows the system to sense, manage, and control the highly variable and sometimes vast scales of differing system architectures. Distributing a RTS improves the scalability and maintainability of the end system. Distributed RTS (DRTS) architectures are widely found in manufacturing systems, process controllers, automotive and agricultural, robotics, marine, and aerospace systems (Lee et al., 2015). Figure 3.3 shows an example physical process with its various sub-processes or subsystems coupled to a distributed computing system. The figure goes so far as to abstract away the physical coupling and dynamics that are possible between the processes' various subsystems, as well as the dynamics of the actuators.

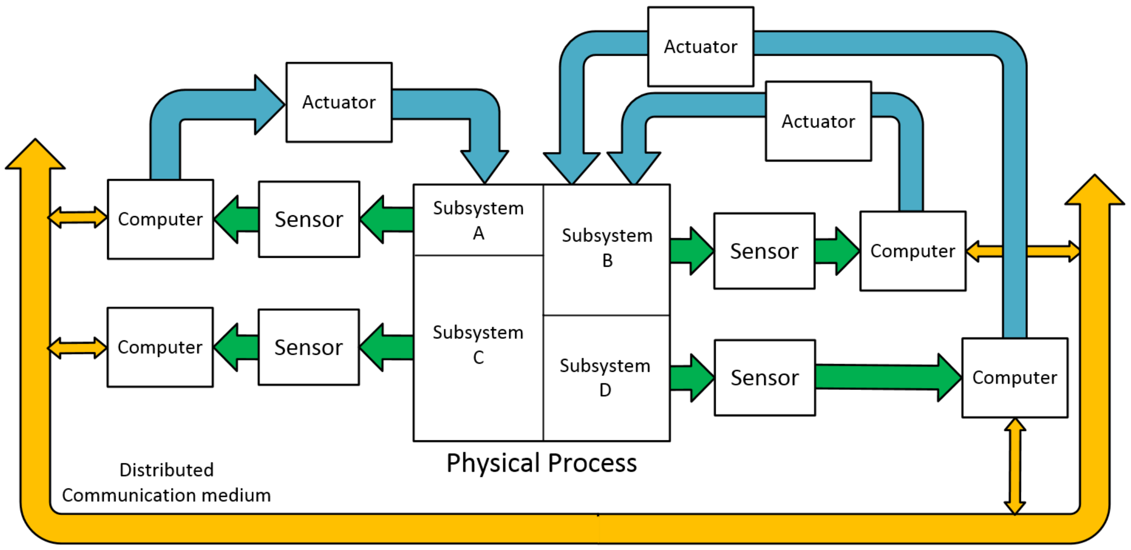


Figure 3.3 - More Realistic Cyber-Physical System

This results in changes to the software developer’s approach to solving problems and designing systems. The changes include a new awareness of the system’s dependability, safety, failure modes, output delivery guarantees, process triggering, and predictability / determinism. To assist systems engineers in the implementation and validation of these features, a model for distributed RT systems is often necessary. Using the model outlined by Kopetz (2011, pp. 80-81), we see a distributed real-time system model which separates the computational resources (components) for each of the sub-processes and subsystems into one domain of concern, and the communication medium that transfers messages between components into a separate domain of concern (Figure 3.4). The model also abstracts away any of the physical processes that exist behind each of the system components.

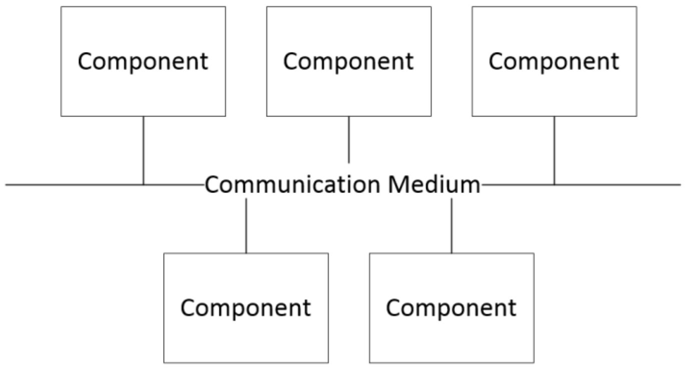


Figure 3.4 - Distributed Real-Time System Model (Kopetz, 2011, p. 81)

With the DRTS components and communication medium distinguished for separate consideration, the models' interactions between the two domains of concern can be characterized. In this model, multicasting is a requirement for component communication (Kopetz, 2011, pp. 80–81). Each component is able to send a message to multiple receiving components in a single message broadcast or transmit event. This is analogous to a radio broadcast over the air (a communication medium), allowing anyone with a tuned radio (receiver component) to receive the message. The transmitting component has no knowledge of which other components actually received and accepted the message. This is a requirement for the model because multicasting enables an external entity (such as an engineer with a logic analyzer, or a message recorder) to see all system messages and interactions from the same place. A consequence of the transmitting component having no knowledge of the reception status of the message is the need for system components to be aware of the passage of "real-time" for the purposes of error detection. A component should use its awareness of real-time to determine the difference between message transmission corruption and serious system errors, such as dead components or “babbling idiots” (Di Natale et al., 2012; Lari et al., 2007).

An additional requirement for the models’ communication medium is unidirectional atomicity (Kopetz, 2011, p. 81). This supposes that each component is able to transmit a message without the dependence on a properly operating receiver (other components). The message should be transmitted without the help or permission of another component. Finally, the message is to be transmitted completely, or not at all. There should never be partly-transmitted messages that arrive on the communication medium. This means that once a component begins message transmission, no other components can interfere with the process or interrupt the transmission process in any way. If the communication channel is unable to handle more than one message transmission at a time, then message atomicity implies that no component can transmit while one component is currently transmitting.

Consider an alternative means of connecting components: multiple communication mediums of which some are un-directional, and some bi-directional (Figure 3.5). Where would the systems engineer “look” in order to observe overall system behavior? It would be difficult to observe and understand the system when some component interactions are exclusive and operate under different rules and assumptions. If one component fails, an error chain is produced - making this system difficult to troubleshoot.

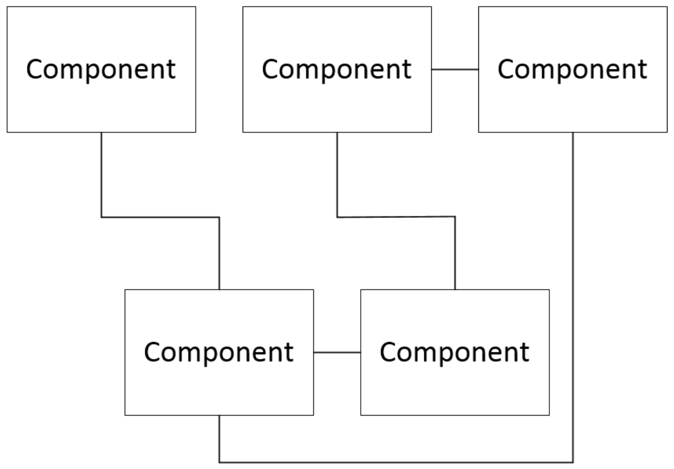


Figure 3.5 – Example Alternative to Distributed Real-Time Model

Chapter 4 Controller Area Network (CAN)

4.1 Introduction

The widespread use of the CAN bus in agricultural machinery warrants a description of the technology. The detail of the description here focuses on CAN from the perspective of its usability in controlled machinery.

Objective:

Provide a background for basic CAN bus system design (from Section 1.1)

4.2 Controller Area Networking

The message abstraction described by Kopetz (2011, p. 80) defines a set of requirements that enables the design of a reliable distributed real-time system (DRTS). The features described fit remarkably well with the Controller Area Network (CAN) bus, a centralized communications physical layer and link layer for harsh automotive environments. Developed in 1986 by Robert Bosch GmbH, the CAN bus has found its way into a large number of industries outside the automotive sector because of its low cost and many advantages as a communications medium.

To move from the abstracted domain of systems characterization to an actual robot control system, the discussion must shift from *communication mediums* to CAN *busses*, *messages* to CAN *frames*, and from real-time system *components* to CAN *nodes*. The equivalencies between these is obvious as the CAN bus becomes understood.

The backbone of many industrial real-time systems is the CAN bus. The CAN bus is a serial bus protocol ubiquitous to *many* automotive, industrial, and aerospace systems.

Being message-based, the CAN bus allows systems engineers to coordinate a number of subsystems (real-time *components*) of a larger system into performing a working singular function. The 2.0b CAN specification allows devices to arbitrate bits to the bus at a rate as high as 1 Mbps (Bosch GmbH, 1991). Since bits are the basic symbols on the CAN bus, one bit equals one baud. In a typical CAN bus, multiple *nodes* (real-time *components*) are connected to establish a network of devices that can communicate in a timely manner. The nodes are typically microcontrollers (MCUs) or field-programmable gate arrays (FPGAs) that interface to sensors, actuators, or communication bridges.

The centralized bus can be thought of as a shared resource that can only be "possessed" or "owned" by one node at a time. That is, if a node has a message to transmit on the bus, it must wait for the bus to be "quiet". This is strikingly analogous to the fixed-priority, non-preemptive scheduling technique known to the shared resource problem and the schedulability problem discussed in RT scheduling theory (Fuster et al., 2005; Meschi et al., 1996; Pedreiras et al., 2002), where the CPU is modeled as a shared resource that gets used by only one computational task at a time. In the case of the CAN bus, each node that holds ready data waits for a currently transmitting message to complete transmission, and further, waits 7 bit-quanta of time before trying to take hold of the bus. This is to force all nodes that are ready to transmit by this time, to begin a new transmission attempt concurrently - in the same instant. This allows one of the CAN buses' more interesting features to solve this shared - resource problem. To best illustrate this, we closely examine the structure of the CAN message packet (Figure 4.1).

The standard CAN data frame is structured similarly to other serial protocols, with a header, payload, and data integrity checks at the back-end of the frame (Cyclic Redundancy Check, – CRC). One particular field of note in the CAN data frame is the identification (ID) field. This field is meant to *identify the frame's contents*. This means that in the absence of a higher-layer protocol, the base CAN data frame contains no sender/receiver information. CAN IDs are source and destination agnostic. This 11/29 bit field will inform all system nodes about *what the message contains*, but it does *not* imply which node transmitted the message, and for which node(s) the message is intended. This means it is possible for two or more nodes to transmit messages of the same ID onto the bus, but this aggravates the shared resource problem and quickly causes Error Frames to appear on the bus. The reason why is highlighted in a description of the protocols' bus contention solution. This is the CAN answer to the shared resource problem.

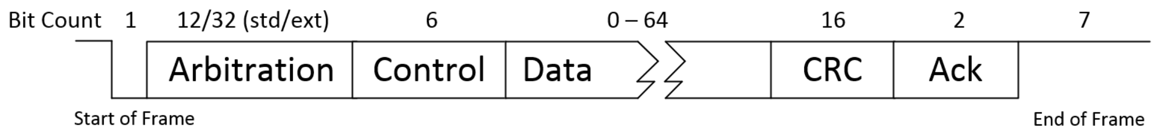


Figure 4.1 – CAN Data Frame (Di Natale et al., 2012, p. 14)

An important premise for understanding the bus contention process is that two or more CAN nodes with ready data frames for transmission are able to begin transmission at the same time, and often do. Seven bit-periods after the last message completes transmission (the inter-frame space), all nodes with transmission data 'ready' (since before or during the last message transmission) simultaneously begin the arbitration process. The arbitration process begins by the CAN Low line dropping to 0.3V (CAN High rises to 5V) for one bit time quanta. This is the Start of Frame bit (SoF). When SoF is detected by other nodes on the bus, the bus is now considered to be “taken”, and no other nodes make an attempt to transmit until the CAN frame transmission completes, and 7 more bit periods pass. Any node which becomes “ready” for data transmission after the SoF remains in a receiving-only state until the next inter-frame period completes.

Supposing two or more contending nodes drop the SoF bit to the bus simultaneously, they each suppose that the CAN bus is possibly theirs to transmit on. This is not a bus error condition, and the arbitration continues on by each of the contending nodes asserting the bits of the ID field of the message they intend to transmit in most-significant bit first order (MSB). As each next significant bit of the message ID is driven to the bus, each of the contending nodes except one (the winner of the bus), eventually gives up its attempt drive bits onto the bus (Figure 4.2). Only one node completes transmission of the entire ID field. The process that causes this effect occurs on a very low hardware level.

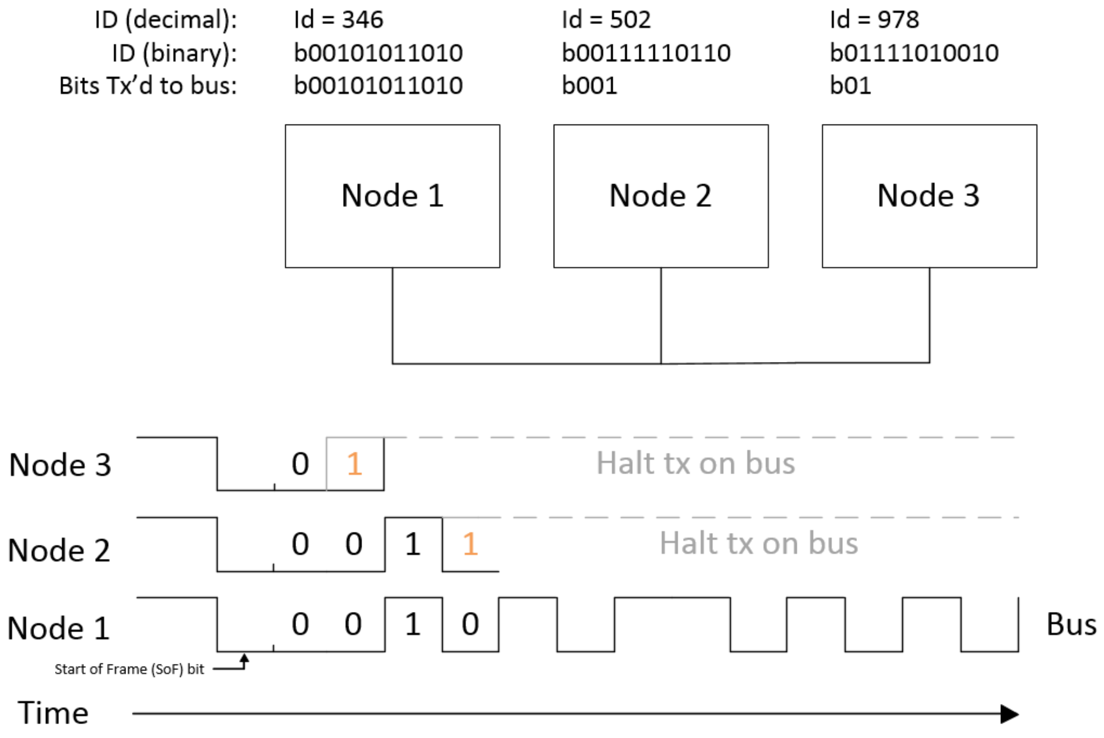


Figure 4.2 – CAN Arbitration with ID Field Contention (Di Natale et al., 2012)

Every time a node shifts out an ID bit to the bus, the underlying digital logic monitors the resulting state of the bus – checking that the bus state matches that of the bit driven forward. If the controller sees the bus voltages differ in this instant from what it ought to be, it assumes that another node is also trying to transmit its message ID. More specifically, each contending message is simultaneously arbitrated to the bus in MSB order, until an ID field's "1" bit results in a bus at a "0" state, or a "0" bit in the message ID results in a "1" on the bus (Kopetz, 2011).

Numerically, this means the message with an ID that contains the least 1's in the most significant bits of the ID field wins the bus at the end of arbitration; that is, *the contending CAN message with the lowest unsigned numerical value in the ID field wins the bus*. So between two contending messages, the first message to attempt to drive a "1" from the ID field on the bus, gives up arbitration and allows the message with the lower ID field to continue transmission.

It should now be evident why two contending messages with identical IDs should cause an error on the bus, since the two messages are not guaranteed to contain the same data payload, while coming from two different CAN nodes. It ought to be clear that it is unwise to design a system where two or more nodes are allowed to transmit messages with identical ID fields. This is why some high-layer protocols such as SAE: J1939 reserve some of the least significant bits (LSBs) of the ID field to designate unique source and destination node addresses defined by the systems engineer. The idea here is that the most significant bits (MSBs) determine bus priority (as per usual), but that contention is secondly determined by a field called PDU format (next MSBs), thirdly by the destination field, and lastly by the source address field (ISO:11783 Part III, and SAE J1939) in the LSBs place. This preserves the requirement for unique ID bit fields across all system-wide messages.

Considering the ubiquitous occurrence of ISO: 11783 in agricultural systems, most systems manufacturers use only a subset of the protocols features and message types. From the perspective of an embedded systems developer, it is easy to see why. A typical web search for a J1939 stack in C returns proprietary software libraries with significant costs, both in funds and system memory. Some J1939 stacks are found to be as large as 5kb. This likely contributes to the ‘partial’ adoption of these standards for resulting systems. Only a subset of the standards’ messages is typically found in use on a given system - used alongside a number of proprietary messages developed by the manufacturer.

The CAN communications state machine built into the node uses message ID fields for priority levels as a solution to the shared resource (bus) problem. By defining message IDs as an inherent possession of a unique priority for CAN bus ownership, the nodes can all agree on which contending message ultimately gets rights to the bus, and which messages have a wait to try again later. While this priority-based message arbitration is powerful, it only happens when two or more nodes drop the SoF bit within the same bit quanta (for a 250k bus, the nodes would have to drop SoF within the same 4 nS period). A system should not be designed to rely on this occurrence, since worst-case message transmission latencies become difficult to predict. Where there is a lack of system determinism, there is a lack of real-time system guarantees. Figure 4.3 demonstrates the difficulty of predicting the transport latency of message with ID 253 contending for the bus through Node A.

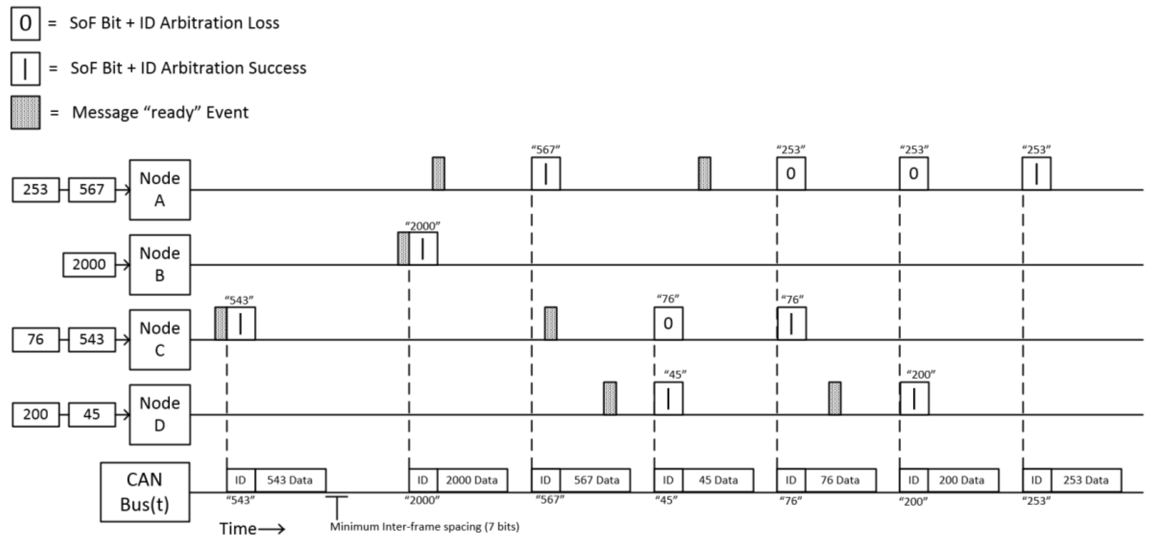


Figure 4.3 - Expanded demonstration of bus arbitration between multiple nodes

With CAN message priority as a direct function of the ID field, the systems engineer could define relative priorities to messages which are planned for the bus. Though as will be pointed out, the message ID should identify the contents of the message's data fields, and no more. These definitions are left up to the systems engineer. The inclusion of source and destination information if needed, should be encoded into the message data payload fields, and not the ID field.

4.3 System Design Using Time-Triggered Controller Area Network

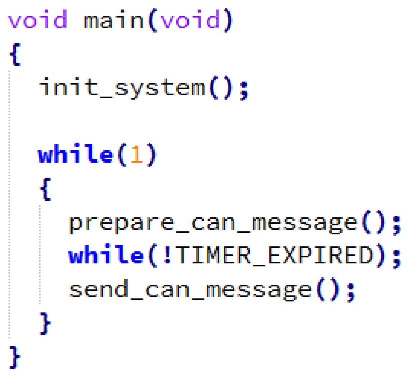
It turns out there is a more systems-conscious approach to CAN bus management that does not involve reliance on data frame ID fields for access to the bus. One solution for increased bus determinism and reliability is a time-triggered (TTCAN) approach to CAN-based DRTS design. TTCAN was developed as an extension to the original CAN specification as ISO11898-4 (Lawrenz, 1997).

Before understanding how time-triggered systems work, it’s helpful to examine the alternative “triggering” mechanisms used in modern CAN systems. There are at least two approaches to the design of a CAN bus that should be mentioned, and then avoided at all costs in a DRTS design.

4.3.1 Self-Triggered Systems

The first of these is what could be called the “Blind Node Bus”, or the Self-Triggered System. With the use a CAN bus monitor, an examination of the message activity on a commercial CAN system reveals that most messages on the bus appear periodically, though not always with the same frequency. Some system messages may appear every 100 mS, while others appear every 2 seconds. This might lead a human observer to suppose that each node is independently transmitting messages periodically on its own time-base, or whenever new data is ready for the bus. This would, after all, be possible since bus collisions are avoided with the message arbitration protocol.

One attempt at producing such behavior might be the following pseudocode (Figure 4.4) in the node’s firmware.



```
void main(void)
{
  init_system();

  while(1)
  {
    prepare_can_message();
    while(!TIMER_EXPIRED);
    send_can_message();
  }
}
```

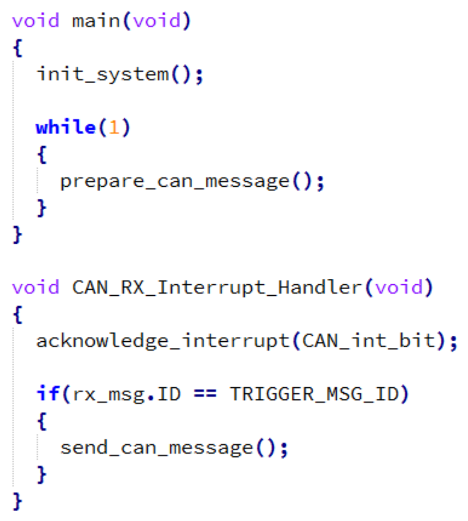
Figure 4.4 - Blind Node Pseudocode

The Blind Node firmware relies heavily on the message arbitration process, and assumes that each message on the bus has a unique ID field (no other nodes ever try to transmit using the same ID). In addition, this approach to message transmission is not network-aware, nor is it real-time aware. The firmware creates node behavior that is completely agnostic to other network activities, and is unable to enforce a deterministic reception of its message by the intended receiving node(s). This approach also includes no method of knowing how accurate its sense of time is, relative to the accuracy of the other nodes' sense of time.

Temperature fluctuations alone could be enough to cause the node to attempt transmission at deviant frequencies without any mechanism to inform the node of such errant behavior. Reception of CAN messages and commands from other nodes do not appear to be handled, and it is unclear where and when they would be, should the node need to receive messages. Blind Node firmware does nothing to time-align or correlate phases of its transmission with data instances already on the bus. Is the transmitted data relevant to events that occurred on the physical system 20 mS ago? Or those that occurred 2 seconds ago? This is the result of losing message idempotency (Kopetz, 2011, p. 124). Blind nodes treat all received messages the same regardless of when they arrived.

4.3.2 Event-Triggered Systems

The second approach is an effort to avoid the self-triggered approach by adding a small amount of "system-awareness" to each node on the bus. This approach becomes tempting to use when the CAN node has either a small number of messages to transfer *to* the bus in *response* to data read *from* the bus. Consider an example where a CAN node reads a sensor and transmits relevant data to the bus. But to retain some time-association of the payload with other messages on the bus, the controller waits for some specific message to appear on the bus as its trigger (an *event*). Only trigger-message reception causes the CAN node to attempt transmission. Replicating this transmission approach across the entire system allows for bursts of data to appear on the bus, with each message appearing on the bus in the order specified in the chain of nodes that trigger other nodes. Event-triggered pseudocode firmware follows this simple logical flow in Figure 4.5.



```
void main(void)
{
  init_system();

  while(1)
  {
    prepare_can_message();
  }
}

void CAN_RX_Interrupt_Handler(void)
{
  acknowledge_interrupt(CAN_int_bit);

  if(rx_msg.ID == TRIGGER_MSG_ID)
  {
    send_can_message();
  }
}
```

Figure 4.5 - Event-Triggered Communications Pseudocode

This approach assumes that each node has data perpetually immediately available, or *ready* at all times, since there is still no explicit construct of time in the hardware or software. Perhaps the first node in the message chain transmits the first trigger message similarly to the self-triggering scheme previously described. But the associated problems of such an approach cannot be assumed to be handled properly with the construct of real-time. In the event-triggered approach, continued and reliable CAN network operation becomes entirely dependent on the correct reception of messages of other system components, other CAN nodes – meaning that each node only operates correctly because of some external dependency. This directly violates the communication medium requirements for the distributed real-time system model laid out by in Section 3.3.

> In an event-triggered system, error detection is in the responsibility of the sender who must receive an explicit acknowledgment message from the receiver telling the sender that the message has arrived correctly. The receiver cannot perform error detection, because the receiver cannot distinguish between *no activity by the sender* and *loss of message*… The sender must be *time-aware*, because it must decide within a finite interval of real time that the communication has failed. This is one reason why we cannot build fault tolerant systems that are unaware of the progression of *real time*. (Kopetz, 2011, p. 91)

The problem with breaching this violation becomes clear after considering what happens to a system designed entirely in an event-triggered communications approach. The unpredicted loss of one system node produces an error chain that propagates down to the most-dependent node, still waiting for an event that is either unlikely to occur, or occur unpredictably. Put simply, this approach is not robust to failure of other system components. Failure of one node in the message chain causes a trickle-down of system failure and no single node has any reason to assume responsibility for repairing the message chain. The occurrence of this malfunction is called an error propagation.

Message data preparation in event-triggered buses includes a generation-to-transmission latency, or a *transport latency* which has an effect on the system that is entirely at the mercy of the message arbitration process and the processing time required to handle message reception events. Both of which are poorly characterized processes since bus arbitration loss for any non-top-priority message can occur for an unpredictable number of instances, and the required processing time for any given message depends on the processor and the application (Kopetz, 2011, p. 178). Software may request that the hardware begins the efforts to place a message on the bus, but a node's lack of system knowledge results in actual message transmission at "some point in the future, " including a significant amount of jitter depending on the bus load. So while software may request message transmission, the execution of the request is still performed by the CAN hardware, which must contend with the rest of the system components in a competitive environment over a limited resource. This makes for a non-deterministic system - unsuitable for real-time applications where the control of machinery is involved.

Finally, it should be noted that event-triggered systems do not operate on the progression of real time. The worst case transport latency of an event triggered message is sum of the transport latencies for each message in the system, since simultaneous events presented to the system can trigger an avalanche of messages on the central bus which can lead to communication medium overloads (Kopetz, 2011, p. 178). This scenario is well illustrated for the possible instance on the bus where all system messages become “ready” with data in the same instant.

4.3.3 Time-Triggered Systems

Time-Triggered architectures are a preferred approach towards safe, reliable, and deterministic system design (Lawrenz, 1997, pp. 26–27). Many complex, controlled systems employ the signal processing techniques of discretely sampled system states from periodically queried sensors. This fact hints at the potential benefits of using a communication protocol which is also highly periodic in nature. Since the signal processing and control algorithms run on periodically sampled system states, a strong portion of the state communication protocol should be assumed to be periodic. System events however, only require sporadic message transmission for signaling changes in the system – meaning the same protocol should also deliver sporadically occurring messages. In a time-triggered system, there is still room for event-triggered messaging, but only in a very specific way (Di Natale et al., 2012, p. 210); namely the communication of events.

1) Event-triggered messages then become more narrowly defined as a message unit meant to signal the occurrence of system-wide *events* or *changes* to the system.

2) A time-triggered message is then defined as a message unit for the periodic communication of system-wide *states* (Kopetz, 2011, pp. 90–91).

Distinguishing these types of messages enables the systems engineer to layout a very predictable messaging schedule obeyed by all system components. This converts the system from a machine that responds to the occurrence of events, to a machine that responds to triggers defined in the context of the passage of real-time (Kopetz, 2011, p. 91). This is not to claim that event-triggered messages are prohibited in a safe system architecture, but that they should not be the basis for system-critical information transfer. This treatment of time-triggered communication ultimately assumes the inclusion of event-triggered messages as a part of a higher communication protocol construct called the *basic period* (BP).

A time-triggered communication bus is typically a periodic arrangement of *scheduled* messages. Each integrated system component can safely participate in this scheduled exchange of messages with the knowledge of the system time, and the schedule to be kept for its highly *specified, defined, and enforced pattern for message transmission*.

4.3.4 System Time

The system-wide basis for the measurement and tracking of time is called the ‘global clock, or the ‘system time’ (Kopetz, 2011, p. 52). This is the time basis used to enable real-time awareness in each of the system components. The unit in which a system-wide clock should be measured is in physical multiples or fractions of seconds of time (seconds, milliseconds, microseconds, nanoseconds). Using CPU cycles to track time, or other relatively arbitrary event-period units makes little sense especially when the system in question interacts with physical objects in real-time which is normally measured in seconds, like those found in a CPS (Kopetz, 2011, p. 52). Ideally, the system time is maintained in each of the system components, and exists either physically in digital hardware, or virtually in software. An additional ideal is that measurements of the system time from within any of the system's components are identical to system time measured from any other component; that is, each of the system components agrees on what the system time is - to a fine level of resolution, even when each of the measurements come from independently maintained clocks. The system clock is typically just a counter which gets incremented or decremented very periodically. Tick frequency of the system clock should be selected to achieve a fast response to schedule progression, but not to disregard the "Reasonableness Condition" described by Kopetz (2011, p. 58). Capturing fast events and sticking to a tight messaging schedule requires a fine granularity of the system clock. Fast dynamics of a CPS require higher system clock frequencies, while slow system dynamics allow system clock frequencies to relax.

4.3.5 Clock Synchronization and the Time Master

In implementation, the internal system clock fails to meet the ideal system-clock characteristics, thereby failing to support the system's real-time requirements. Even just the thermal dependence of electrical system dynamics and manufacturer variability ensures a system-clock drift and offset to be present on any DRTS. Temperature-compensated oscillators as internal system-time references are increasingly a viable alternative for increased time keeping accuracy, but are still insufficient without some kind of time correction algorithm that enforces synchronization between system components for indefinitely long periods of operation.

The addition of an external time source or a “time server” for time corrections can enable highly accurate, low-drift system-clock implementations. This works by the inclusion of a time master – defined as a system component that acts as a centralized source of system time (Kopetz, 2011, p. 68). While the system time can be derived from a number of different sources such as the Internet (a source of multiple time references) or global positioning satellite networks (a source of coordinated universal time (UTC) time), the system’s time-master’s primary functions are to receive time data from the time server, convert it to system time, and serve the system time to the entire system of components. The time master converts some external or internal clock source into system time, then distributes that system time to the entire system uniformly. This distributed system-time is used by each of the components to derive and compensate for the time differences between the time master and the component’s local version of system-time. This centralized time server method is a reliable way to ensure that each system component simultaneously receives identical clock updates.

The message used for distributing the system time is then defined as the *time reference message* (TRM). Clock synchronization and correction occurs when a system component periodically receives the TRM from the time master. In the DRTS with a centralized communication bus, the TRM can be effectively received by all nodes simultaneously. The reception of the TRM by any system component also marks an epoch in the scheduled round of messages.

Consider the DRTS system in Figure 4.6, where the time master generates a TRM from the output of the time server, and transmits the time to components A, B, C, and E. Each of the system components includes an internal time reference (crystal oscillator) used to track the system time in-between reception events of the TRM. The internal time references differ slightly from one another, and vary depending on the environments' temperature, so the reception of the TRM ensures all the internal components completely agree on the system time – limting the effects of unchecked long-term clock drift. Psuedocode that shows a simplified TRM reception with time adjustment and message transmission scheduling based on the system-time is shown in Figure 4.7. A more accurate method of clock correction would use the time-stamp of the TRM SoF bit, which arrives at each CAN node within the same bit-time, indpendent of the message reception hardware and corresponding interrupt handling latency.

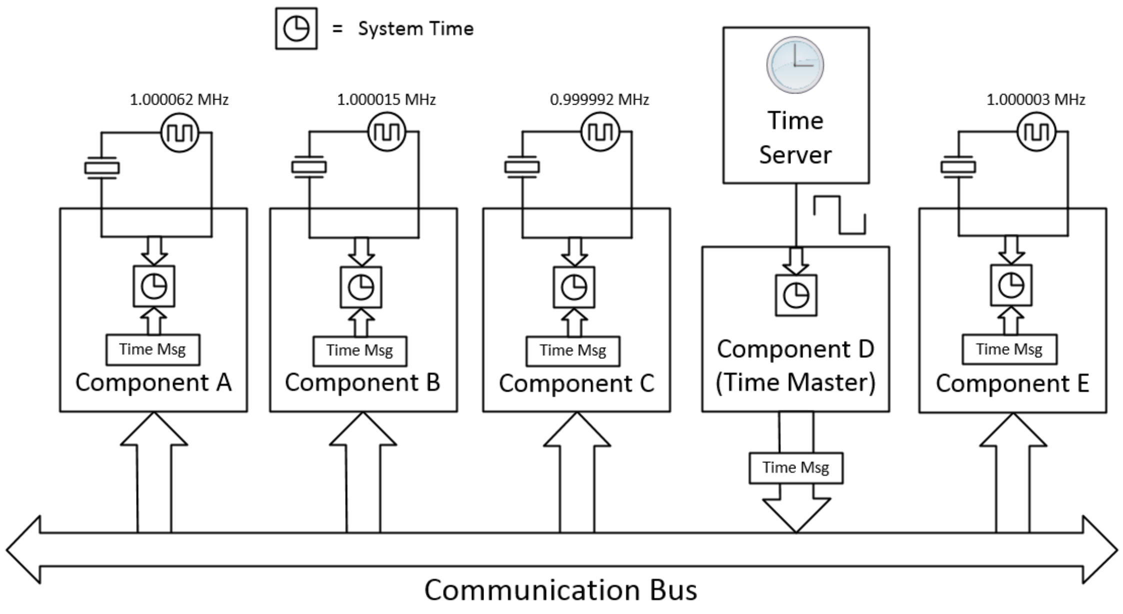
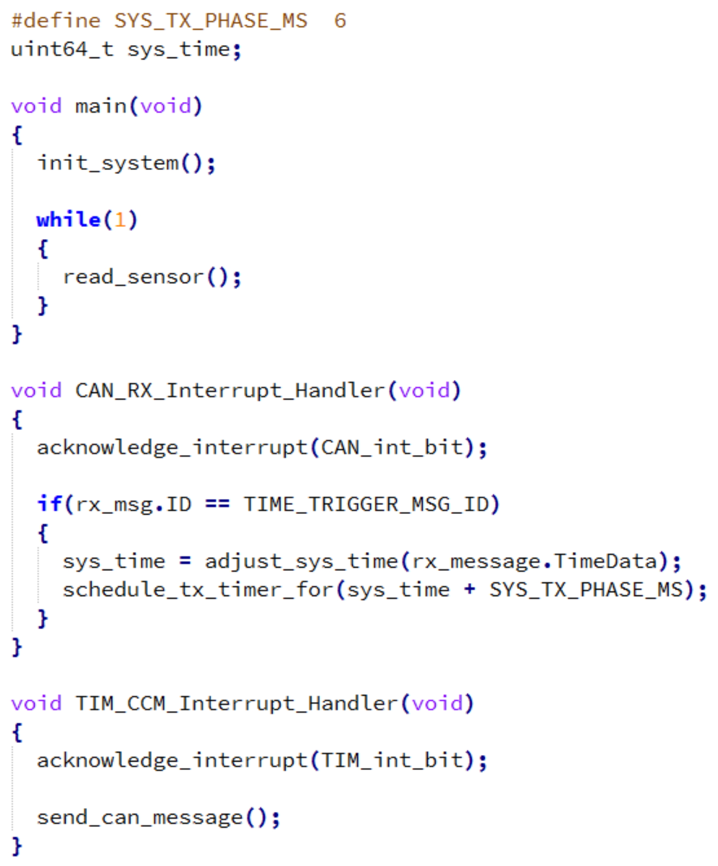


Figure 4.6 - Distributed RTS with time master



```
#define SYS_TX_PHASE_MS  6
uint64_t sys_time;

void main(void)
{
  init_system();

  while(1)
  {
    read_sensor();
  }
}

void CAN_RX_Interrupt_Handler(void)
{
  acknowledge_interrupt(CAN_int_bit);

  if(rx_msg.ID == TIME_TRIGGER_MSG_ID)
  {
    sys_time = adjust_sys_time(rx_message.TimeData);
    schedule_tx_timer_for(sys_time + SYS_TX_PHASE_MS);
  }
}

void TIM_CCM_Interrupt_Handler(void)
{
  acknowledge_interrupt(TIM_int_bit);

  send_can_message();
}
```

Figure 4.7 - Simple time-triggered message transmission

4.3.6 Messaging Schedule

To meet the highly periodic needs of system state communications, a system matrix is developed. The system matrix describes the periodic messaging schedule in the time domain. However, there are usually two levels of periodicity built into the matrix. The first is that the entire system matrix is a timeline which is itself, repeated. This is analogous to a week of time passing, because the second periodic aspect of the system matrix is in the basic period, which is repeated within each passage of the system matrix.

4.3.6.1 Basic Period

The basic period (BP) is a unit of time described by some integer number of system time ticks, and is an unchanging system-wide constant defined as the time between consecutive TRMs (Lawrenz, 1997, p. 211)(Figure 4.8).

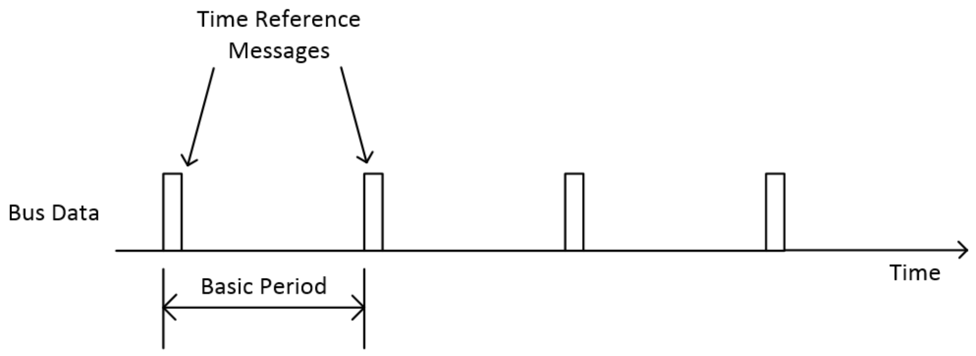


Figure 4.8 – The Basic Period is measured between TRMs

During the offline system design process, every instance of a BP is divided into an arranged sequence of messaging windows that define the type of messaging allotted during each window. There are three basic types of messaging windows used in a basic period:

1) Exclusive Schedule Window (ESW)

2) Sporadic Arbitration Window (SAW)

3) Free Window (FW)

Additional variants of these window types can be allocated for a basic period design, but should only be used to serve some unique, system-specific needs of the design. The number of instances of each window type allowed in a single BP is not limited to one, as is depicted in Figure 4.9. The ESW is a period when only a pre-determined sequence of scheduled messages may propagate to the CAN bus in a timely manner. This is the same window in which system states are pushed to the bus. In this window, each message can only be transmitted to the bus during its own exclusive time slot. If the node misses this time slot, it keeps the message off the bus to prevent collision with other scheduled messages in the ESW.

The SAW can be allocated for sporadic messages, meant to communicate system changes and events. Since events such as button presses do not need to be continually communicated, the SAW can be used to allow ordinary bit-wise CAN message ID arbitration to give the bus to whichever message arbitrates first, or arbitrates with the lowest ID. Scheduled messages that missed their time slot in the ESW can use the SAW to send data even if late. This is a possible error detection mechanism.

The FW is used as a system design margin, and enforces a limit to the overall bus load. Messages that appear on the bus during this period are instant indicators of a system error, enabling the offending node a chance to off-bus at the request of other system components. System nodes such as bus monitors can employed to watch and enforce the schedule in a TTCAN protocol. Bus monitors can even be placed in control of a system component’s access to the CAN bus, enabling remote physical off-bussing of offending and non-complicit nodes such as babbling idiots (Di Natale et al., 2012, p. 214).

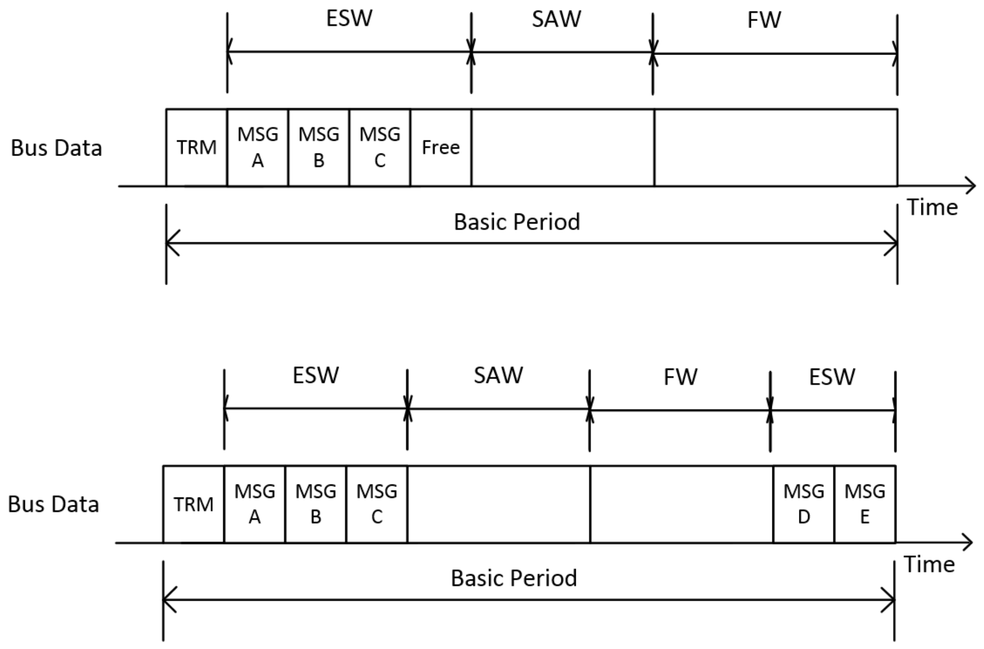


Figure 4.9 – Two example arrangements of the basic period

4.3.6.2 System Matrix

A system matrix represents the periodic communications schedule of the system; it is the highest level of periodic activity on the bus, as it is a representation of a cycle-of-cycles, or rather, a schedule of basic periods - each of which are a schedule of message time slots. Just as a month is a cycle of weeks, a system matrix is a cycle of basic periods.

A system matrix is typically a statically-sized length of real-time transmission windows that can be characterized by its transmission frequency (Freq_SM) and dimensions. The dimensions of the system matrix are determined by the number of basic periods (Num_BP), and the number of message windows in the constituting basic periods (Num_MW) – making the dimensions of a system matrix sized as a [Num_BP x Num_MW] matrix of real-time units (uS, mS, S). This requires the length of all comprising basic periods be equal, even if their transmission window arrangements are not identical to each other. For example, if the two difference basic period arrangements in Figure 4.9 had equal lengths (equal to Num_MW real-time units), they could feasibly be part of the same system matrix, each with any number of instances (Num_BP).

Consider the example system matrix in Figure 4.10, which accommodates 5 messages between 3 transmission frequencies. Note the allocation of ESW, SAW, and FW periods in each of the basic periods. This example system matrix shows the entire system schedule – a complete accommodation of system message needs, and full allocation of basic periods including FW periods for system expansion. This matrix represents a relatively simple time-triggered system organization.

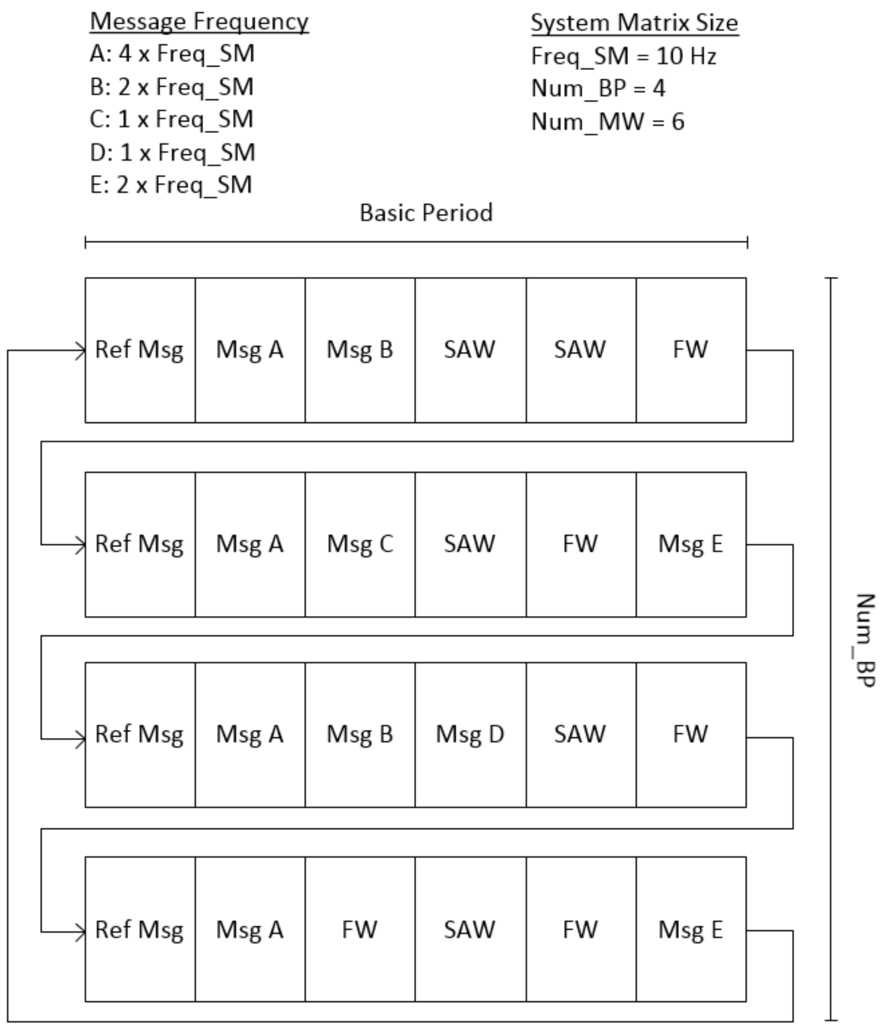


Figure 4.10 - Example of a system matrix (Lawrenz, 1997, p. 31)

4.3.6.3 Scheduling Techniques

The development of a TTCAN schedule can be done offline or online, and can be designed as static or dynamic. The goals of the schedule include the guaranteed allocation of message slots in the system matrix, for all sporadic and periodic system messages - each with some transmission frequency. Scheduling techniques are thoroughly discussed in the literature, and a review of Chapter 1 points out some of the more commonly explored techniques, including dynamic earliest-deadline-first (EDF) methods.

One scheduling method relevant in this work uses the set of time-triggered message frequencies to derive a schedule that ensures the schedulability of time-triggered messages and a static, unchanging system matrix – simplifying schedule management for each of the participating nodes in the network. The method requires each message frequency in the message set, be an integer power of 2 of the lowest message frequency (F_L). Consider the message set given in Figure 4.10, and assume the lowest and highest message transmission frequencies are F_L and F_H, respectively.

To ensure the scheduled occurrence of messages with transmission frequency F_L, the transmission frequency of the system matrix is set equal to F_L. Scheduling messages with transmission frequency F_H is ensured by allocating a transmission window in each basic cycle where:

$$Num\ BP = \frac{F_H}{F_L}$$

As depicted in Figure 4.10, the window arrangements of the included basic periods of a system matrix do not need to be identical, and the inclusion of at least one instance each of ESW, SAW, and FW periods is also not required, but is recommended for the minimization of event-message latency and the maximization of system matrix expandability for future system matrix additions.

4.3.7 Fault Tolerance, Missed Deadlines, and Error Handling

The intent of time-triggered communications do not address the reliability of the individual nodes on the bus. Rather, the focus is on the ability to detect and handle errors caused by individual nodes. This is an acknowledgement of the instances where system failure can originate from poorly behaving system components. In assuming a non-zero probability of some component failure, the design of a robust bus can be made to handle the failure, either enabling a fail-safe or fail-functional system response. Fault detection and handling in event triggered and time-triggered systems are treated heavily by Kopetz (2012).

Node failure can assume a variety of modes, and can be handled appropriately according to the mode. It is important to point out, that node failures can be easily detected on a time-triggered system as opposed to the self or event-triggered systems. The modes of node failure are easier to identify when system components are "system-aware", in that they possess knowledge of the system-matrix. Any detected deviation from the expected pattern of the system matrix can then be characterized as some node failure under some failure mode:

1) Dead nodes – A mode of component failure that indicates an absence of messaging activity from the bus. This can result from halted firmware execution, unreliable power, or loss of a physical connection to power or the bus. Node death is difficult to recover from without physical system inspection, meaning that the system has to resort to either a crippled mode of operation or a fail-safe mode, depending on the relative importance of the data provided from the absent messages.

2) Lost nodes – Indicates that a node is lost in the schedule. A transmission effort is detected (the correct messages are being produced), but the messages do not adhere to the system matrix. This naturally results from the loss of correct system-time either from missed time-reference reception, or a damaged internal time reference. This failure mode is very dangerous to system health since the node forces its messages to the bus in a way that forces other nodes to compete for their own allotted transmission time windows in the standard bit-wise arbitration fashion. This forcefully causes other nodes to miss their message schedules. Handling this mode of failure requires a fast response in the form of a power reset to the node or, in extreme cases, a physical disconnect from the bus in an effort to preserve the still-functional parts of the system matrix.

3) Babbling Idiot – This mode of failure is similar to the lost node mode, but far more extreme in scope. Usually caused by a software bug, a node might enter a looping state that randomly and frequently transmits garbage or randomized CAN packets to the bus. This blocks error-handling messages and other important time-triggered messages from running on schedule, and can even interfere with transmission of the time-reference message. Detection of the babbling idiot failure can be difficult to detect and difficult to respond to, since the effect is akin to the theft of the microphone by an auctioneer at a presidential debate. While handling the condition is similar to the how lost nodes are managed, timely detection can be the difference between a robust and non-robust error handling mechanism.

Detection of node failure can be performed by any or all of the system components. The offending behavior of a babbling idiot in a system failure can be detected by any other node, but it is more common to dedicate a single system component on the bus called a “bus arbiter” to detect and characterize node failures, and then coordinate the measures needed to minimize “system down time, ” or failure repetition, depending on the mode of the component failure. The bus arbiter can be thought of as a bus manager that tracks each component’s adherence to the system matrix, and drives the system to some failure-handling state (fail-safe, fail operational) either restoring correct system behavior, enabling system services at a reduced quality, or safely halting all system operations. Any time-triggered system can benefit from the inclusion of a bus arbiter, though the additional cost to the system, and development time required for verifiable and repeatable robust performance deserves a discretely pursued research focus. Inclusion of a bus arbiter in this work was skipped for this reason.

Chapter 5 Unmanned Agricultural Ground Vehicle (UAGV)

5.1 Introduction

The demonstration of a time-triggered communication bus did not require a complex vehicle as the system basis. The pervasive use of Ackermann-steering configurations on mobile agricultural machinery indicated the availability of many control models that could be used to demonstrate a basic navigation feedback controller; so an Ackermann steering solution was preferred. The platform selected was an accessible size; appropriate, as was suitable for prototyping system architectures at minimal safety risk to the testing environment and the users.

Objective:

Introduce the UAGV and describe the development of its components (from Section 1.1)

5.2 Materials and Methods

5.2.1 System Architecture

The platform was manufactured at the Agricultural Machinery and Research Laboratory on the University of Kentucky campus. Originally made for multi-robot cooperation demonstrations, the platform has been proven suitable for scaled-down demonstrations of FSM and Behavioral layers of multi-robot control architectures (Figure 5.1) in previous work (Pitla, 2012). The wheelbase of the chassis was 109 x 79 cm (Figure 5.2). The chassis was fitted with a 24V DC drive motor (Model NPC 41250, NPC Robotics, Mound, MN)) coupled to a differential on the rear axle – providing rear wheel drive with minimal tire skid. A 24V linear actuator (Model 85915, Motion Systems Corporation, NJ) was installed on the Ackermann steering pinions, enabling electronic control over the steering angle of the front wheels (Figure 5.3). Position feedback was available with the built-in linear potentiometer. Power was provided by two 12V rechargeable lead-acid automotive batteries in series for a system power bus of 24-28 VDC (Figure 5.4). Power was distributed first through a resettable 80A circuit breaker, and through a screw terminal array underneath the batteries.

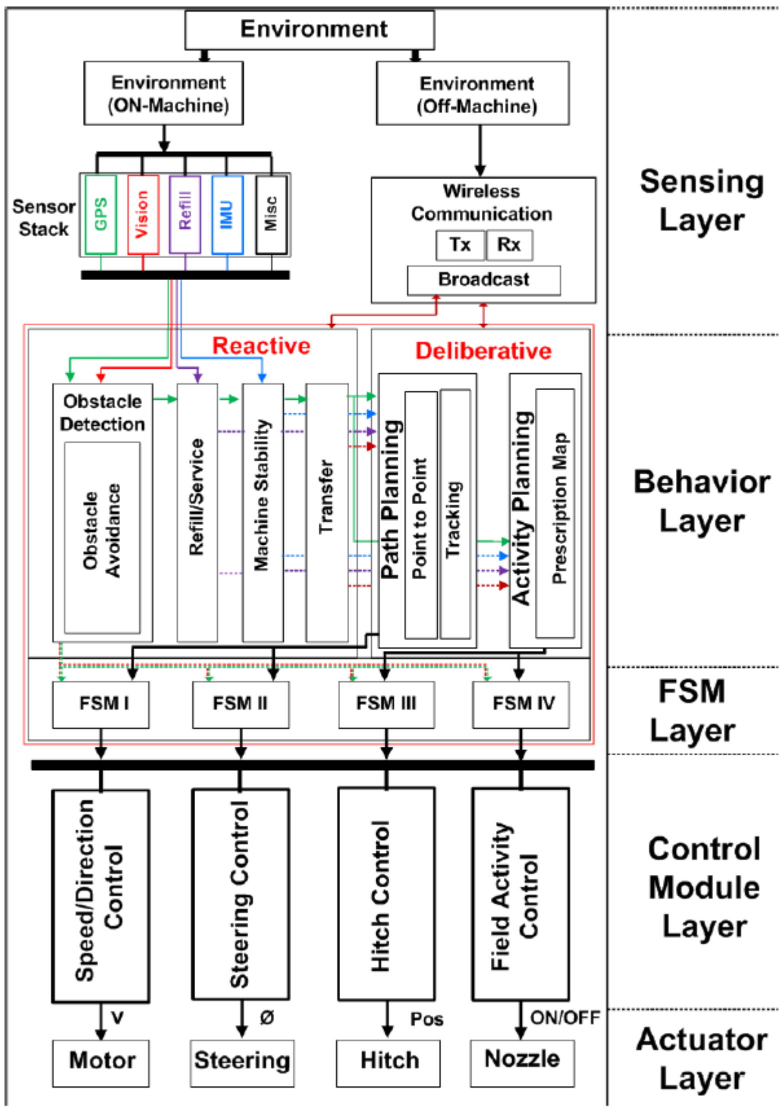


Figure 5.1 - Individual Robot Control Architecture (IRCA) (Pitla, 2012)

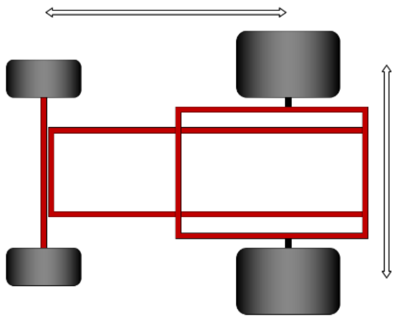


Figure 5.2 - Chassis layout

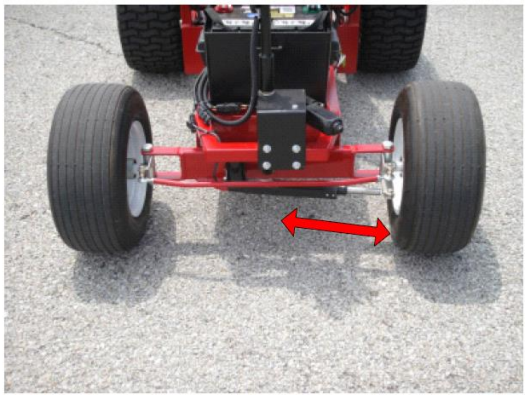


Figure 5.3 - Linear Actuation for Steering (Pitla, 2012)

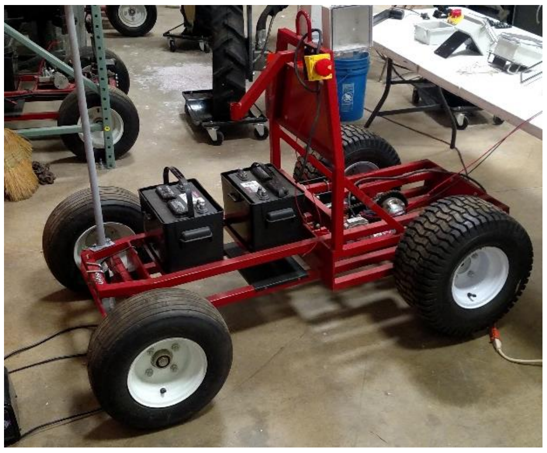


Figure 5.4 - Robot chassis including batteries, motors, and emergency-stop switches

From a control architecture perspective, it can be said that the entire system's foundation operates on the lowest hierarchy levels – the hardware, which is to acknowledge that the robot ultimately runs on wheels, propellers, motors, rudders, and blinking lights – are all mechanically or electrically coupled to a mobile chassis. This structure is very similar to the hierarchy found in the description of system abstractions (Figure 3.1). This works to the advantage of the systems engineer, since these architecture descriptions can be easily divided into the various computational resources required for a complete system implementation. That is, the computational tasks done at the Behavior level of the system, differ greatly enough from the tasks on the Sensing layer, such that the two layers can assumed to be performed on two distinct computational nodes as tasks. System design can then proceed with the assumption that each of the architecture layers is implemented on a separate computer; this provides the benefits of using a distributed system design. The distributed design of the system was implemented on a CAN bus because of the availability of CAN network development tools, the already ubiquitous use in agricultural systems. The arbitration of CAN ID fields to the bus as the means for bus access prevents TTCAN from being truly deterministic, since modes of CAN node failure can block critical messages from access to the bus (Ehrl and Auernhammer, 2007). A Flexray implementation would enable a very deterministic system, but demonstration of a simple time-triggered communication protocol can be done on a CAN bus.

5.2.2 Controller Area Network Node Hardware Environment

5.2.2.1 A Case for Custom Hardware

CAN systems have been around for over three decades now, and getting started with CAN node development has become far easier than it used to be, and the tools available for doing this range from drag-and-drop graphical programming interfaces, to hardware description languages (HDLs). Each solution applies some level of abstraction to the node developer - giving a range of control of system details to the system developer.

For very simple applications on blind-node or event-driven CAN protocols, a node based on the ubiquitous Arduino (Turin, Italy) boards could be used as a simple and inexpensive CAN system development platform. However, the Arduino's microcontroller has no CAN transceiver or CAN peripheral. For an added cost, a CAN interface board or "shield" can add all the required hardware to the microcontroller over a Serial Peripheral Interface (SPI) bus (Figure 5.5). While the included CAN controller is rich in features, the software libraries available do not make use of the entire feature set, and often hide system details that greatly impact network performance. For control applications on potentially dangerous systems, this solution is not recommended.

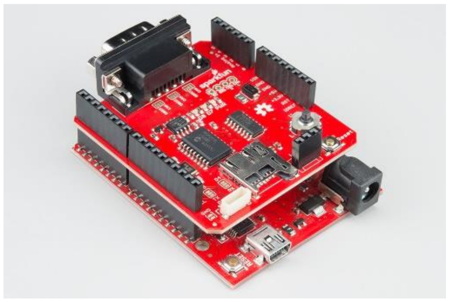


Figure 5.5 - Arduino with CAN controller interface

Single-board computer platforms however, have a much larger computational capacity for complex applications, and the well-supported CAN hardware interfaces and software libraries offer a high-performance solution to CAN system design and implementation. Devices similar to the Raspberry Pi and the Beaglebone run a supported Linux kernel and a rich set of digital interfaces available through a large number of programming languages. However, serious consideration of chapter Chapter 3 reveals a jump from these device's intended roles. Simply put, the Linux Kernel was not made for real-time applications. While a substantial amount of hardware might reassure the system developer about meeting system deadlines, the ability to make a deterministic system remains out of reach. The Linux operating system is complex, and the number of sub-processes it runs are likely to pre-empt user-level applications at any time. Additional considerations such as startup/reset time and file system corruption prevention further complicate the system design. The control module layer of the robot control architecture is supposed to be deeply embedded with the machine actuators, so using single-board computers at this layer seems inappropriate. Consideration of these devices in other control architecture layers however, demands different parameters.

Further searching for a more appropriate CAN hardware solution returns what the industry uses in commercial applications. Consider the industrial microcontrollers manufactured by Danfoss (Nordborg, Denmark, Figure 5.6): the devices are available in a wide range of available internal features, available GPIO pins, and operating voltages. The GPIO pins are heavily protected with surge protection and level shifting converters, and are shrouded in an automotive-compliant connector hood. The internals consist of one or two high-performance microcontrollers that run the application. While the hardware of these controllers is unquestionably most-suited for commercial products, the application development workflow is fundamentally constraining to the systems designer for two primary reasons:

1) The media access layer (MAC) protocol is either “Blind-Node” or event-triggered only. There are no software constructs for time-triggered system design in the workflow. This is inherently a major roadblock when researching time-triggered communication modes.

2) Limited support for external device interfacing greatly constrains the types of sensors and devices that the controller can use. Standard interfacing to PWM, PPM, analog, binary state switches, and encoders are simple to implement with the included development tools, but when considering access to an external device’s digital protocols over serial port, SPI, I^2C, LIN and parallel ports, access becomes significantly more difficult. This includes access to MEMS inertial sensors, digital thermocouple interfaces, GPS receivers, and other commonly-needed devices for robot navigation and control.

Figure 5.6 - Model: MC024-010 (Danfoss, Nordborg, Denmark)

A well-balanced solution to the CAN system hardware problem should include the flexibility to interface to any external analog or digital device with enough computational resources to ease software constraints in meeting real-time deadlines. Lastly, a free and open development toolchain that enables an easily documentable workflow, development methods, and simplified project collaboration. This balance can be implemented with a microcontroller with more computational power and more digital interfacing peripherals than is found on the Arduino, yet has a large amount of available resources for software development at a low cost.

In an attempt to satisfy these requirements, a re-examination of the robot control architecture is recommended. The entire robot system is divided into several hierarchal subsystem layers that can be individually developed and then integrated into a single system to produce a working robot. The control architecture (Figure 5.1) under further consideration was developed and more fully described in associated work (Pitla, 2012).

The actuator layer of the demonstrated system was introduced in Section 5.2.1, meaning the next steps of system development involve the control module layer, and its interface to the FSM and actuator layers. Consider the highlights to Figure 5.1 shown in Figure 5.7. The control layers are encompassed inside distinct “domains of computation, ” which divide all the control tasks between different types of computational resources that make up the robot control system. This is done to prevent a single, complex computer from holding responsibility for the entire system, but also allocates tasks to appropriately scaled processors and software frameworks. This is not to be confused with the standard distributed system architecture described in Section 3.3 – which corresponds to the implementation of the Control Module Layer only. For the remaining layers of the control architecture, a distributed processing mechanism serves the purpose of assigning specific architecture layers to appropriate computational resources. For example, this can be expressed as an acknowledgement that according to the given robot control architecture (Figure 5.1), the computational resources (processors, threads, software libraries) that run the Control Module Layer are not likely to be the same as those assigned to the tasks in the Behavioral Layer, and vice-versa.

In Figure 5.7, the solid blue outline denotes a set of tasks appropriate for a remote computational resource, an off-machine computer dedicated as a mission/task server, and communications bridge to other machines. The green dashed boxes encompass tasks that are well-suited to on-machine, embedded applications processors capable of advanced inferencing and mass data processing. These are likely single-board computers running advanced software specializing in machine vision, machine learning, complex FSMs, path planning, obstacle avoidance and system-level watchdogs. Finally, the red dotted boxes denote the tasks well-suited to simpler processors dedicated to deterministic sensor sampling and actuator feedback control, which streams lightly processed robot states to the upper layers on other computational resources. This method of task division is found on cyber-physical architectures meant for a wide variety of applications (Hehenberger et al., 2016). The following implementation of the Control Module Layer on the UAGV follows this pattern by assigning actuator feedback control to a family of CAN connected microcontrollers suited to providing a stream of robot states to upper layers of the control architecture.

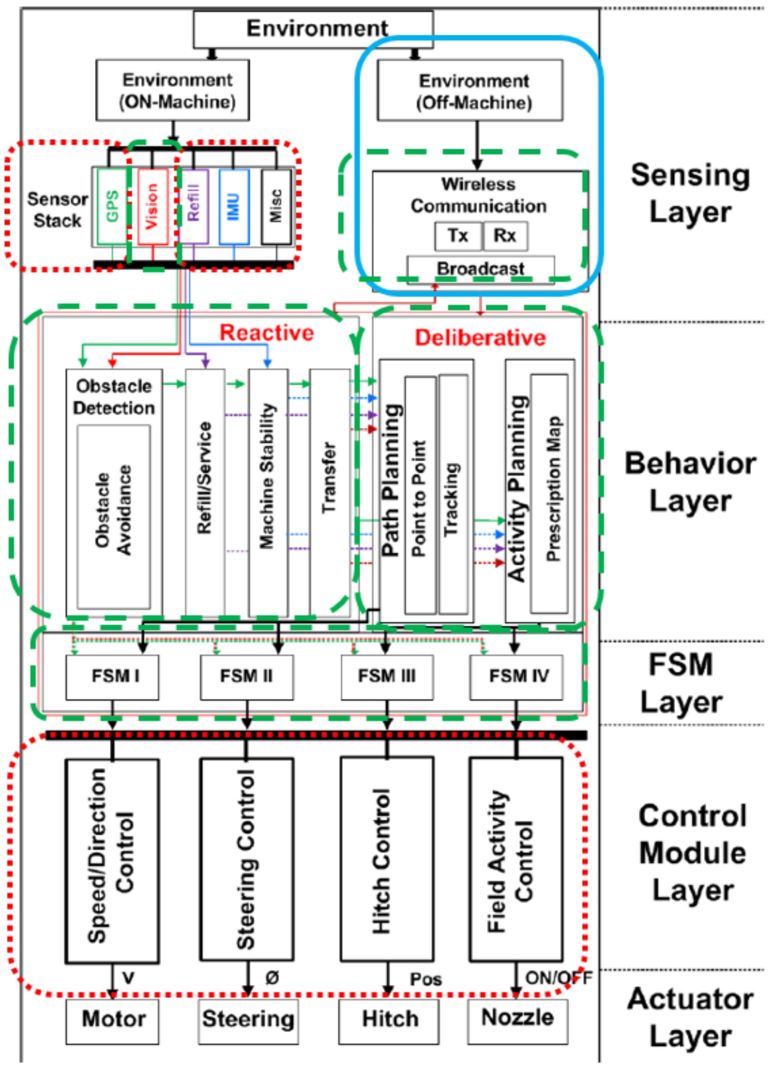


Figure 5.7 - Domains of computation across the control architecture

The Danfoss industrial controllers contain STM32F205 microcontrollers manufactured by ST Microelectronics (Geneva, Switzerland). This led to the selection of inexpensive STM32 development boards as the basis for a CAN node with the right combination of features. The “Nucleo” development board line is a very widely-ranging set of development boards available between $10 and $25 each. These boards are essentially the digital internals of the Danfoss industrial microcontroller “broken-out” on a more accessible board. Each board includes an Arduino-compatible pinout, and a complete debugger / flash downloader device that interfaces the target microcontroller to the development toolchain on a PC. As a complete development platform, the Nucleo boards are remarkably inexpensive.

Two development boards were selected for the robot CAN bus implementation. The F303k8 board was chosen for motor control nodes because of its small form factor, enabling it to fit into small enclosures where motor drivers were likely to be housed. The F3 family of the STM32 includes a 72 MHz, 32-bit ARM Cortex M4 processor, and includes a peripheral set for “mainstream” applications as described by ST Micro. The other development board selected was the F446RE for its extremely fast processor clock in a 64-pin package, and it’s inclusion of 2x memory-mapped CAN peripherals (Figure 5.8). The F446 has a 180 MHz, ARM Cortex M4 processor with a set of peripherals designated by ST Micro for “high-performance” applications.

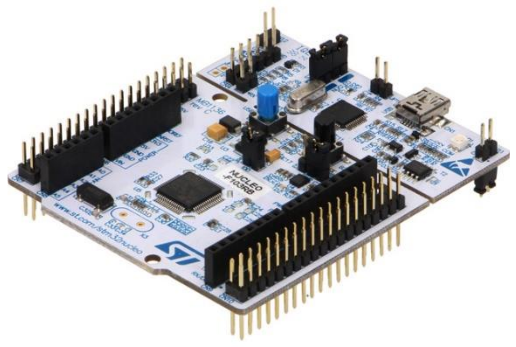


Figure 5.8 - ST Microelectronics microcontroller development board (F446RE)

This part focuses on the implementation of a control module layer since an actuator set was already in place on the robot chassis. It is common for a controls engineer to be commissioned for the design of a controller, given an already-existing control plant.

5.2.3 System Node Allocation

Identifying the needs of the system is important to the allocation of system resources. As will be shown in chapter 7, a typical kinematic steering model includes cross track error, heading, steering angles, and sometimes other states to provide a sufficient feedback signal for a steering controller. To collect this minimal information, sensors can be interfaced to the CAN bus with a microcontroller as the gateway. Depending on the type of state data needed, many sensors may be required to collect one system state. It is also possible to derive many states from a single sensor.

To act on a system, a control algorithm must have access to actuators that influence the system states either directly, or through some coupled relationship described by the system model. Dedicating a CAN node to each actuator is likely necessary to abstract actuator control signals to a simple CAN message definition. To satisfy the needs of the UAGV system with 3 states and 2 actuators, 4 CAN nodes were allocated:

1) GPS Node – Provides position data needed to derive 2 controller state signals (cross track error, and heading).
2) Drive Motor Node – receives and executes drive speed commands on the drive motor from the controller node.
3) Controller Node – receives sensor data, calculates systems states, generates control messages, and performs user interfacing to robot.

4) Steering Node – receives steering rate commands from controller node and provides steering angle sensor data.

This results in the CAN bus network as shown in Figure 5.9. A CAN logger device is shown for completion.

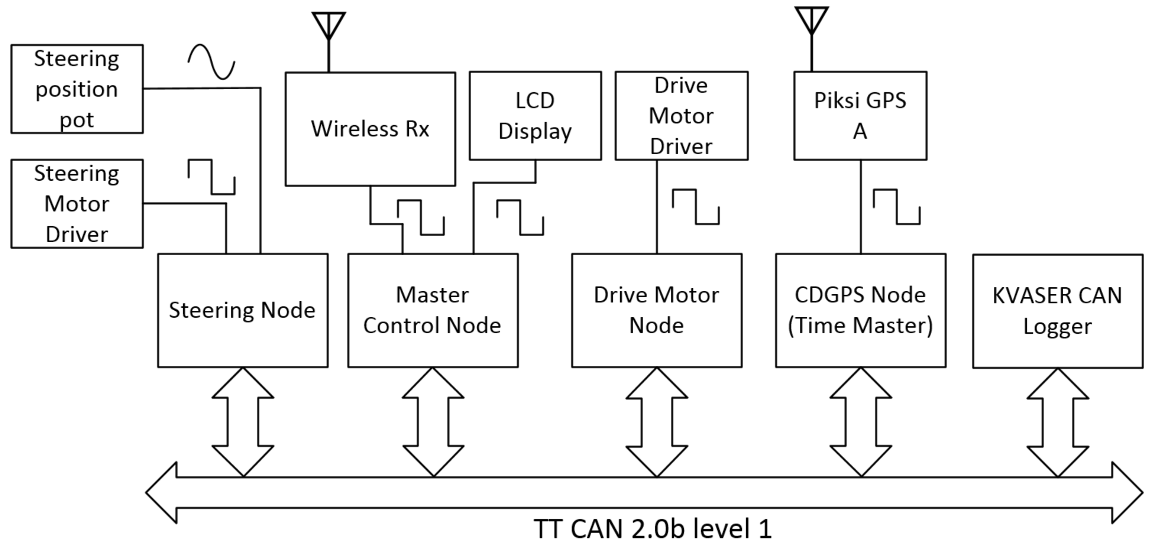


Figure 5.9 - UAGV CAN bus view

5.2.4 Design of Universal Inexpensive Node Hardware

5.2.4.1 Board Requirements

Exposing the benefits of the STM32 microcontroller required the development of an interfacing board to adapt the CAN bus and power sources of the system to the requirements of the microcontroller. The STM32 device required a 3.3V supply, which was provided with a 3.3V voltage regulator on the Nucleo board. To power the regulator along with the ST-Link debug device on the Nucleo board, a 5V power supply was required. The addition of an over-current safety mechanism was added as a requirement since the variety of different power buses and a common chassis ground increased the risk of short-circuit connections as the result of unintended debris in contact with the board.

The board required the ability to interface to two CAN buses simultaneously to accommodate both controller peripherals available on the STM32. This increased the board's usability in future systems with two system buses present. The ability to access the Nucleo board's original GPIO pin arrays was considered a secondary, though desirable feature.

5.2.4.2 Layout and Parts Selection

To enable universal use of the CAN node, a wide-input (6.5 – 32VDC) switching power supply module (CUI V7805-1000R, CUI Inc, Tualatin, OR) was included to provide the 5V bus of the design. A 1-amp ATO fuse mount was added to the interface board design to prevent excessive current draw, and enable fast fuse replacement. The SN65HVD250 CAN bus transceiver (Dallas, TX, USA) was selected to perform the bidirectional conversion between single-ended signaling of the microcontroller and differential logic of the CAN bus supported the decision to operate the CAN bus with 3.3V transceivers. These major board components are shown in Figure 5.10.

There were few, yet important circuit layout design considerations in the CAN sleds. Beyond supplying the device power requirements, shielding the CAN signaling traces and enabling easy connectivity informed the choices in the board layout. CAN signaling is differential, which encodes symbols into the difference between two voltages, rather than the absolute value of one voltage. Differential signaling relies on both signaling conductors to be tightly electromagnetically coupled. This is typically done by twisting the wire pairs together to ensure they are equally affected by noise. If the wires of the pair are unequally affected by noise, the benefits of differential signaling are reduced. The CAN specification calls for the use of twisted pair for this reason.

Preserving the effects of a twisted wire pair is not typically done on the printed circuit board, due to the planar arrangement of the board’s conducting mediums. To preserve the signal quality of the CAN bus as it passed through the board, the bus conductors were increased in width to minimize overall impedance and were surrounded by vias to shield the bus from noise (Figure 5.11).

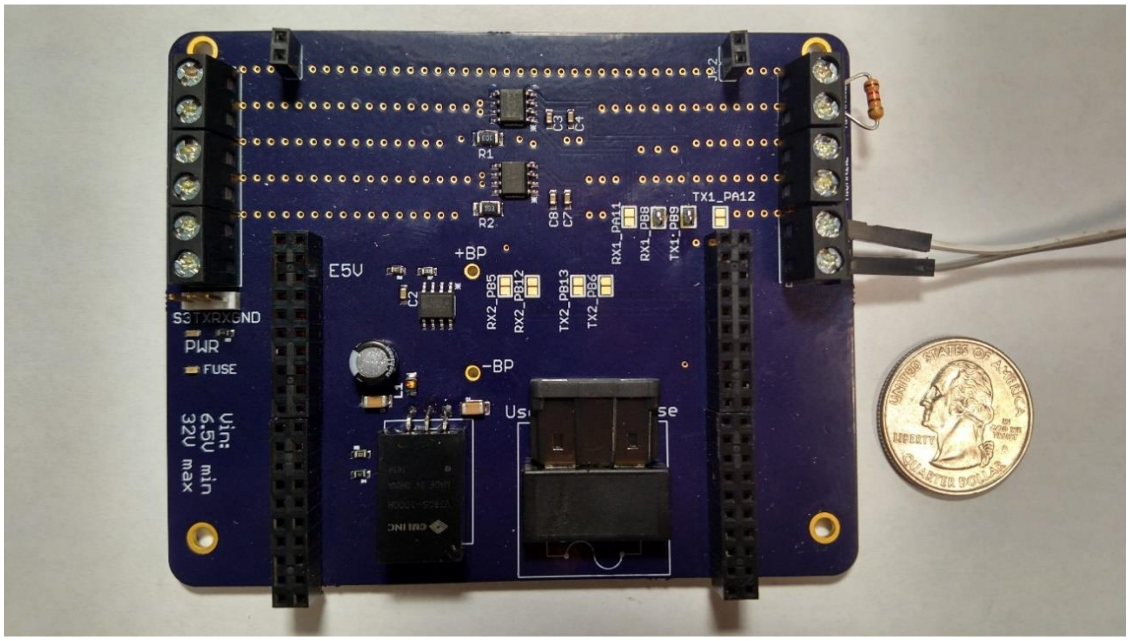


Figure 5.10 - CAN bus "sled" assembly

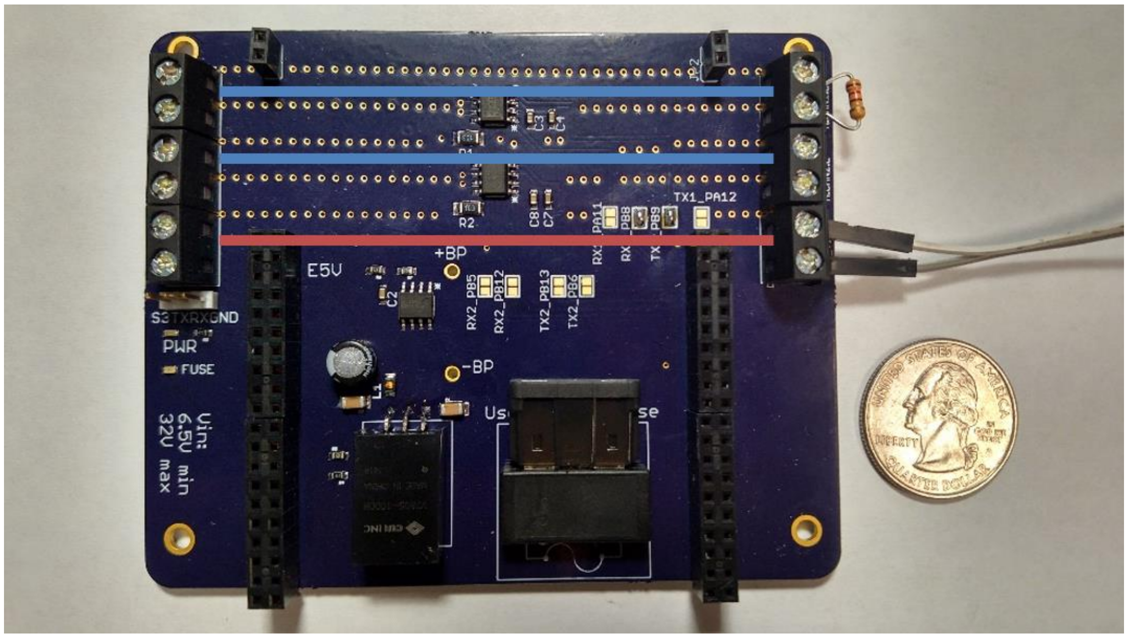


Figure 5.11 - Pass through traces for power and 2 x CAN buses

The development board mounted to the CAN sled by the GPIO pin arrays. This was an important feature in implementing a modular control module layer. The boards can be easily swapped without having to break CAN bus connections of an already-installed system. The bus connectors also functioned as a place to install bus-terminating resistors (Figure 5.12).

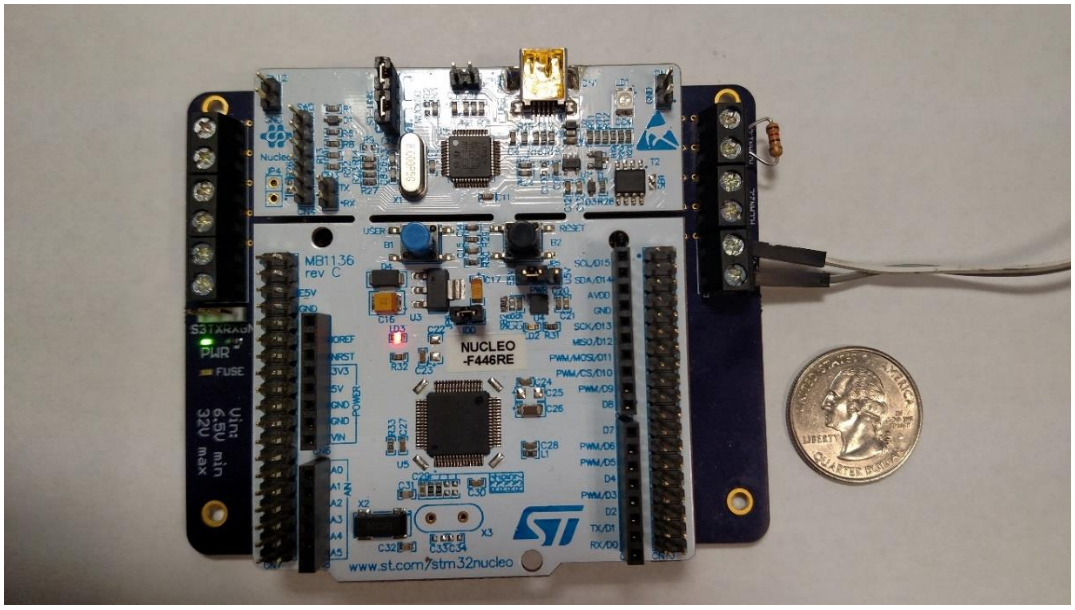


Figure 5.12 - Development board and debugger installed onto the CAN sled

5.2.4.3 Board Testing

The CAN sleds were tested by the application of power without any Nucleo development board installed. This tested the on-board power circuitry including the voltage regulator, main fuse, and “fuse-out” indicator lights.

5.2.5 User Interface and Finite State Machine (FSM) System Modes

The interface to the user was kept simple. The control module layer of the robot being presented is a layer of control not normally available to the user of the robot. Normally, an FSM layer or Behavioral layer (Figure 5.7) running on a separate computing system would be the “user” in control of the control module layer operations. The provision of GPS waypoints, start/stop commands, and behavioral inputs needed to be simulated to a degree that demonstrates how the robot’s control module layer can be easily interfaced. A very lightweight FSM layer was implemented in the controller node’s firmware, and inputs to the system for other modes of operation were provided with wireless buttons, a joystick, and an LCD display.

The wireless pin toggle module functioned very similarly to a commercial vehicle's key fob remote (Adafruit Industries, NYC, NY). By pressing a button on the remote, a corresponding pin of the receiver module could be toggled low/high (Figure 5.13). This enabled easy FSM layer control by a human. This setup required the human to act as the Behavioral layer of the control architecture (Figure 5.7), but was sufficient for control module layer demonstration.

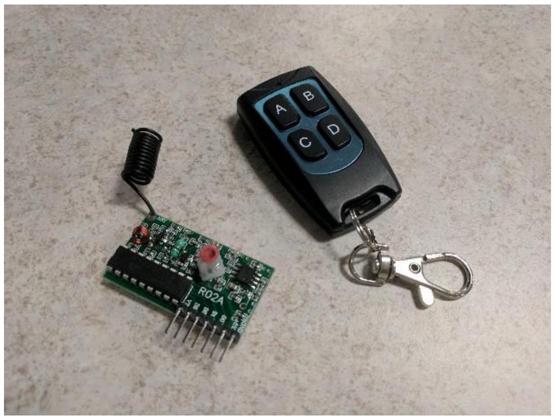


Figure 5.13 - 4 button remote controller for mode switching

An additional simulation of the behavioral layer was required for robot usability in control layer demonstrations. The use of a speed and steering input sensor enabled effective reactive/deliberative sets of user influence to the system. A joystick (Adafruit Industries, NYC, NY) with two analog outputs enabled basic speed/direction and steering control to the system (Figure 5.14). The joystick included a momentary switch "J", which was interpreted in software as equivalent to a "D" button as found on the remote controller. This allowed the user to steer the robot to and from testing sites, and to pose the robot for initial conditions of the experiments. The joystick was interfaced to the controller node.

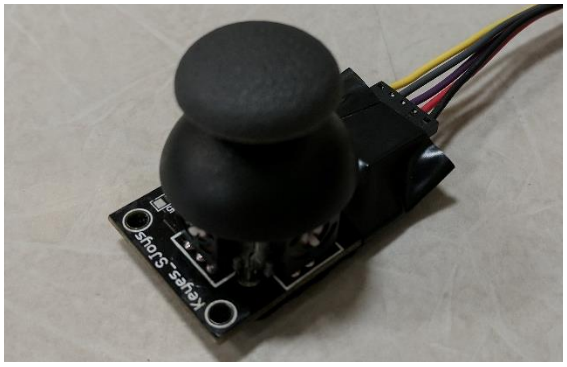


Figure 5.14 - User joystick for manual actuator control

For feedback to the FSM and behavior layers (the user), a thin-film transistor liquid crystal display (TFT LCD / LCD) was installed (Best Circuits Inc., LakeZurich, IL). See Figure 5.15. The LCD visually provided the user a large number of internal system states. Among the printed states were waypoint coordinates, northing-easting position, cross track error, heading, heading error, speed, steering angle, system time, operation mode, state of the CAN bus, and state of the CD-GPS fix. All these states represent information shared to the middle layers of an automated control architecture, which in the case of the UAGV was the human user.

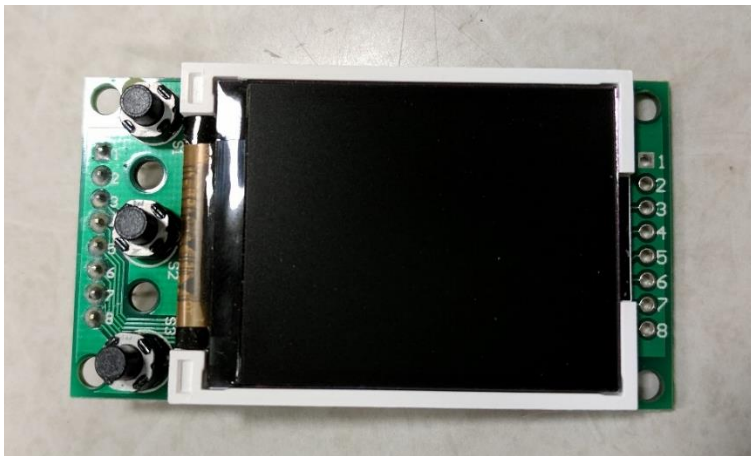


Figure 5.15 - Liquid crystal display for system state visualization

The operation modes of the robot were controlled by a finite-state machine that pointed the robot's full set of possible behaviors to a specific context. The FSM layer could be represented as a multiplexer switch that routes outputs of the behavioral layer to the inputs of the control module layer (Figure 5.16). It governed which control inputs possess the steering actuator and the drive motor at any given instant. This simplified FSM layer for the robot was controlled by the behavioral layer (the remote control) and routed (multiplexed) the control layer inputs (joystick and GPS position) to the robot's motors (Figure 5.17).

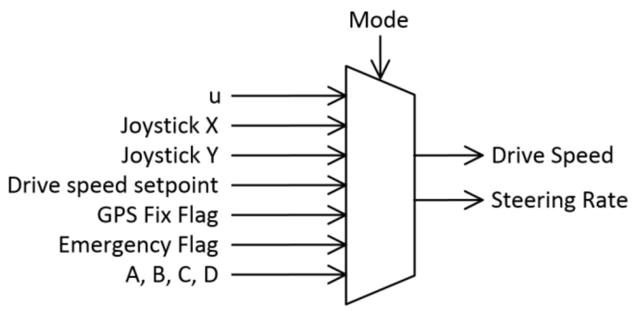


Figure 5.16 – Simple behavior layer as a multiplexer

State transitions were produced by remote control (Figure 5.13) button events and internal system states. This allowed a user to remotely control the behavior of the robot while avoiding bad undesirable behavior due to the environment. For example, the FSM locks out the “Autonomous Mode Run” state from being reached if the GPS receiver has not yet found a positioning lock. It also automatically leaves the “Autonomous Mode Run” state if the GPS lock is suddenly lost during runtime. Emergency button presses were also handled in each state to allow the robot to reach a “safe state” from any other mode.

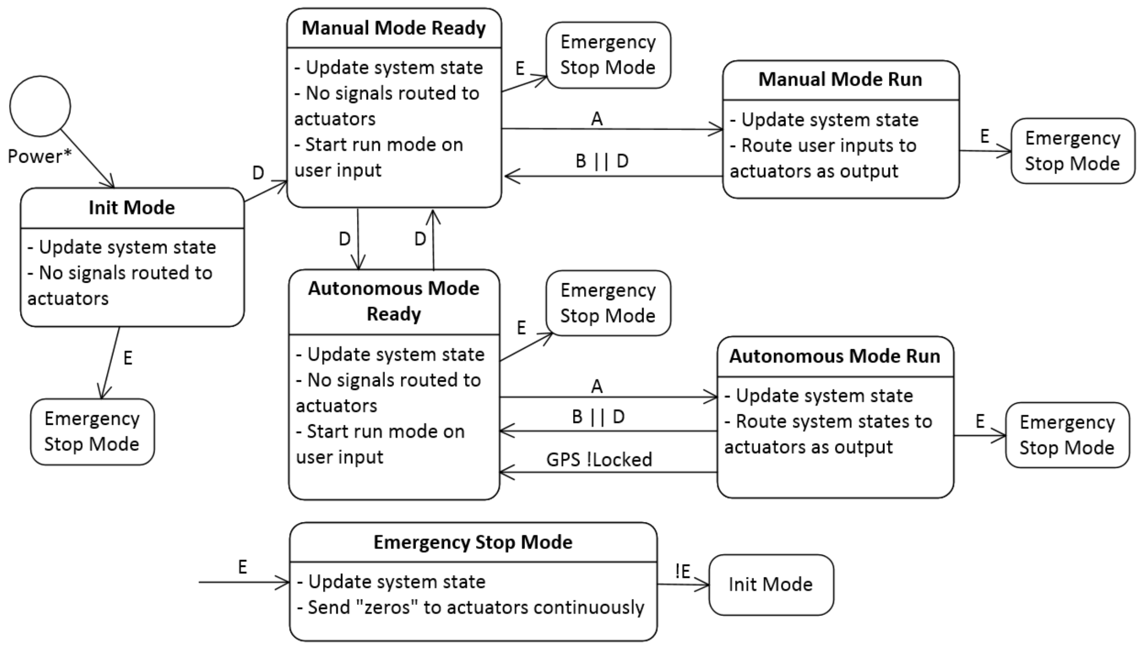


Figure 5.17 - Finite state machine for operation modes

5.2.6 Steering Actuator Calibration and Modeling

Mapping the voltage from the steering position potentiometer (built into the actuator) to steering angle of the front wheels required the collection of CD-GPS position data while the robot was manually driven in circles of varying diameters. This was done with a Memorator CAN data logger (Kvaser Inc, Mission Viejo, California). The points of the circles were measured from the robot position plot, and correlated to the ADC samples measured by the steering CAN node, and transmitted to the bus. A linear relationship was used for the calibration, and allowed access to the steering angle "δ" state (Figure 5.18).

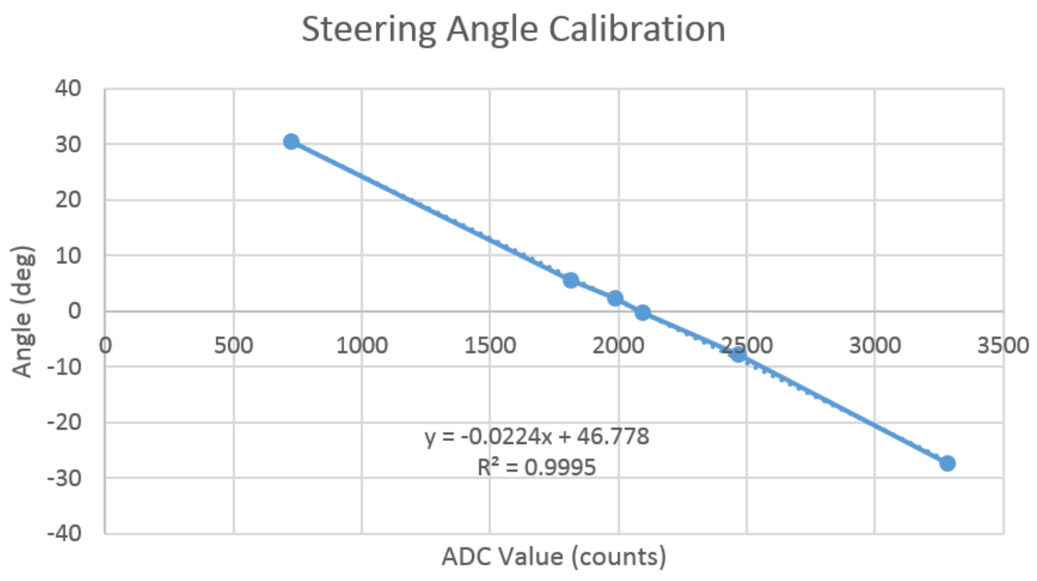


Figure 5.18 - Relationship between Steering Angle and 12 bit ADC Output

To produce meaningful commands for the steering actuator in units of degrees per second, the steering angle was collected to relate its derivative to the PWM commands routed from the joystick. Figure 5.19 shows the steering angle as some scaled integral of the command value.

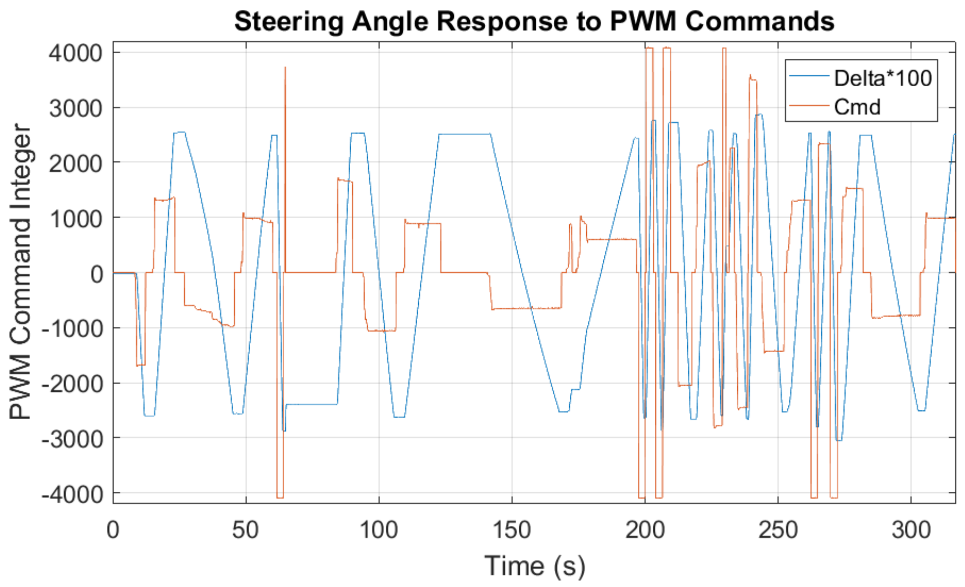


Figure 5.19 - Steering speed was linearly coupled to the PWM commands

Collecting the slopes of the delta series in Figure 5.19, and plotting them as the independent variable produces the linear relationship with dead-band shown in Figure 5.20. Commanding the steering actuator was then implemented with the use of two resulting first-order calibrations, each selected during runtime for the sign of the speed to be commanded.

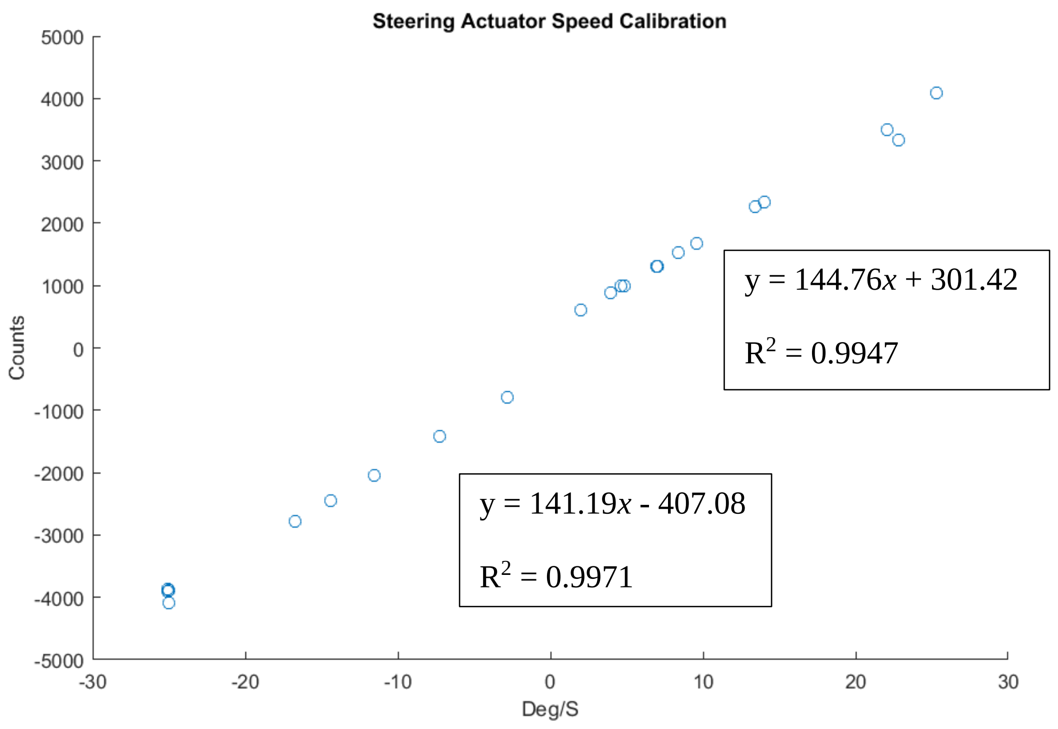


Figure 5.20 - PWM command vs desired steering rate

5.3 Results and Conclusions

The completed UAGV (Figure 5.21) was capable of a 2 m/S driving velocity, ±25 degree steering angles, and ±25 degree/S steering rates. The platform operating modes could be controlled from the wireless remote control, and steered/driven with the joystick. The CAN bus relayed all system variables, set-points, commands, and states at 10 Hz, making the system responsive to user input and emergency-stop button events. Manual operation of the vehicle did not require a GPS connection of any kind, but still used the GPS CAN node as the CAN bus time reference.

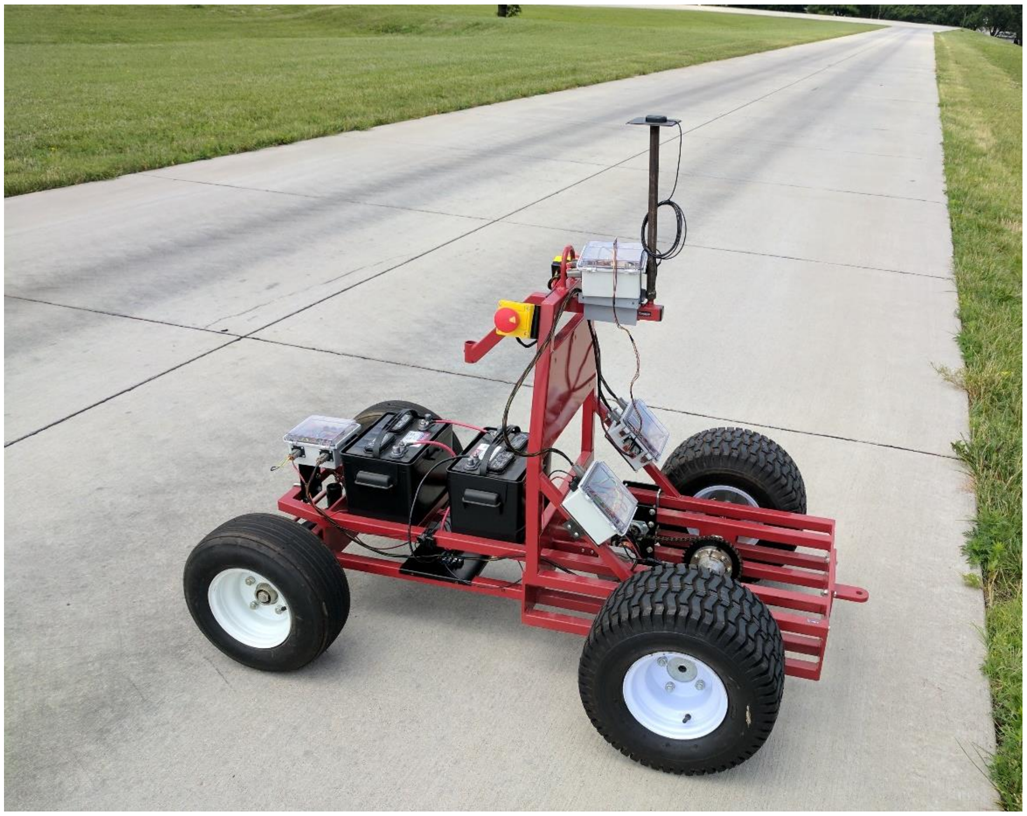


Figure 5.21 - Completed UAGV on the Nebraska Tractor Test Track

Chapter 6 Control Module Firmware Layer

6.1 Introduction

In Section 4.3.6, the basic period and system matrix of a time-triggered communication schedule was defined and presented with basic examples. In this section, the implementation of these design principles on the UAGV is presented to give a complete sense of scope required to benefit from deterministic communications and to document the technical details implemented for future researchers.

Objective:

Develop and demonstrate the hardware and software of the TTCAN communications protocol (Section 1.1)

1) Outline the system messages (6.2.1)
2) Present the basic period and system matrix of the TTCAN protocol (6.2.2)
3) Describe the CAN peripheral on the STM32, and derive it's configuration for the application (6.2.3)
4) Show the CAN peripheral's interfaces to memory and other microcontroller peripherals
5) Present the software architecture and it's memory interface to input and output data (6.2.4)
6) Describe the design of firmware drivers for configuration and control of key peripherals and sensors

6.2 Materials and Methods

6.2.1 System Message Definitions and Data Encoding

The design of a system matrix requires a definition of the system messages and their frequencies. In this application, all system messages shared the same 10 Hz frequency, which simplified the system matrix which is shown later. First, each system message and their contents along with the data packing operations are outlined (Figure 6.1 - Figure 6.12). The byte order is described by labels that defined the least significant byte (LSB) and the most significant byte (MSB) of a multi-byte digital vector. The byte ordering for each vector was arranged as a matter of convenience and could easily be modified in software to run in a reversed direction.

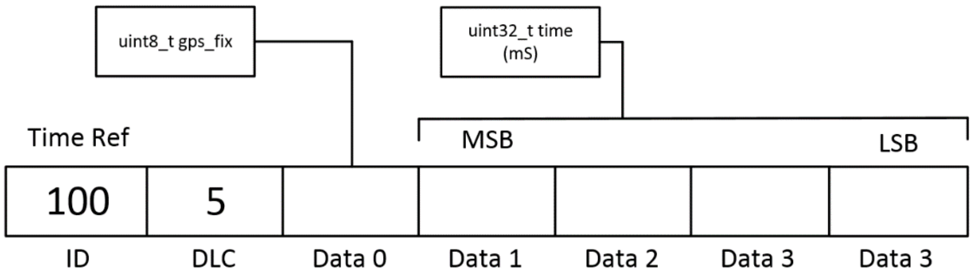


Figure 6.1 - GPS fix flag and system time reference message

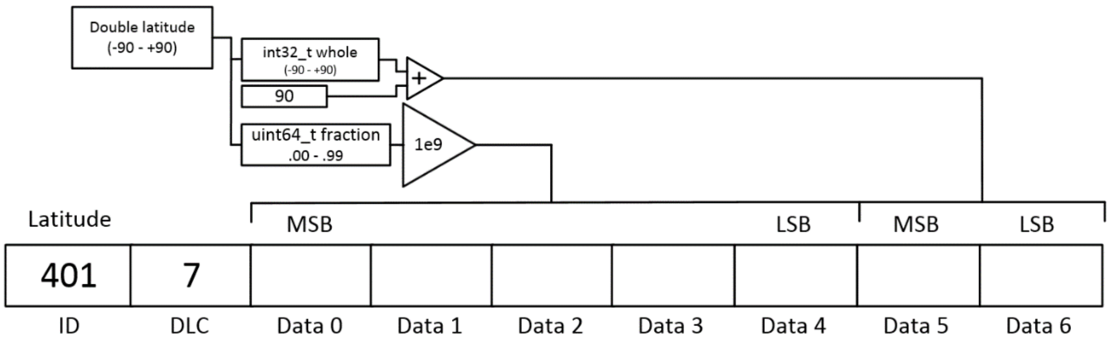

Double latitude (-90 - +90)
int32_t whole (-90 - +90)
90
+
uint64_t fraction .00 - .99
1e9
Latitude
MSB
LSB
MSB
LSB
401
7
ID
DLC
Data 0
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6

Figure 6.2 - GPS latitude with available 10 decimal places of precision

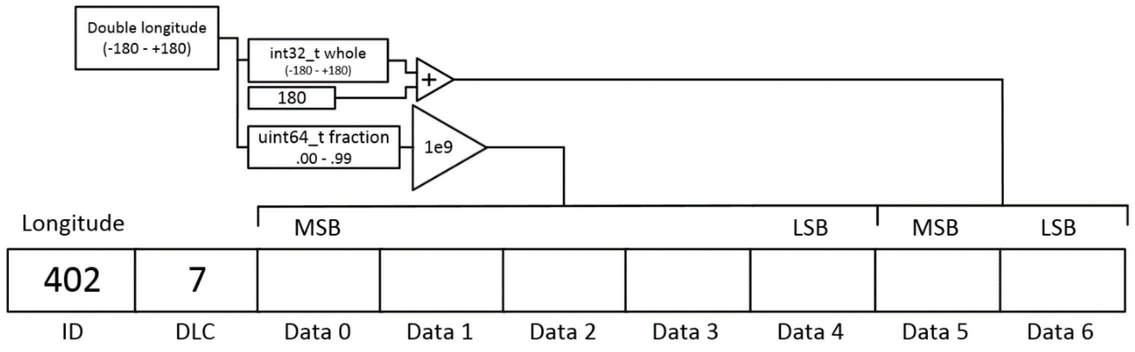


Figure 6.3 - GPS longitude with available 10 decimal places of precision

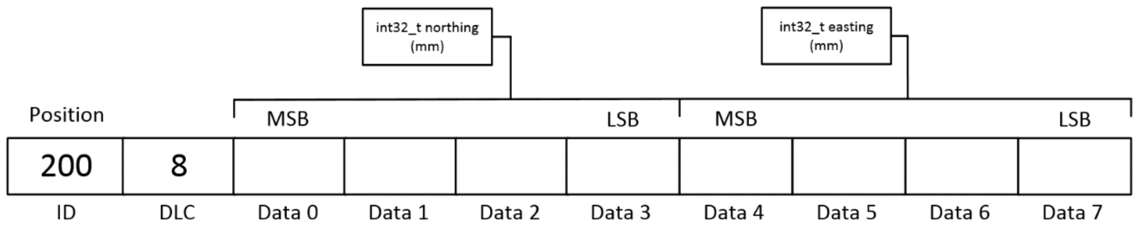


Figure 6.4 – Northing-easting position message

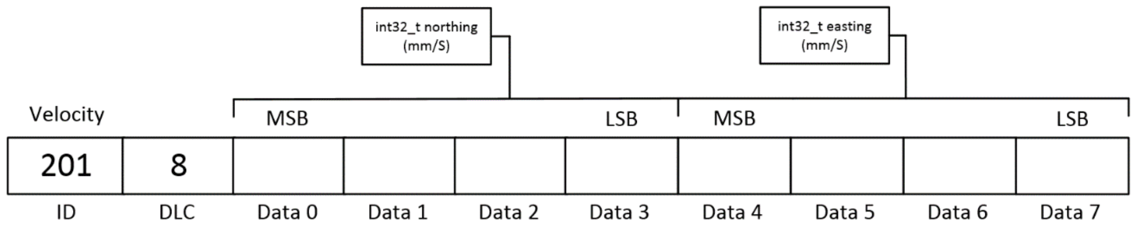


Figure 6.5 - Northing-easting velocity vector message

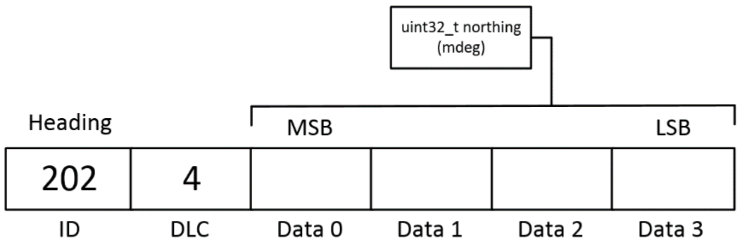


Figure 6.6 - Heading angle in milli-degrees

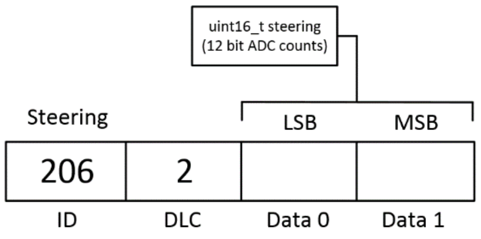


Figure 6.7 - ADC samples from the steering actuator potentiometer message

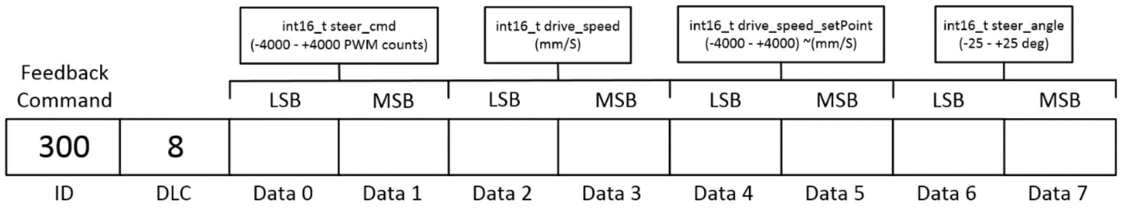

int16_t steer_cmd (-4000 - +4000 PWM counts)
int16_t drive_speed (mm/S)
int16_t drive_speed_setPoint (-4000 - +4000) ~(mm/S)
int16_t steer_angle (-25 - +25 deg)
Feedback Command
LSB
MSB
LSB
MSB
LSB
MSB
LSB
MSB
300
8
ID
DLC
Data 0
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Data 7

Figure 6.8 - Actuator commands in feedback message

The firmware of the microcontrollers directly interpreted data marked as “signed integer” types (int_xx_t) as two’s compliment for negative integer representation, so transferring signed data on messages between microcontrollers was a very efficient bit-wise copy operation (Messages 200, 201, 300). Some internal system variables were shifted into in an unsigned (positive) number range in message packing for simplified interpretation of the data logging messages by MATLAB, which unpacked the same system variables with the reverse of the procedures shown for messages 401, 402, and data messages 0-3. Four logging messages were produced and transmitted by the Control Node", which were logged and used for system data collection.

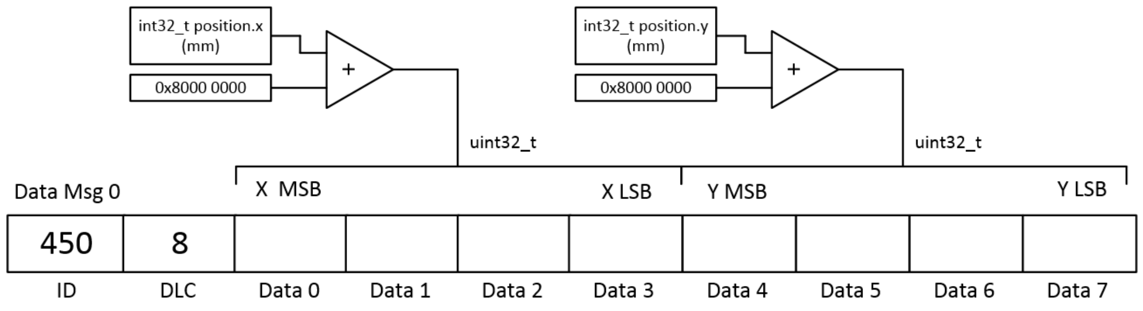


Figure 6.9 - Robot x and y coordinates in the D0 message

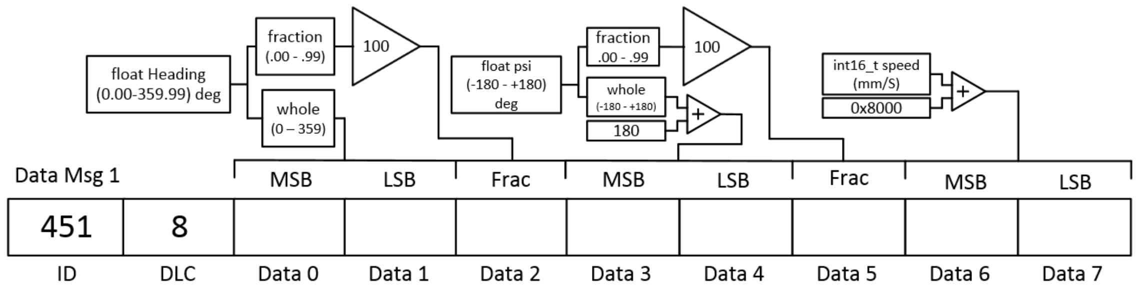


Figure 6.10 - Heading, PSI, and speed in the D1 message.

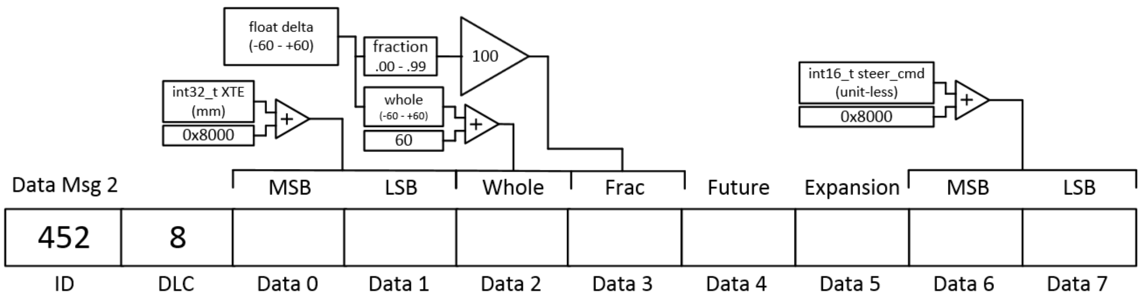


Figure 6.11 - Cross track error, steering angle, and command in the D2 message.

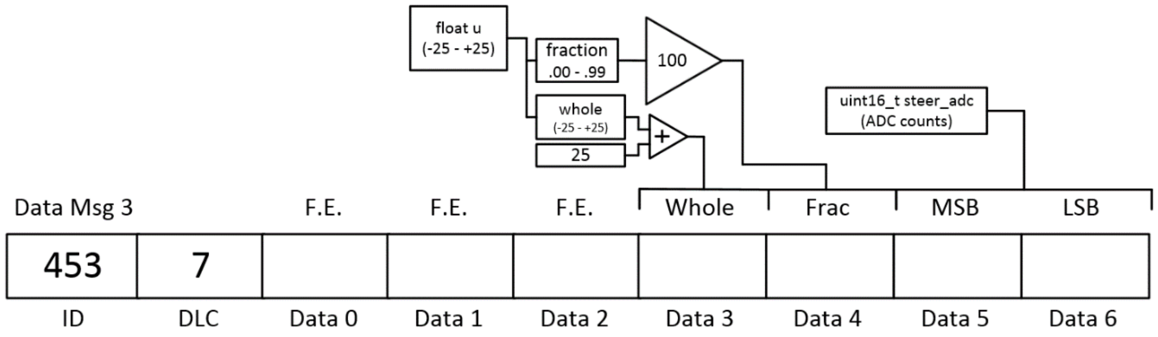


Figure 6.12 - Controller input signal U and steering pot ADC count packed into D3.

6.2.2 Basic Period and System Matrix

A [1 x 100] system matrix to handle all system messages was defined. Considering the worst-case CAN message transmission time of 520 uS (8 byte CAN data frame, 11 bit ID, maximum stuff bits, 250 kbps), a delay margin was included by defining the transmission window to be roughly double this length of system time (1 mS), which was a conservative margin because of a low number of system messages. This enabled a CAN node to have a maximum of 480 uS to respond to the beginning of a message window for the longest possible message on the system.

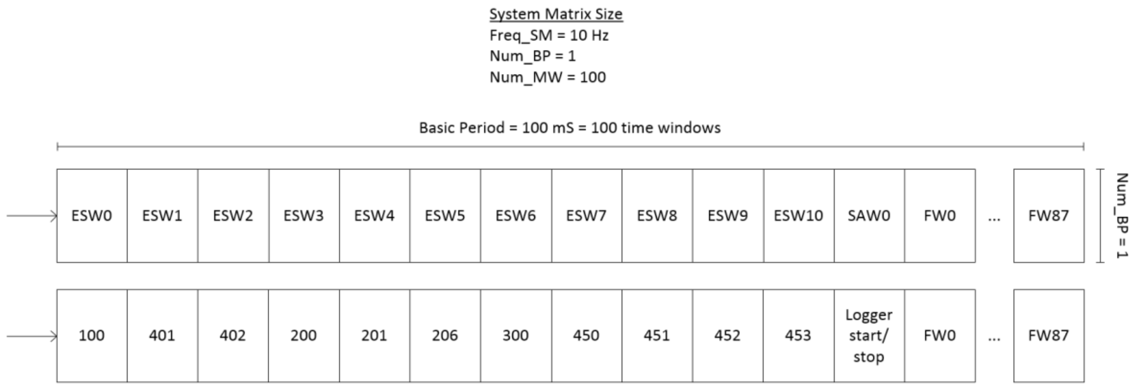


Figure 6.13 - UAGV time-triggered system matrix design

Each transmission window is 1 mS wide. Since each system message frequency was equal to Freq_SM (10 Hz), Num_BP was equal to 1. The end result was a greatly simplified system matrix design. This simplification was reached by matching all state message frequencies to 10 Hz.

6.2.3 Controller Area Network System overview

To enable an accessible and flexible CAN bus interface to the software developer, a C library of CAN related functions was developed. The library was written to accommodate the CAN bx peripheral found in the STM32 microcontroller family, enabling the development of time-triggered CAN nodes using any of ST's economy, access, low power, or high performance lines of microcontrollers. The library handled the simplification of the CAN bx controller configuration, automatic buffer management for transmission and reception message buffers, transmission window tracking, and system time correction.

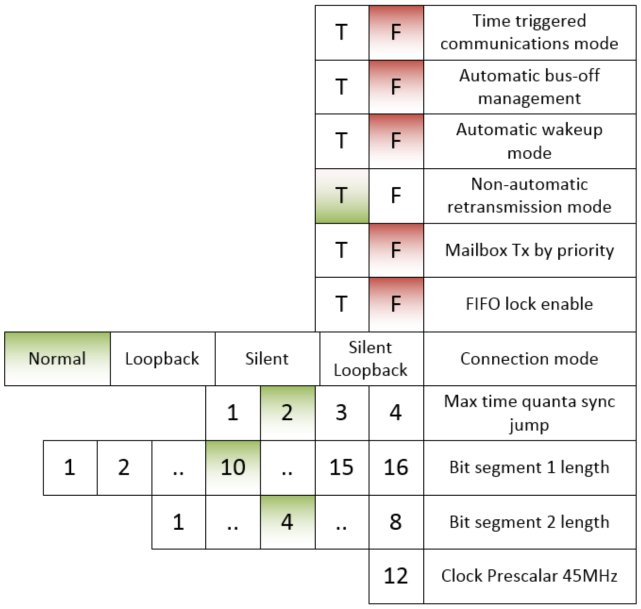
The library was written on top of the Standard Peripheral Library provided by ST Microelectronics, which is a layer that abstracts away the system peripheral accesses to well-documented C functions. This was the only major dependency of the developed TTCAN stack.

6.2.3.1 Controller Area Network Peripheral configuration

The CAN bx peripheral of the STM32 microcontroller line provides a large number of options for optimal communications configurations for many applications. The STM32F4 part used has errata material describing that the *time triggered communications mode* is not implemented. This mode enabled a dedicated 16 bit timer that stores timestamps on each SoF bit that arrives through the ID filters. This was meant to enable very high accuracy tracking of system time drift, since the reception of the SoF bit is a more accurate instant of time to capture, rather than the interrupt processing method used in the end application. The relatively large messaging window (1 mS) used in the final system prevented the need for such accurate time keeping requirements on each of the nodes. A peripheral time managed in the CAN message interrupt handler proved to be sufficient for system time tracking.

Additional important settings for the implementation of time-triggered communications included activating *non-automatic retransmission mode*, which prevents the CAN controller from violating the system matrix schedule should its first attempt to write to the bus fail. Leaving this mode off would cause catastrophic system communication schedule failure. Disabling *Tx FIFO by priority* was also done to prevent the same issue. To keep a transmission schedule, the CAN bx module must not be allowed to re-arrange the order of transmission given by the application. *FIFO Lock Enable* was disabled to allow the application to process the newest buffered CAN messages in the receive FIFO, instead of the oldest messages.

The remaining settings used the application included turning off the *automatic bus-off management*, which is capable of physically removing the controller from the bus should the count of errors it causes on the bus exceeds some threshold. *Automatic wake-up mode* was not used since the processor was never placed into a low-power state. Figure 6.14 contains the summary of CAN bx peripheral settings that can be configured by the TTCAN C library.



					T	[✓ F]	Time triggered communications mode
					T	[✓ F]	Automatic bus-off management
					T	[✓ F]	Automatic wakeup mode
					[✓ T]	F	Non-automatic retransmission mode
					T	[✓ F]	Mailbox Tx by priority
					T	[✓ F]	FIFO lock enable
			[✓ Normal]	Loopback	Silent	Silent Loopback	Connection mode
			1	[✓ 2]	3	4	Max time quanta sync jump
1	2	..	[✓ 10]	..	15	16	Bit segment 1 length
		1	..	[✓ 4]	..	8	Bit segment 2 length
						12	Clock Prescalar 45MHz

Figure 6.14 - CAN module settings

Setting the baud rate of the CAN peripheral was a process that required knowledge of the CAN bit segments defined in ISO11898, and the clock speed of the associated CAN peripheral. The clock tree as depicted in Figure 6.15 shows the system clock settings used in the CAN node application. The Advanced Peripheral Bus 1 (APB1) is shown to be clocked at 45 MHz, which was the source clock for the microcontrollers CAN peripheral circuitry. This was the “seed” of the CAN bus baud rate derivation.

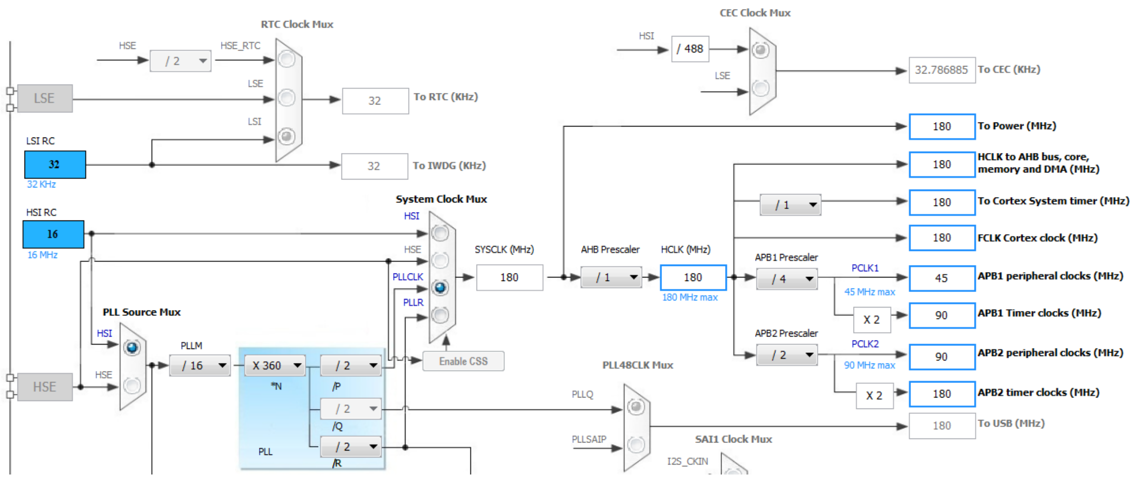


Figure 6.15 - STM32F446RE system clock tree (ST Micro, Geneva, Switzerland)

The algorithm for setting the CAN timing register (CAN BTR) of the peripheral required the speed of the APB1 bus and the desired baud rate of the CAN bus as inputs. The segments of the CAN bits were then derived using the constraints of the CAN bx peripheral.

First, a bit frequency (BF) was derived from the ratio of the input clock to desired baud rate.

$$BF = \frac{APB1\_Freq}{Set\_BaudRate}$$

Next, the BF was divided down with a prescalar that clocks the CAN peripheral to derive the number of time-quantas (TQ) that sum to a single CAN bit of time. A single CAN bit period is the sum of the 3 ISO11898 segments, tq, BS1, and BS2. The rest of the configuration process was mapping these segments into the TQ time period. The CAN bx peripheral can satisfy this allocation within the constraints on the following definition:

$$TQ = \frac{BF}{Prescaler} = (1 + BS1 + BS2)$$

$$such\ that\ TQ\ is\ some\ whole\ integer, and\ 8 < TQ < 25$$

TQ can be divided into the three time periods defined by ISO11898, where tq is always one, and BS1, BS2 set the sampling point of the bit. A desired bit sampling point is informed by the baud rate, bus voltage, and bus length to allow for bit propagation delays. In this application, the CAN bus length was very short (< 10m), and the baud rate was set to 250 kbps (25% of max baud), so a sampling point between 70 and 80% was chosen for all system nodes. This was achieved by choosing bit segments BS1, BS2 such that

$$BS1 \cong 0.75 * (TQ - 1)$$

since

$$0.70 - 0.80 \cong \frac{BS1}{BS1 + BS2}$$

$$where\ BS1\ and\ BS2\ are\ whole\ integers$$

Finally, the second half of the bit time segment can be derived by

$$BS2 = (TQ - 1) - BS1.$$

For the final application, this process used the following calculations:

$$BF = \frac{45\ Mhz}{250\ kbps} = 180,$$

$$TQ = \frac{BF}{Prescaler} = \frac{180}{12} = 15\ time\ quanta\ per\ CAN\ bit$$

$$BS1 = 0.75 * 14 = 10.5 \cong 10\ time\ quanta\ before\ bit\ sampling\ point$$

$$BS2 = (15 - 1) - 10 = 4\ time\ quanta\ after\ bit\ sampling\ point$$

This produced a bit sampling point at

$$\frac{BS1}{BS1 + BS2} = \frac{10}{14} = 71.4\%$$

The contents of the CAN BTR register contained the values:

Prescalar = 12

BS1 = 10

BS2 = 4

These values were fixed to the CAN bx peripheral using the Standard Peripheral Library functions. The result produced a bus baud rate of 250 kbps when the CAN bx peripheral was clocked on APB1 bus at 45 MHz.

The entire CAN bx peripheral of the STM32 microcontroller handled the bit-wise message arbitration to the bus, and the message filtering and buffering from the bus. It can do so independently for two CAN buses in simultaneous connection to the device (Figure 6.16). Each of the two controllers in the CAN bx module had two pin connection options for each transmit and receive pin. Making the final connection to the CAN transceiver from the microcontroller pins required shorting a solder jumper on the CAN sled that matched the user's choices for pins. This enabled more pinout flexibility in the final design of the CAN node.

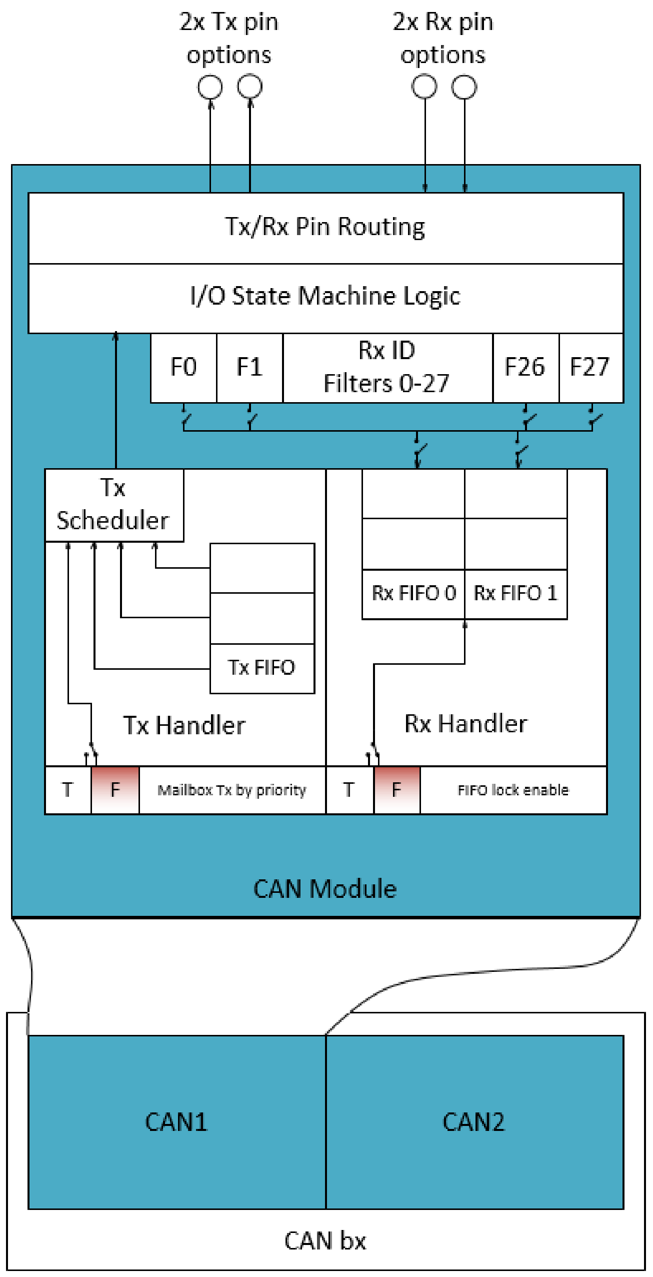


Figure 6.16 - The CAN bx peripheral

6.2.3.2 Interrupts and Message Buffering

The CAN bx module has the ability to generate a hardware signal to the processor called an interrupt. Interrupts typically occur in response to events caused by systems external to the processor to signal the need for an event to be handled. By activating this interrupt signal, the processor forcibly jumps from its current section of program memory to run a short section of code before returning to its original task. These short sections of code are called interrupt handlers. Interrupts are needed to describe the functionality of the STM32 CAN library.

In short, the interrupts handlers of the CAN stack do all the TTCAN related tasks of timely message reception, and timed transmission. An interrupt that runs in response to a new message in the CAN bx Rx FIFO copies the message to a message ring buffer in the application memory for later processing by the application. The same interrupt updates the system time when a time reference message arrives and also starts a timer that tracks the application's current time-window position in the system matrix. When the timer expires, a timer interrupt handler runs to transmit a CAN message from the Tx Ring Buffer. If there is more than one message in the Tx Ring Buffer to transmit, the timer interrupt can reset the timer to the length of the time window (1 mS), allowing the next CAN message to be transmitted by the same interrupt handler when it runs 1 mS later – in the next time-window of the basic period. Figure 6.17 shows the interrupt signals and how they enable the processor to time access to messages that need to be transmitted or received.

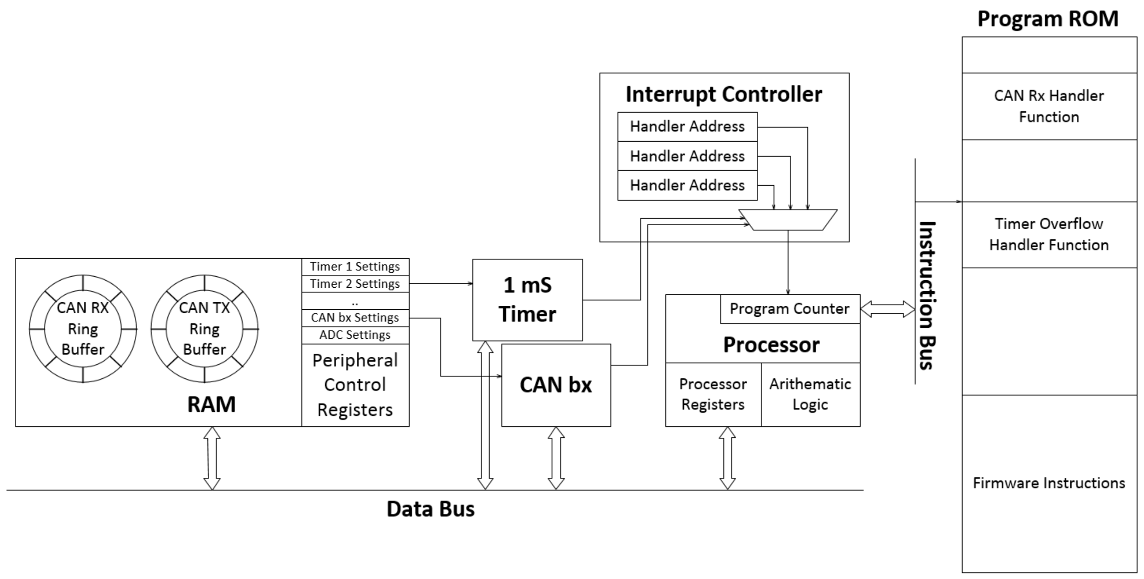


Figure 6.17 - A representation interrupt events produced by peripherals

6.2.4 Peripheral Firmware Components

6.2.4.1 Controller Node Firmware Structure

The peripheral configurations in the Controller CAN node enabled a particularly simple software architecture for easy modification and simplified performance verification. Most of the digital peripherals were configured for fully autonomous operation, where they only required attention from the CPU during start, stop, and configuration steps of the peripheral setup. This meant that once the peripheral was configured and started, the results of the peripheral were placed in memory automatically without the help of the processor. The advantages of this configuration showed the usefulness of the microcontroller’s ADC, which autonomously placed signal samples of all system buttons, switches, and joystick pins into a single memory array – keeping the samples up-to-date at all times. Additionally, non-critical functions of the serial communications peripheral were able to send large chunks of data to the LCD without requiring constant time and attention from the processor – letting the more time-critical tasks have the processor more often. The key to this autonomous peripheral configuration was use of the direct memory access (DMA) peripheral. This peripheral enabled semi-concurrent streams of data to move from system input/output peripherals to specified locations in memory – statically allocated as variables and arrays of variables in the C language.

The peripherals that ran only semi-autonomously were central to the CAN node application, and required use of the processor only during critical instances of time. The CAN bx and two timer peripherals required the use of interrupts to operate, though only one of the timers was needed for the time-triggered schedule tracking and system time keeping, while the other timer interrupted the processor at only 5 Hz to periodically toggle a global system flag. The resulting application was a "round robin with interrupts" architecture (Figure 6.18 and Figure 6.19). This means there was no centralized task allocation, scheduling, or operating system to manage the application.

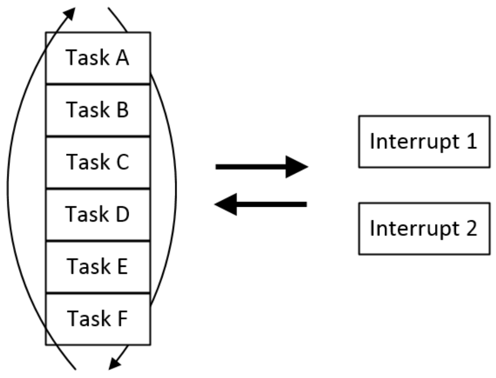


Figure 6.18 – Generalized model of Round robin with interrupts task execution

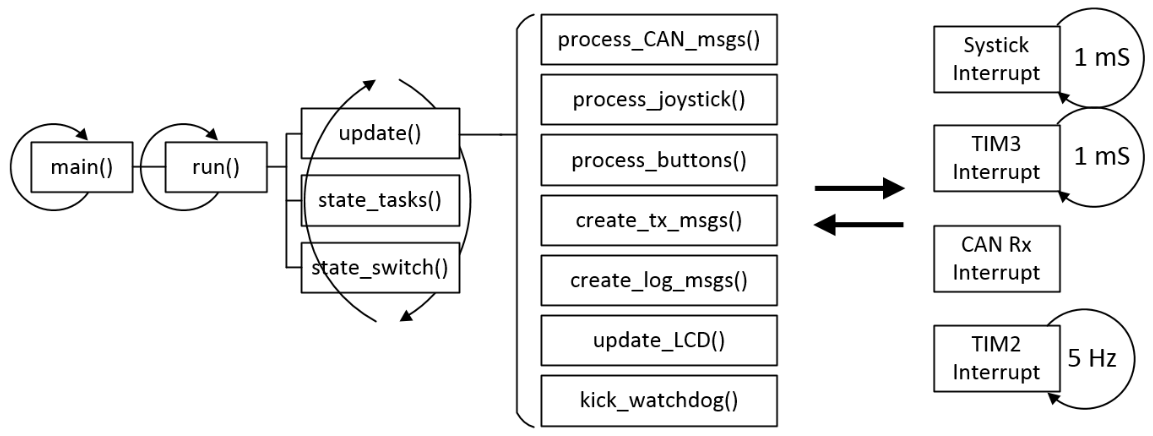


Figure 6.19 - Model of finalized application tasks and interrupts

The processor only had to read the memory objects provided by the autonomous peripherals to check for system state changes, data processing tasks on incoming data. Infrequently the processor handled an interrupt by copying data between memory buffers. The resulting number of “blocking” tasks was zero, meaning that no single process was able to cause the processor to stay in some single subroutine for too long (Figure 6.20). Application task execution order was preserved from Figure 6.19 to show the order in which memory objects were accessed and modified.

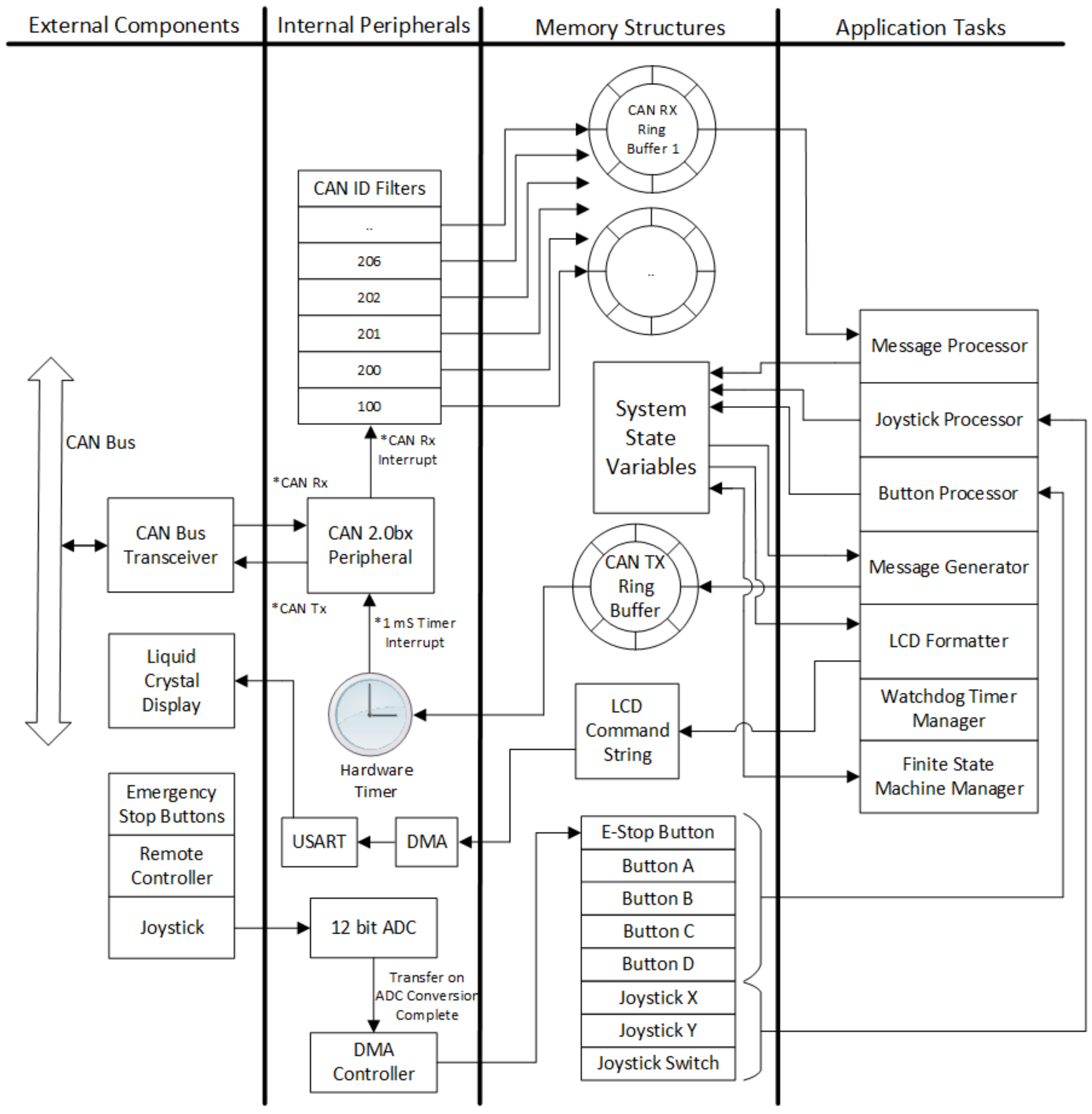
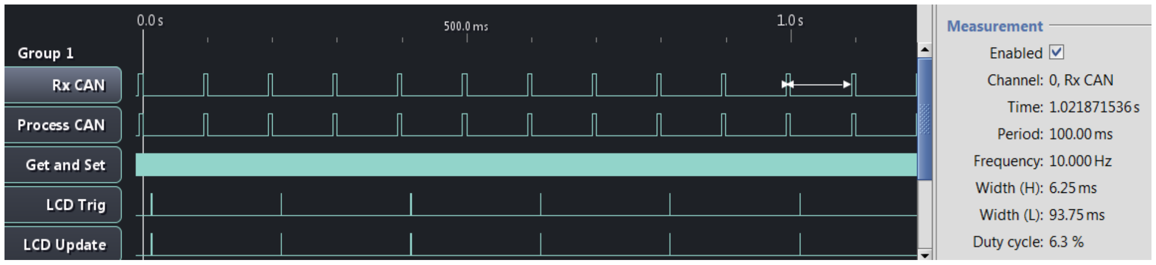


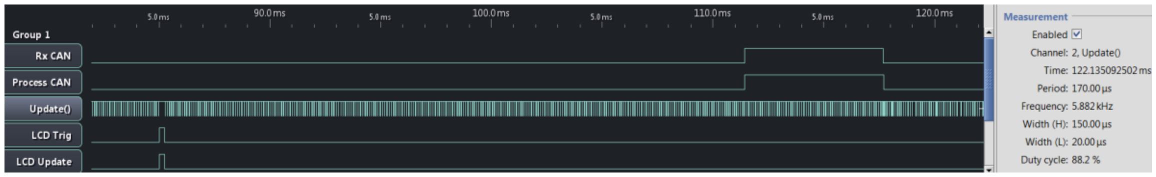
Figure 6.20 - The microcontroller firmware was structured as a memory interface

There were four enabled interrupts in the Controller Node application. Three of the four occurred in 1 mS intervals, and two of those occurred only in short bursts every 100 mS. The remaining interrupt raised an “lcd_update” trigger flag every 200 mS. The worst case execution time was then measured as sum of the execution times of all functions in the mode loops and interrupt handlers, accounting for the possibility that all interrupts become “ready” simultaneously. A “tail chaining” feature of the ARM Cortex processor enabled direct jumping to successive multiple interrupt handlers, without restacking registers from the originally interrupted context. This saved restacking time and allowed for deterministic execution of interrupt processing. The worst-case execution times of the update task was measured to be 8 uS, which was significantly faster than the 100 mS deadline imposed by the system requirements. This is the result of the decision to dedicate a powerful processor for these low-level tasks. The low interrupt handling latency is shown in Figure 6.21 and Figure 6.22.



Measurement
Enabled ☑
Channel: 0, Rx CAN
Time: 1.021871536 s
Period: 100.00 ms
Frequency: 10.000 Hz
Width (H): 6.25 ms
Width (L): 93.75 ms
Duty cycle: 6.3 %

Figure 6.21 – The basic period of the bus was visible in the task execution of the software. The LCD device's 5 Hz update rate was also visible.



Measurement
Enabled ☑
Channel: 2, Update()
Time: 122.135092502 ms
Period: 170.00 µs
Frequency: 5.882 kHz
Width (H): 150.00 µs
Width (L): 20.00 µs
Duty cycle: 88.2 %

Figure 6.22 - The update() function ran longest when it processed an LCD update event, which required under 400 uS to execute, and the update() function was still called many times during bursty CAN message reception.

6.2.4.2 Analog to Digital Converter Driver

Use of the DMA extended to the analog-to-digital converter of the microcontroller, which was able to sample multiple channels in a specified order, each at 200 Hz, causing a DMA transfer request after each conversion. The DMA stream left the processor with an automatically-updated array of memory to read asynchronously, instead of a hardware peripheral (ADC) to manage for every channel conversion process. The array of memory contained the 12 bit samples of pins connected to user buttons, emergency stop buttons, and joystick.

6.2.4.3 Carrier Differential Global Positioning System Serial Byte Protocol Driver

The carrier-differential GPS devices were configurable from the vendor supplied windows application, which controlled update rates, message types, message frequencies, and a wide host of other settings (Swift Navigation, San Francisco, CA). The vast majority of the factory-default settings were retained in the final application of the GPS CAN node. The CAN sled was able to supply the required 5V rail to power the GPS receiver, and only a serial interface was used to collect positioning data. The GPS node acted as the time master of the CAN bus, and derived master system time from the highly reliable 10 Hz update rate of the of the GPS receiver.

All GPS state and event information was automatically sent to the microcontroller at the 1 Mbps baud rate specified by the configuration software. This baud rate was possible because of the short serial wire length between the GPS device and the STM32 microcontroller. All GPS data arrived in packetized bursts of data, which could not be transmitted on the 250 kbps CAN bus as quickly as it arrived over serial protocol. This was remedied with a classical buffering technique, which allowed the firmware to process the serial GPS packets as they arrived, and store results in a CAN message transmit buffer, to be transmitted as a function of system time in 1 mS transmission windows according to the time-triggered system matrix.

The arrival of GPS data from the receiver was plotted in Figure 6.23 along with the transmission of CAN messages in 1 mS windows. Events and durations were timed on a logic analyzer to capture how the buffering process is able to start immediately after the reception of GPS time information.

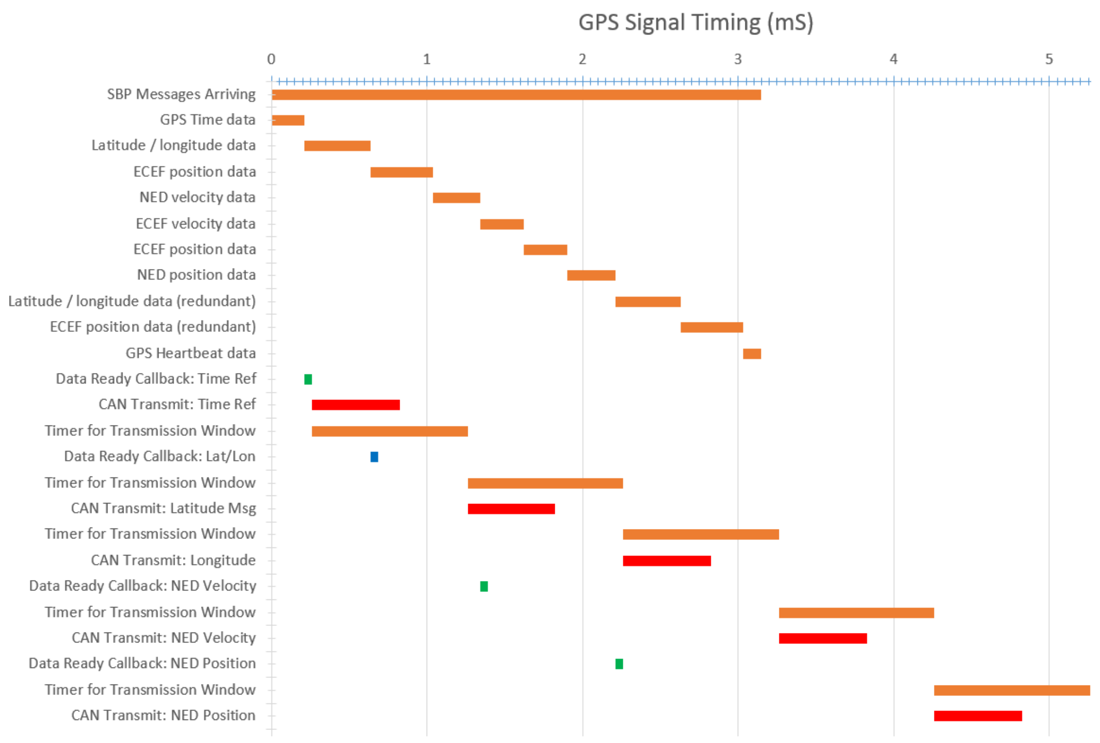


Figure 6.23 - The GPS node buffers the incoming serial GPS data, and transmits at fixed transmission window intervals to meet the system schedule. Data was sent to the microcontroller in 3.15 mS.

The CD-GPS receivers were able to resolve position estimates at a rate up to 10 Hz. Position readings of the rover-side CD-GPS receiver (Figure 6.24) were in a Northing-Easting-Down (NED) coordinate system (Figure 6.25) relative to the stationary CD-GPS receiver with coordinate (0,0) in millimeters as the base. As opposed to using an Earth-centered earth fixed (ECEF) or latitude / longitude coordinate system, the NED coordinates greatly simplified the calculation of cross track error, the definitions of robot and waypoint positions, and required no floating point calculations in the processing operations of the microcontroller.

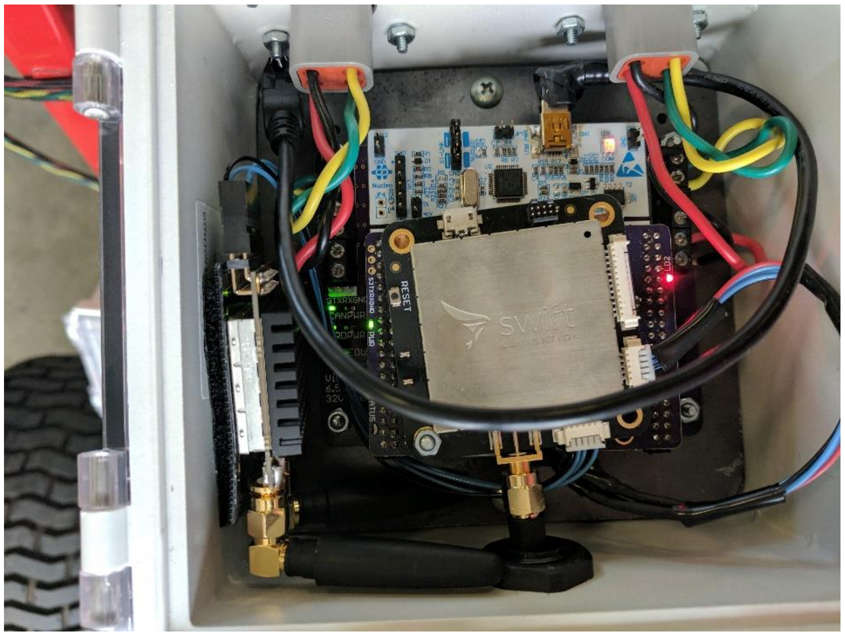


Figure 6.24 - Carrier-differential GPS CAN node (Swift Navigation, San Francisco, CA)

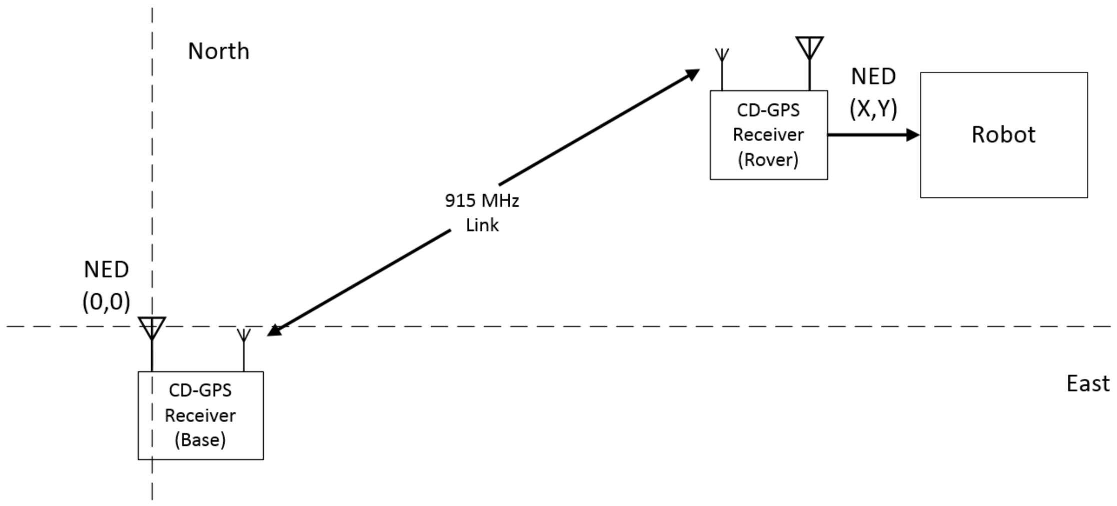


Figure 6.25 - A representation of the coordinate system established by the CD-GPS receivers.

6.2.4.4 Motor Control Driver

Software-control of the UAGV motors was achieved with SyRen 50 (Dimension Engineering, Hudson, OH) 50-amp motor drivers. The drivers were commanded with a simple, unidirectional single-byte serial UART protocol. The serial transfers were single byte, and therefore required no DMA stream or interrupts. The microcontroller firmware simply sent “stop” commands to the motor driver on startup and on the repeated CAN message reception timeout. If the motor node (Figure 6.26) misses either the time reference CAN message or the feedback command message in two consecutive basic-periods of time, the software sent the robot’s motors into a safe stopped state. This safety feature was tested by holding down the “reset” switch on the GPS CAN node microcontroller, causing an absence of time-reference messages on the bus.

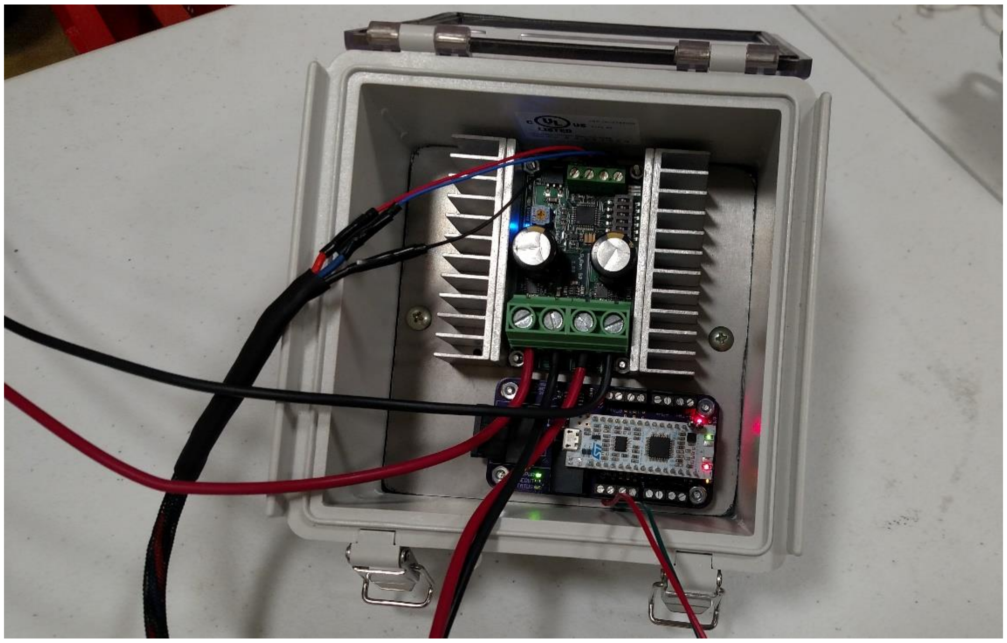
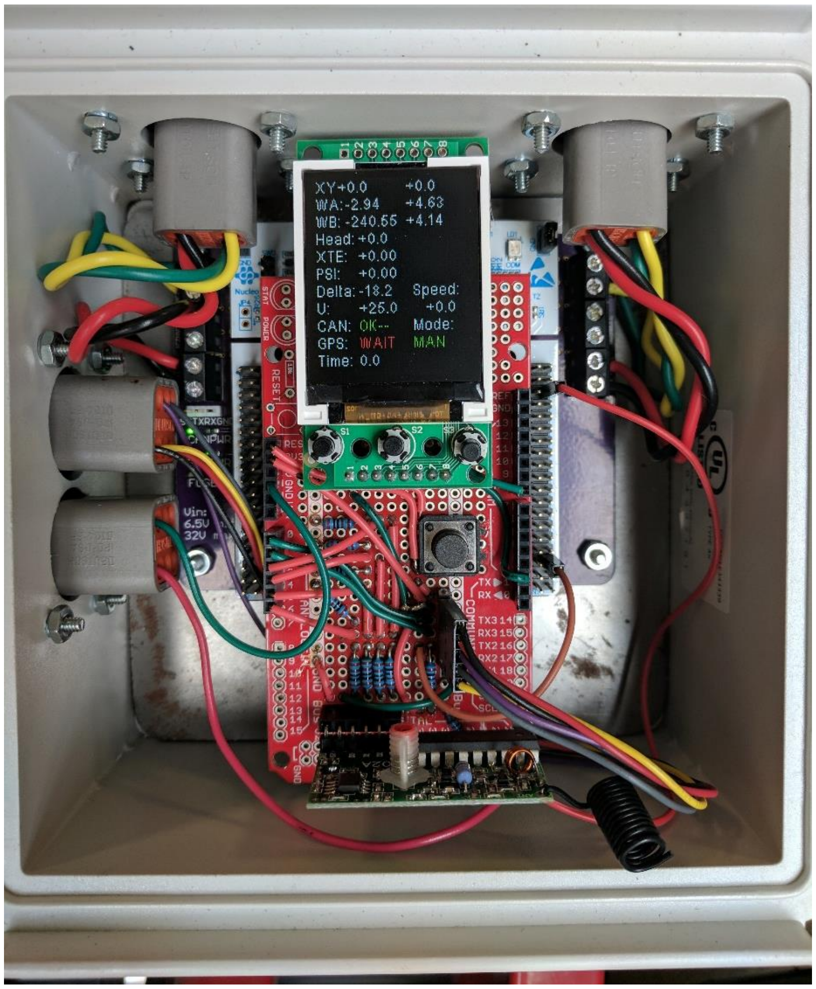


Figure 6.26 - Motor control CAN node

6.2.4.5 Liquid Crystal Display (LCD) Driver

The thin-film transistor (TFT) LCD (Figure 6.27) driver responded to the periodic setting of a global flag representing the passage of 200 mS of time. The driver formatted a long character string of all LCD cursor and color commands, then printed the system variables in the ordered required to write to the screen from top-down, left to right. After string formatting was complete, the command string and it’s length was passed to the DMA peripheral which streamed the data to the LCD serial port, relieving the processor from having to handle the transmission of every individual character. Transfer of the entire string to the LCD took 78 mS, which was sufficiently short for the required 5 Hz update rate which imposed a 200 mS deadline for the entire transfer. Figure 6.28 shows the beginning of the byte-by-byte serial transfer process over the LCD serial pins. The LCD module can be seen to acknowledge every successful command reception by replying with ‘0x55’. Binary states were encoded into a character color scheme, where red indicated a “not ready” state, and green indicated a “no error” state.



```
XY+0.0        +0.0
WA:-2.94      +4.63
WB: -240.55  +4.14
Head: +0.0
XTE:  +0.00
PSI:  +0.00
Delta: -18.2   Speed:
U:     +25.0     +0.0
CAN:  OK--     Mode:
GPS:  WAIT     MAN
Time: 0.0
```

Figure 6.27 - Remote control receiver, reset button, and LCD interface circuit on the "Controller" CAN node

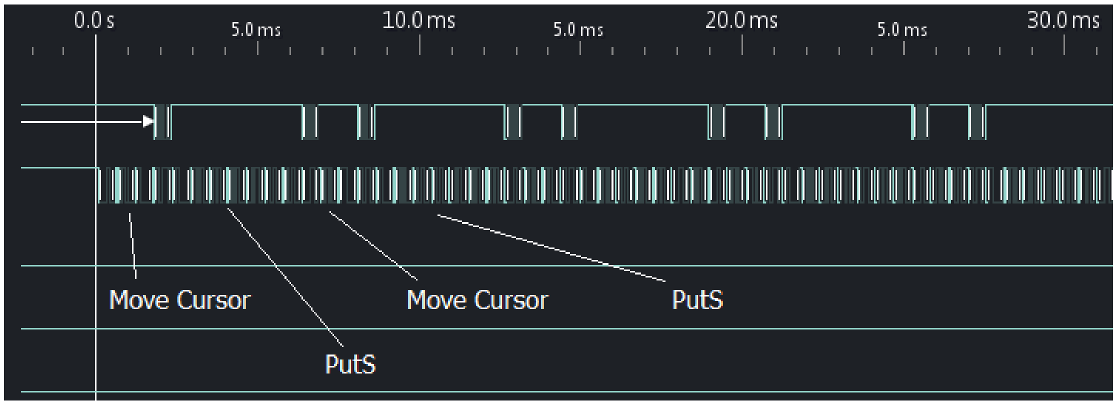


Figure 6.28 - Byte stream of serial LCD commands managed by direct-memory-access peripheral

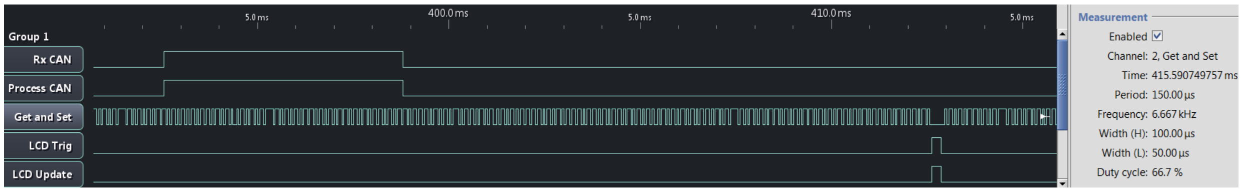
6.2.4.6 Serial Console Output Driver

Each CAN node utilized a hardware USART peripheral of the microcontroller to output software status and variables to a serial stream. The streams were accessible with a standard serial-to-USB converter by opening COM ports on a PC. The stream of variables and system status was reformatted and re-printed to the console at 10 Hz so that every state change could be visibly captured or recorded. The driver for the USART was non-blocking, and utilized a DMA stream to manage the transfer of each byte. A two-dimensional character array was statically allocated in the firmware to buffer any messages it sent to the stream while previously-started DMA stream were still incomplete. This enabled the application to use a low-priority “DMA Transfer complete” interrupt to handle the creation of streams for the waiting buffered messages.

6.3 Results and Discussion

The deployment of software involves a large amount of testing and the development of the C modules in use on the microcontrollers was no exception. The firmware had to sample from a number of sensors, communicate them over a distributed link, process them to produce state variables in the desired engineering units, and do something reasonable after detecting that something has gone wrong – all in a timely manner dictated by the definition of the system matrix.

The resulting performance of embedded software can be difficult to characterize and even interpret in a meaningful way. To show what the software is capable of, an oscilloscope and logic analyzer were both used to verify system performance on virtually all levels. General-purpose output pins of the microcontroller were toggled at the beginnings and ends of application function calls to characterize task execution in the time domain (Figure 6.29). The CAN bus was monitored with an oscilloscope and a CAN bus analyzer to verify the basic period of the system matrix (Figure 6.30 - Figure 6.33).



Enabled [✓]
Channel: 2, Get and Set
Time: 415.590749757 ms
Period: 150.00 µs
Frequency: 6.667 kHz
Width (H): 100.00 µs
Width (L): 50.00 µs
Duty cycle: 66.7 %

Figure 6.29 - The handling of all CAN messages was handled in under 7 mS

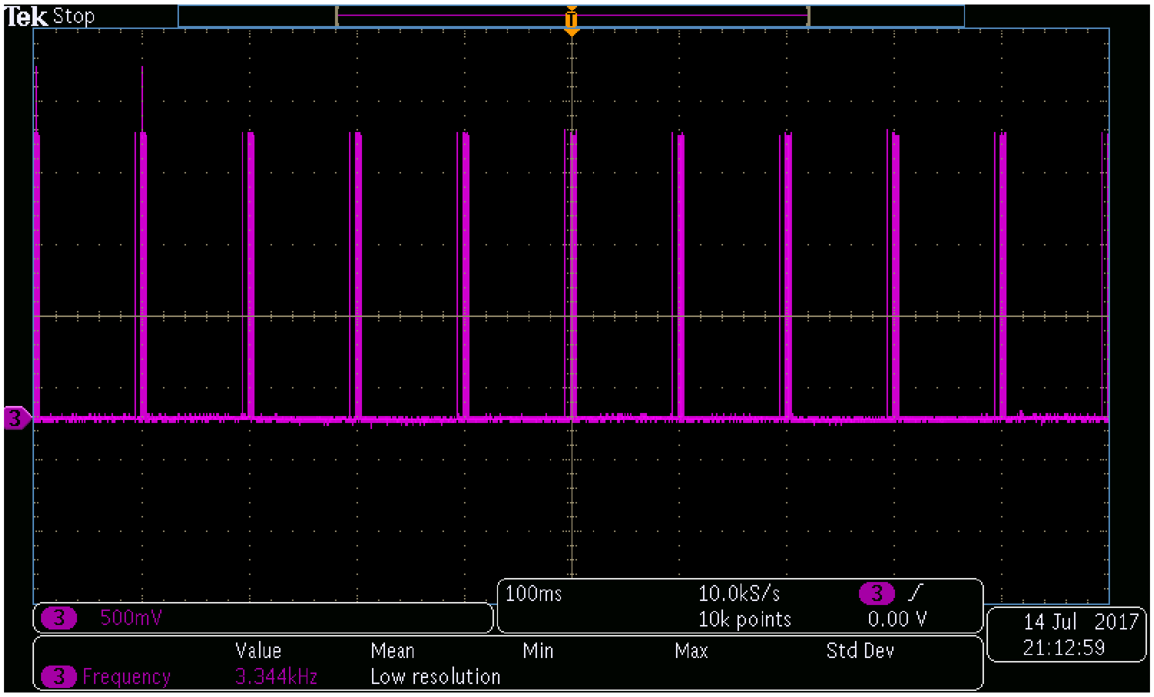

Tek Stop
3 500mV
100ms 10.0kS/s 10k points 3 0.00 V
14 Jul 2017 21:12:59
Value Mean Min Max Std Dev
3 Frequency 3.344kHz Low resolution

Figure 6.30 - The 100 mS basic period is visible on an oscilloscope

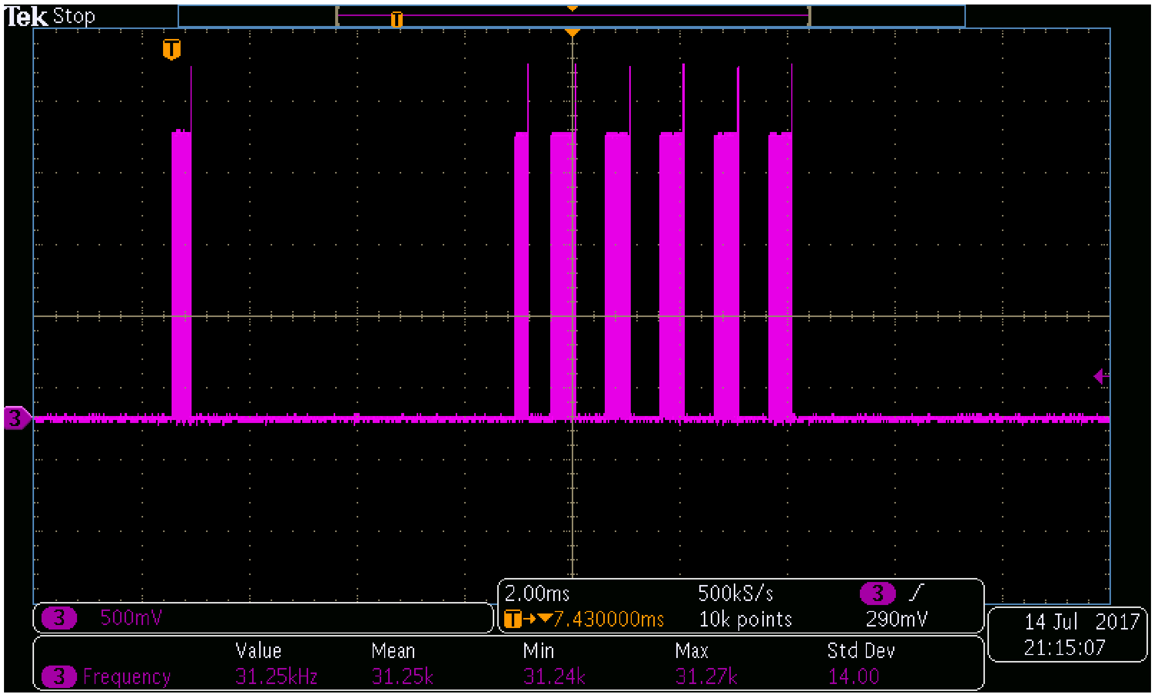

Tek Stop
3 500mV
2.00ms 500kS/s 10k points 3 290mV
T→▼7.430000ms
14 Jul 2017 21:15:07
Value Mean Min Max Std Dev
3 Frequency 31.25kHz 31.25k 31.24k 31.27k 14.00

Figure 6.31 – Basic period capture during GPS signal loss

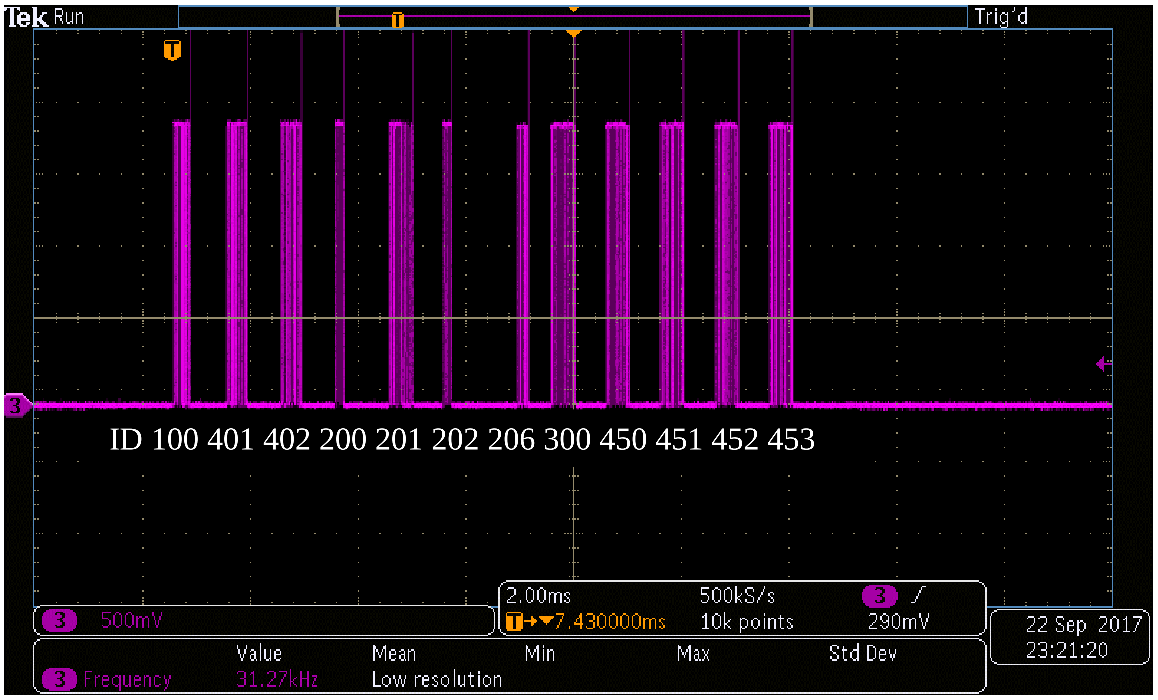
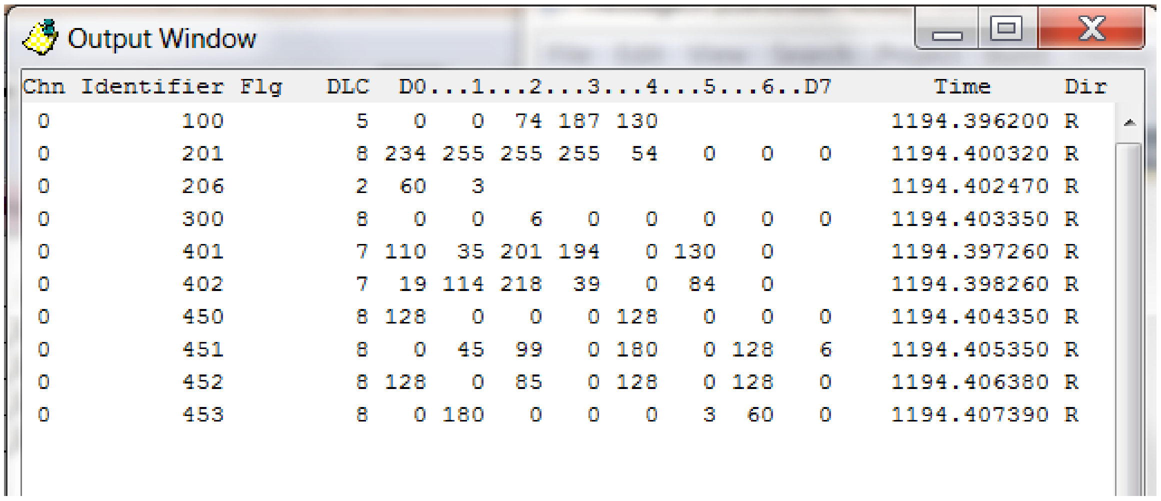


Figure 6.32 - 1 mS message windows



```
Chn  Identifier  Flg   DLC   D0...1...2...3...4...5...6..D7          Time     Dir
 0          100          5    0   0  74 187 130                   1194.396200 R
 0          201          8  234 255 255 255  54   0   0   0       1194.400320 R
 0          206          2   60   3                               1194.402470 R
 0          300          8    0   0   6   0   0   0   0   0       1194.403350 R
 0          401          7  110  35 201 194   0 130   0           1194.397260 R
 0          402          7   19 114 218  39   0  84   0           1194.398260 R
 0          450          8  128   0   0   0 128   0   0   0       1194.404350 R
 0          451          8    0  45  99   0 180   0 128   6       1194.405350 R
 0          452          8  128   0  85   0 128   0 128   0       1194.406380 R
 0          453          8    0 180   0   0   0   3  60   0       1194.407390 R
```

Figure 6.33 - CAN bus activity of one basic period. Position and heading messages are not present when the GPS lock is not "fixed".

6.4 Conclusions

The control system requirements of the UAGV were satisfied within the constraints of the hardware selected, and were implemented with a TTCAN software stack that was verified to successfully schedule system messages even during predicted instances of missed message deadlines. Each of the UAGV components was integrated into a functional time-triggered system architecture, demonstrating a measure of system robustness, an awareness of message losses, and a user interface, validating the intended design.

Chapter 7 Control Design for Unmanned Agricultural Ground Vehicle

7.1 Introduction

In Agricultural and Biological Systems engineering curriculum, any exposure (if there is any at all) to control systems is usually a presentation of PID control, where the student is given three tuning parameters that are heuristically adjusted on the basis of some generalized tuning guidelines - or on no basis at all. This method of exposure leaves the impression that PID control is a silver bullet – the ultimate control solution that should be sought-after first. Something lost in this attitude is the analytical consideration of system dynamics with the compensator.

Objective: Design and demonstrate a basic state space control system on the UAGV with TTCAN Control Module Layer (from Section 1.1)

1) Introduce state space modeling and feedback control concepts
2) Show controllability of UAGV and the convergence of system states
3) Comment on implications for future control module layer research

7.2 State Space Modeling and Feedback Control Design

7.2.1 Continuous and Discrete-time Linear Time Invariant System Representation

As discussed in Chapter 2, the dynamics in an agricultural vehicle can be difficult to determine due to the easy slide into excessive system complexity. Consider the following scenario in the automatic steering control of a tractor (Figure 7.1).

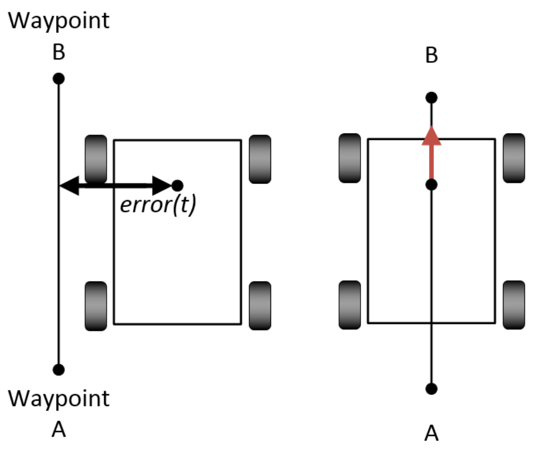


Figure 7.1 – Derivation of a 1-dimensional error signal. The resulting position and angle of the velocity vector allows for straight steering travel on level ground.

A single cross track error signal, *error*(*t*), is described by the scenario as the magnitude and direction of the cross-track error of the GPS antenna from some straight line defined by a pair of waypoints. This cross track error signal is commonly used as feedback in agricultural vehicle controllers. Darr (2004) derives a complete PID controller, step by step – demonstrating the effects of each successive control design step. This controller was shown to work for some forward driving speed. It enabled tight cross-track error regulation. Assuming a PID controller is sufficient for control of the vehicle using this cross track error signal presumes assumptions about the dynamics of the error signal, implying that the cross track error signal can be modeled to behave as a low-order plant with constant delay. This approach to control comes with no formal performance guarantees for system stability or parameter sensitivity.

When considering a wider scope of navigation scenarios, it can be beneficial to control the vehicle under a wider variety of operational states. Consider the positioning of a vehicle when a strong disturbance (steep lateral slope) to the system causes errors not measured by the GPS position. A controlled steady state cross track error signal only assumes the heading direction of the vehicle to be parallel to the desired path of travel (Darr, 2004, p. 57). If the vehicle is on a steep incline with considerable tire slip, a controller that lacks vehicle heading information might allow the vehicle’s rear hitch position to stray from the desired path – offsetting any hitched agricultural implement equipment (Figure 7.2). Including additional error signals could remedy the additional errors simultaneously.

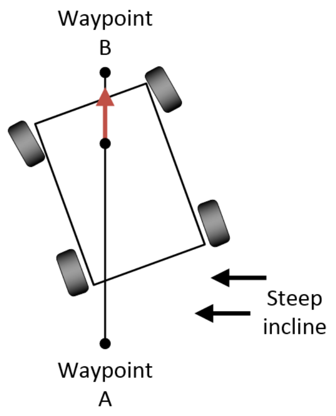


Figure 7.2 - A control error of zero is achieved while the rear hitch position is considerably offset and the velocity vector still indicates straight line travel from the perspective of the error signal.

Other forms of control however, take advantage of more advanced system models that account for multiple system inputs and outputs. Modeling the system involves deriving equations of motion using laws of kinematics and dynamics. The equations may be nonlinear, and therefore are often linearized around a small operating point. The resulting model is a linear time-invariant (LTI) system for which a controller may be easily designed. For an LTI system with p inputs, q outputs, and n states, the system parameters can be organized into a set of system matrices:

A – System matrix [n x n]: Describes the current system states' contribution to the current rate of change of the system states.

B – Input matrix [n x p]: Describes the weight of the system inputs' contribution to the rate of change of the system states.

The rate of change of the current states is described by the sum of these matrices' weights on the systems' current states and inputs that give the control engineer a model for the system's behavior, where the weights are the physical system parameters. Powerful simulation and control design techniques for the system can be done when it can be described by the continuous time state space LTI representation in equation 7.1:

$$\dot{x} = \mathrm{A}x + \mathrm{B}u \tag{7.1}$$

The continuous rate-of-change of the system states is simply the sum of the weights (system parameters / coefficients of the linearized differential equation) on the current states with the weights (gains) on the current inputs.

System states are not always physical states to begin with, so control design needs to be done with the help of a system output equation, which describes the outputs of the system as the sum of the linear combination of the availability of the system states, and the direct influence of system inputs. Together, C and D describe the output of the continuous time system in equation 7.2:

$$y = \mathrm{C}x + \mathrm{D}u \tag{7.2}$$

where

C – Output Matrix [*q* x *n*]: Describes the linear combination of system states that contributes to the outputs of the system; which may include the variables that are to be controlled, (temperature, speed, angle etc.)

D – Feedthrough Matrix [*q* x *p*]: Describes the linear combination of input signals that directly add to the outputs of the system. In most cases, this is the zero matrix.

Graphically, the state space equations (state and output equations) can be represented as a flow signals between operators, as shown in Figure 7.3. The nature of the system model as a multivariable differential equation becomes evident. The feedback across an integrator operator shows a system that is described by both a state vector and their derivatives.

Reviewing the matrix descriptions of the continuous time state space equations reveals their graphical equivalence in Figure 7.3. Both A and B contribute to the state derivative $\dot{x}$, the representation of state change. C and D contribute to the representation of the system output *y*. We see two sums of the state space equations as the intersecting summation nodes. Additional examination of the state space equations shows two uses of the signal vectors *x* and *u*, which manifest in the diagram as signal branches. Vector/matrix multiplication is then the operator described by a vector signal pointing to a matrix.

While noise in the system is not represented in the figure, it is often considered in control design techniques. For presentation simplification, this basic model of a linear time-invariant system does well enough for basic simulation demonstration purposes, but is likely not sufficient for a full stability or parameter sensitivity analysis.

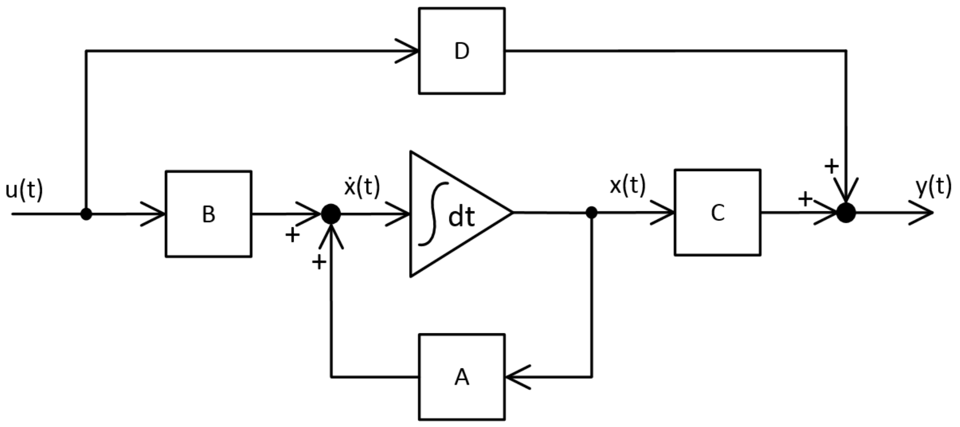


Figure 7.3 - Continuous-time, linear time-invariant system

In the control design process, it is common to use this continuous-time model, but in the implementation of state feedback controllers on computers, the continuous process is sampled as shown in Figure 7.4.

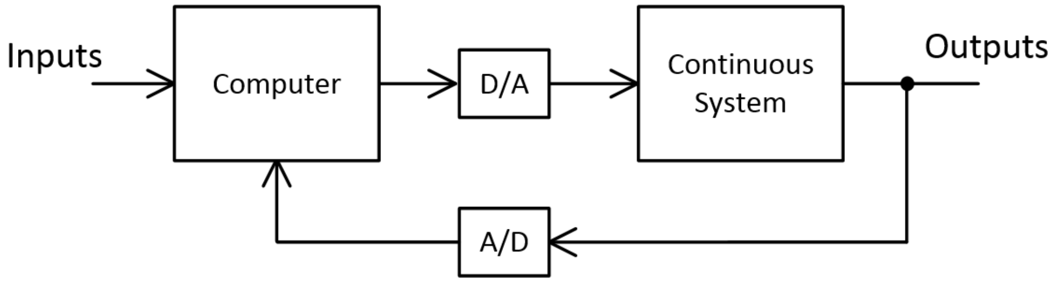


Figure 7.4 - Conversion to a discrete-time system to by sampling

From the sampled system outputs, a controller in the computation node generates the control signals that drive the system through a digital to analog converter or actuator. The controller parameters can be designed according to simulations of the sampled continuous time system, or by converting the continuous time system to a discrete time system by defining a discrete time model where

$$x[kT+T] = Fx[kT] + Gu[kT] \tag{7.3}$$

and

$$y[kT] = Hx[kT] + Du[kT] \tag{7.4}$$

for

$$F = e^{AT} \tag{7.5}$$

and

$$G = \int_0^T e^{A\tau} B d\tau \tag{7.6}$$

and

$$H = C \tag{7.7}$$

where *k* is the sample count and *T* is the sampling period. Equation 7.3 is noticeably analogous to the system equation 7.1, where *F* is simply the state transition matrix

$$\Phi(t) = e^{At} \tag{7.8}$$

for the discrete time instant of *T*. This conversion process can be easily done with computational tools such as MATLAB's c2d(sys, T) function.

F[n x n] is then a specific of instance of the state transition matrix; a description of the contribution of the current state sample *x[kT]* to the changes observed on the next sample of the state vector *x[kT+T]*, given some sampling period *T* and initial condition x_0. The integrator of the continuous-time system can be represented as a delay in the discrete time system as shown in Figure 7.5. The delay element has a delay of one sample, which has the time-equivalent value of the sampling period *T* of the discrete system.

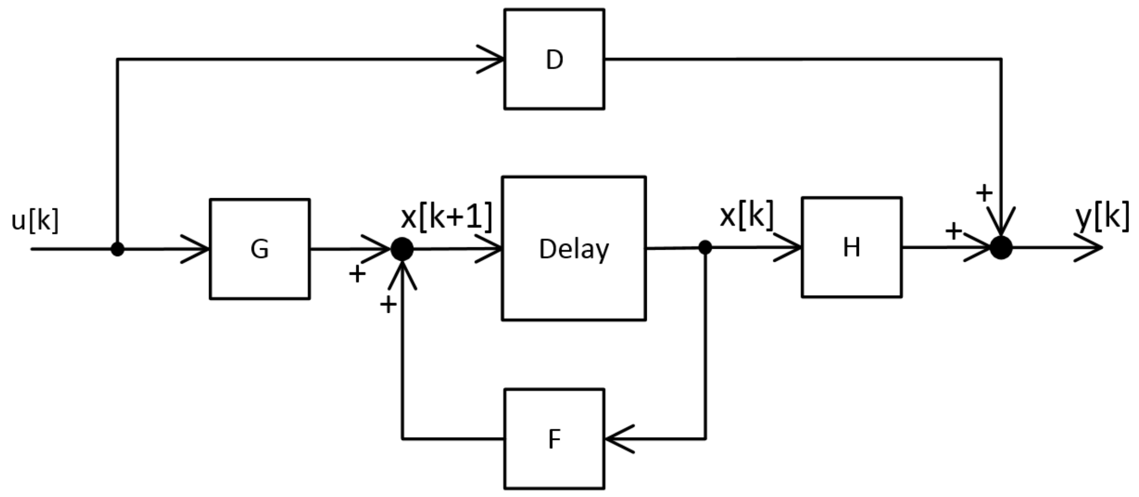


Figure 7.5 - Discrete-time linear time-invariant system representation

7.2.2 System Characterization

Given any system initial state *x₀*, if the inputs to an LTI system can be used to drive the states to zero ($x_0 \to \mathbf{0}$), then the system is said to be fully controllable. Controllability can be determined using the system's controllability matrix (eq. 7.9).

$$CO = [B|AB|A^2B| \dots |A^{n-1}B] \tag{7.9}$$

The system is fully controllable is *CO* has full rank, or, with A of dimension n x n

$$rank(CO) = n \tag{7.10}$$

where *n* is the number of system states.

The dual to system controllability is system observability. When dealing with LTI systems, some of the states might not be available for direct measurement. For state feedback control of LTI systems, the states can be derived from the outputs of the system if the system is fully observable. If a system is observable, then 7.11 holds true, and a state estimator, or observer can be used to provide state feedback using the available system states and inputs.

$$\operatorname{rank}\begin{bmatrix} \mathrm{C} \\ \mathrm{CA} \\ \vdots \\ \mathrm{CA}^{n-1} \end{bmatrix} = n \tag{7.11}$$

7.2.3 State Feedback Control

Feeding back some linear combination of the system states (*Kx*) to the input can enable the input signal to drive the output *y* to the zero vector by driving the states *x* towards the zero vector (Figure 7.6).

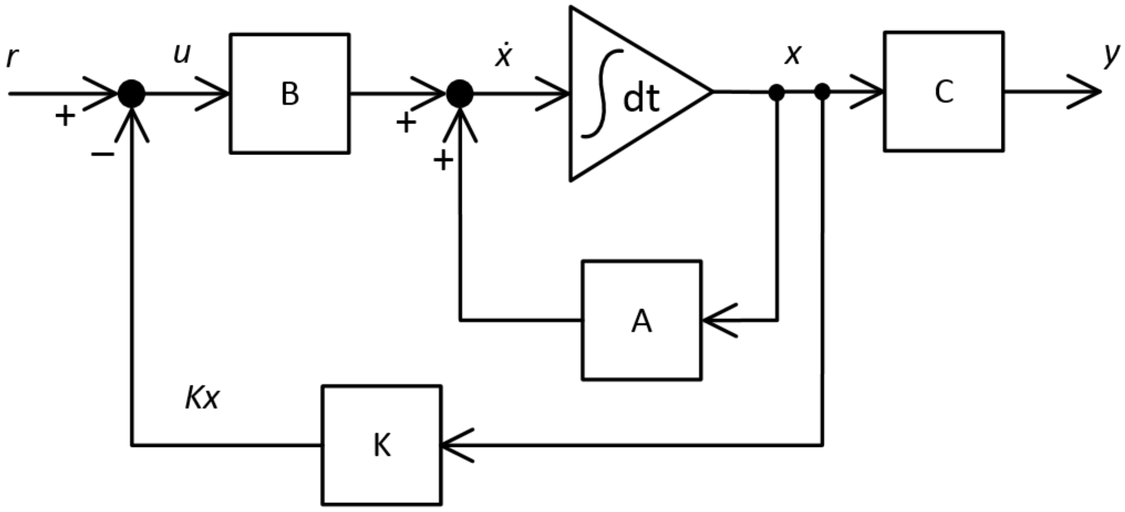


Figure 7.6 - State feedback

Meaning that the feedback signal is now a linear combination of the measured states x, or estimated system states $\hat{x}$, and the feedback gains (*K*), assuming the state regulator scenario where *r* = 0.

$$u = -\mathrm{K}x \tag{7.12}$$

Note there are some system model parameters that are required to make the feedback signal this simple to produce:

1) All the system states are directly measureable.
2) The LTI system has full controllability.

If these conditions can be met according to the system model, equation 7.12 can be substituted into equation 7.1, resulting in an LTI system (Figure 7.6, equation 7.13) with state feedback to regulate the system.

$$\dot{x} = (A - BK)x \tag{7.13}$$

where K is the feedback gain matrix. The stability of the LTI system can be greatly influenced by K, which needs to be designed such that the system poles are in the left-half of the complex plane, or:

$$Real(eigenvalues(A - BK)) < 0 \tag{7.14}$$

Modern state space control methods deal with the derivation of K, given some well-characterized system defined by A, B, C, and D. For optimal control design given a continuous-time fully controllable and fully observable LTI system, the dlqr() function in MATLAB (The Mathworks Inc.) returns the controller gains *Kd* for digital control of a continuous time LTI-modeled system.

This control technique among others deserves a more detailed treatment due to the variety of complex systems to model and control. The methods extend further to deal with system identification, estimation, and optimal control in the realms of offline, online, and adaptive control scenarios. Some of the same methods even extend to deal with time-variant linear systems, and the broad scope of non-linear systems. Excellent treatments of the topics can be found by (Brogan, 1991) and (Franklin, 1998). A more specific treatment of the topics in the context of vehicle steering control, as well as a derivation of the model used for control simulation in the next section was done by (O'Conner, 1998).

7.2.4 Kinematic state space model

Kinematic models of line-following vehicles tend to share a few common system states in the feedback controller. The models usually include some form of the cross track error measured from some point on the vehicle as a system state, a heading angle state, and a steering angle state (Bell et al., 1998; Elkaim et al., 1997; O'Conner, 1998). These states as shown in Figure 7.7, can be thought of as individual error signals that, when regulated to zero by a controller, describe a vehicle that perfectly follows the line described by waypoints W_a and W_b. The linearized model of system kinematics is shown in Figure 7.8 and a detailed derivation of this model is described by O'Conner (1998).

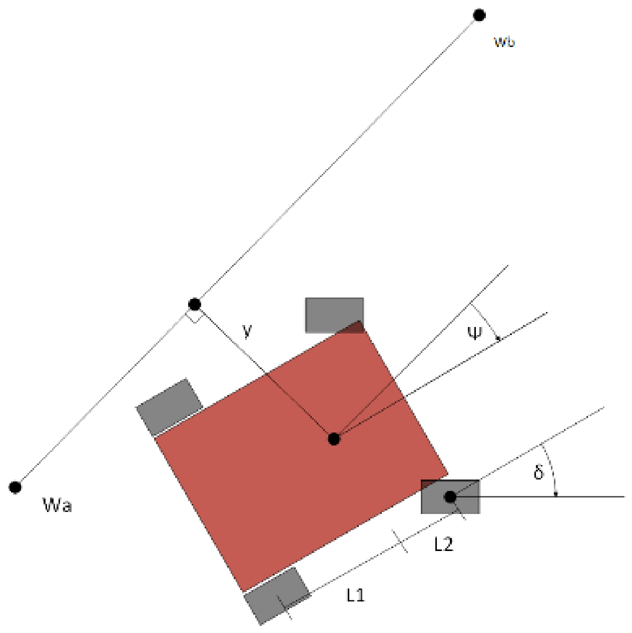


Figure 7.7 - State-space model for navigation (O'Conner, 1998)

In using this model for control design on the UAGV, simulation of the model demonstrates some of the systems requirements and a good starting point in selecting the feedback gains for the navigation controller, even if the model is fairly inaccurate. This also demonstrates the controllability of the UAGV from gains informed by the simulation points to some degree of validity to the model.

First, the full model shows that the system input is a steering speed command to the steering actuator. This is also how a human driver would actuate, or influence the states of the system. The model also shows that a constant low driving velocity is assumed with no tire slip present. Violating these assumptions makes the model very inaccurate, and likely takes the system to some non-linear range of operation. This model supposes that the machine operates as a linear system if the states remain relatively close to zero. This means that letting one or more of the states become large enough in magnitude could send the system into instability, which manifests itself as steady or growing system oscillations. For a robot navigating on a line, this might mean that the vehicle would wiggle across the line back and forth, without ever steadily tracking the line, or driving all the states to zero.

$$x = \begin{bmatrix} y \\ \Psi \\ \delta \end{bmatrix} A = \begin{bmatrix} 0 & V & \frac{VL_1}{(L_1+L_2)} \\ 0 & 0 & \frac{V}{(L_1+L_2)} \\ 0 & 0 & 0 \end{bmatrix} B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 7.8 - System Model (O'Connor, 1998)

For V = 2 meters/second, L_1 = 0.5 meters, and L_2 = 0.5 meters, A and B produce a controllability matrix CO with full rank. Since all system states of the UAGV are measured using GPS and a steering actuator sensor, H (the observability matrix) is full rank. The system is shown to be both fully controllable and fully observable – simplifying the control design. The system also has no D matrix, which means there is no feedthrough control effort accounted for. The combination of these model observations results in a system where C is abstracted away since all system states are directly observable and considered the same variables as the outputs, and where D = 0.

Simulating this system in software was done with a set of initial conditions and a loop that performed the Euler integration method to derive future system states to estimate the behavior of the model at a frequency of 10 Hz. An example system step response (Figure 7.9) was generated to demonstrate convergence of the system model states using a gain matrix K, derived from the dlqr() function in MATLAB (The Mathworks Inc.). State convergence is simulated in Figure 7.9 with the following parameters:

Td = 0.1 s

V = 0.25

L1 = 0.255

L2 = 0.845

K = [5.0 4.0 3.0]

X0 = [1.0 0.0 0.0]’

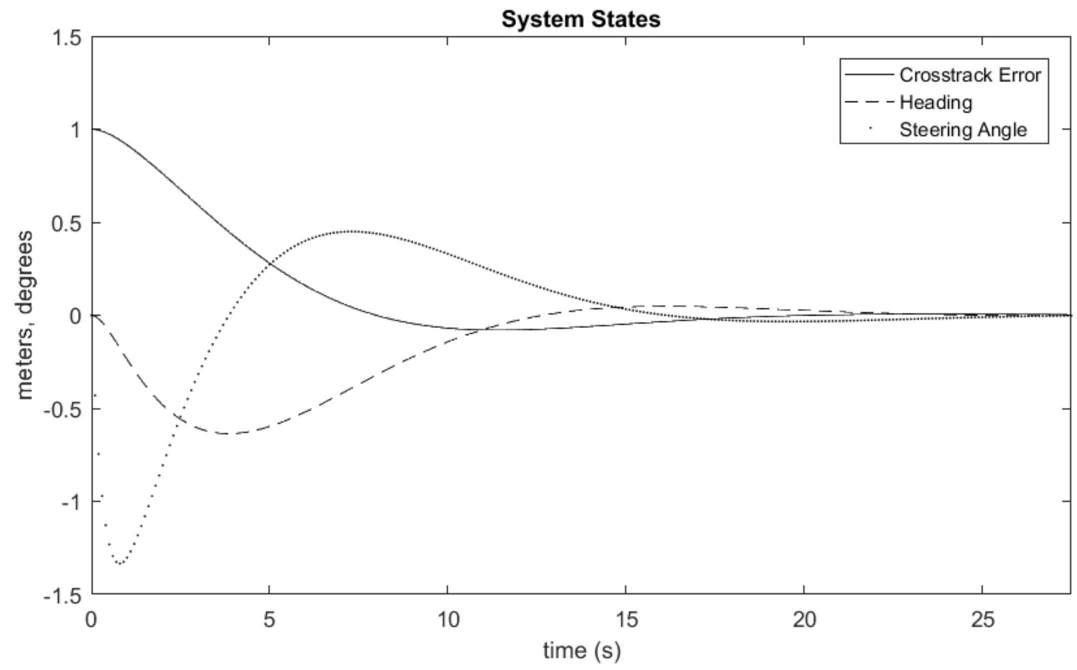


Figure 7.9 - Simulation of state convergence for the O'Connor UAGV model.

7.2.5 State Estimation and Augmented Models

The lack of state availability from a system model can yield a need to estimate the unavailable states. This can be done with the addition of an estimator or observer, which monitors the current system states and inputs, and computes state estimate vector ($\hat{x}$) from an internally simulated version of the system model (Figure 7.10). An estimator can provide state estimates usable for full-state feedback control. Some types of estimators can utilize sensor noise and modeling inaccuracies as parameters of the system to estimate the state vector from noisy sensor measurements.

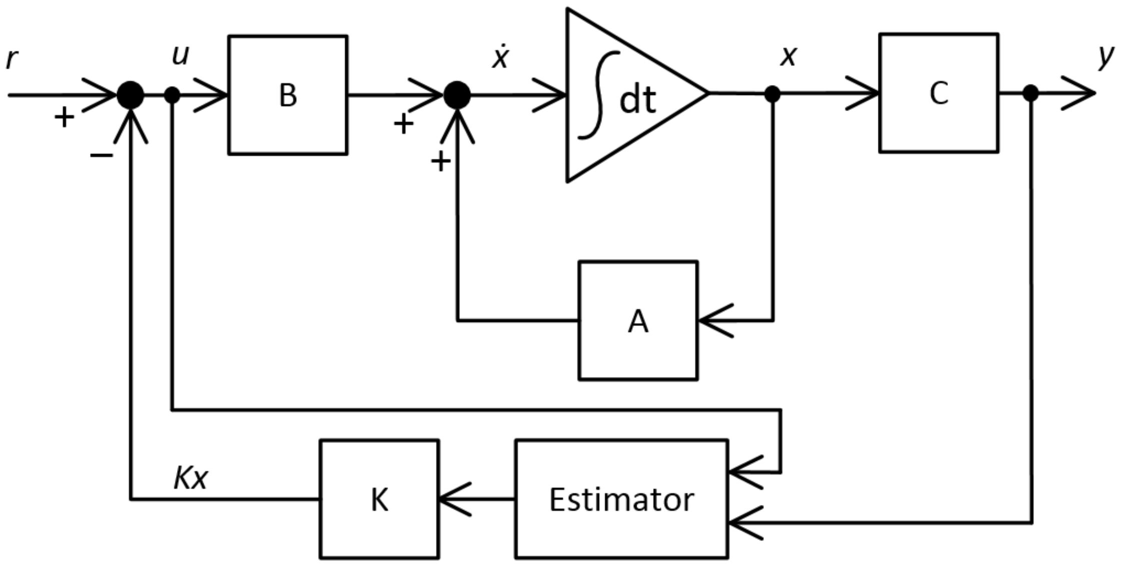


Figure 7.10 - Addition of a state estimator

Additionally, estimators can utilize augmented system models to estimate and capture offsets and biases in sensor measurements when modeled as observable (added to observability matrix H), non-controllable states (no dynamics in system matrix A). If a poorly calibrated compass and misaligned steering angle sensor on a UAGV caused steady state error in controller performance, the offsets could be estimated and then compensated for in the state estimate (Figure 7.11).

$$x_{est} = \begin{bmatrix} y \\ \Psi \\ \delta \\ \Psi_{bias} \\ \delta_{bias} \end{bmatrix} \quad A_{est} = \begin{bmatrix} 0 & V & \frac{VL_1}{(L_1+L_2)} & 0 & 0 \\ 0 & 0 & \frac{V}{(L_1+L_2)} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B_{est} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad C_{est} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Figure 7.11 - Estimator model can be augmented to include bias states

If the incline on a hill is enough to cause a steady-state error on the cross track distance state, the addition of a controllable, non-observable state that accumulates with the magnitude of the cross-track error in every time step can serve as a state integral that can be fed back to the actuator through an augmented feedback matrix (K) to account for the new state (Figure 7.12). Notice that such an integrator state includes no element in H since it is not observable, even though it can contribute to the feedback signal with an augmented K. The addition of an integrator state can increase the tracking performance of a controller with a non-zero reference input *r* in Figure 7.10.

$$x_{est} = \begin{bmatrix} \int y \\ y \\ \Psi \\ \delta \\ \Psi_{bias} \\ \delta_{bias} \end{bmatrix} \quad A_{est} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & V & \frac{VL_1}{(L_1+L_2)} & 0 & 0 \\ 0 & 0 & 0 & \frac{V}{(L_1+L_2)} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B_{est} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad C_{est} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Figure 7.12 - Estimator model can be augmented with an integral state

Simulating state estimation can be done with the addition of noise in the sensor measurements, and use of an augmented model in an estimator. See an example of the previous simulation with included noise, and the resulting estimated states in (Figure 7.13).

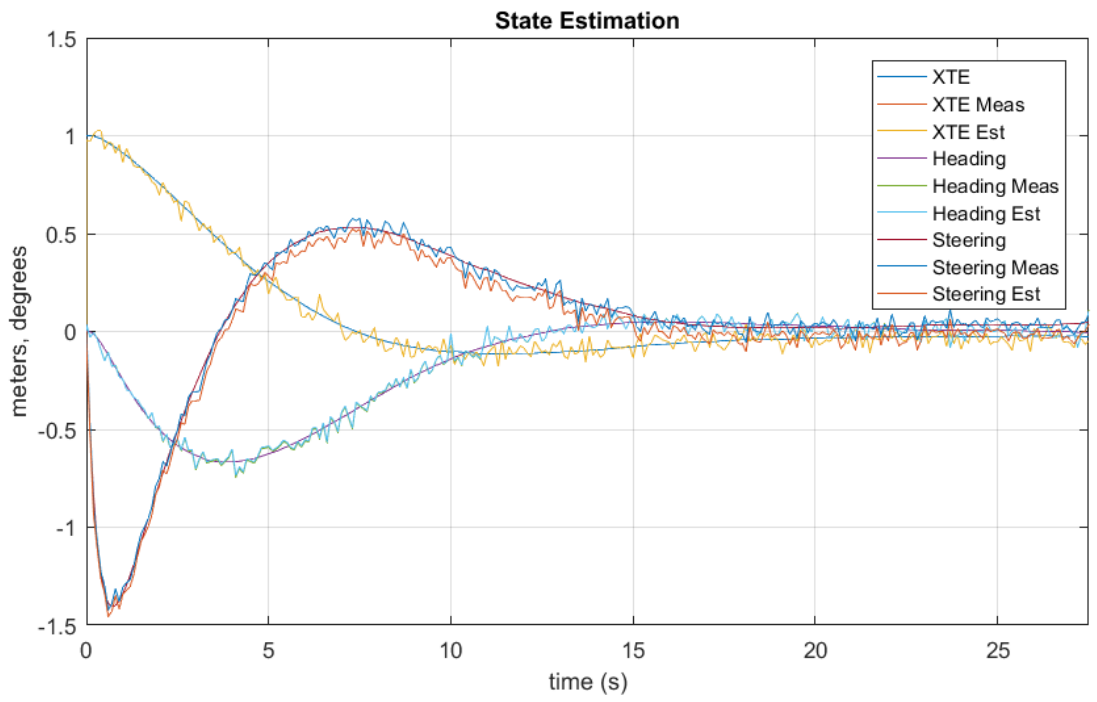


Figure 7.13 - Simulated state estimation to remove measurement noise

7.3 Materials and Methods

Demonstration of state space feedback control was decidedly necessary to show successful utilization of the TTCAN-based control module layer of the UAGV. A line defined by two waypoints with NED coordinates on the Nebraska Tractor Test Track was used to validate the control module layer's ability to retain controllability of the UAGV for each time step of feedback control (Figure 7.14). The line was 243 meters long, and enabled the controller to reach steady state conditions. Manual operation of the CD-GPS system was used to measure the waypoint coordinates ahead of time.

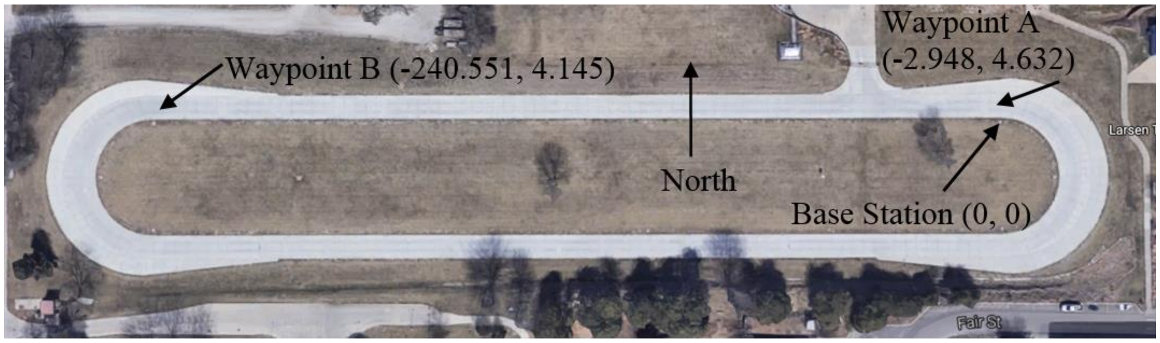


Figure 7.14 - Nebraska Tractor Test Track route for autonomous navigation tests

Feedback gains were derived from MATLAB's (The Mathworks, Inc.) implementation of linear quadratic regulation (LQR) methods (dlqr()). Resulting system behavior with the generated gains were simulated in MATLAB before uploading the gains to the Controller CAN node. To show the transparency of the control module architecture during test runs, no state estimator was used for the generation of the feedback signal. Controllability was sought to be retained through the control module layer running the control algorithm in real-time. Consistent convergence of the system states and system stability was desired for step inputs on single states and on multiple states. Tests for state convergence at 1 m/s were arranged for an initial cross track error of 1.0 meter, and heading error and steering angle at zero degrees.

7.4 Results and Discussion

The states converged (Figure 7.15) to nominal minimum values after 15 seconds of travel time, and followed the waypoint line (Figure 7.16).

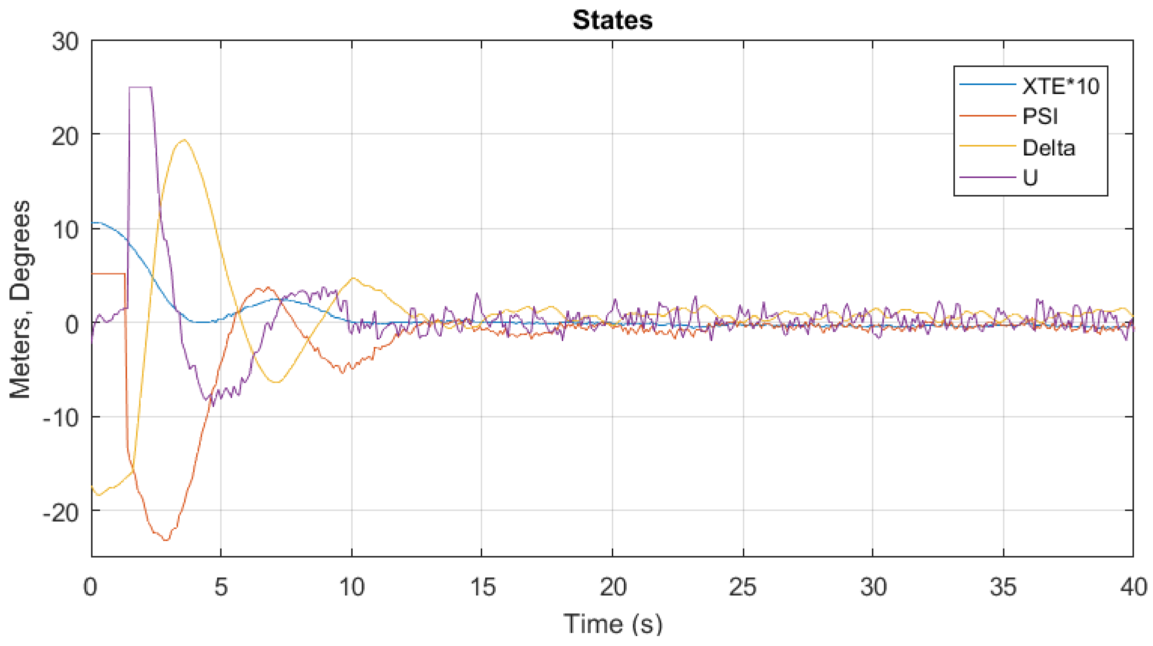


Figure 7.15 - State convergence from step response test

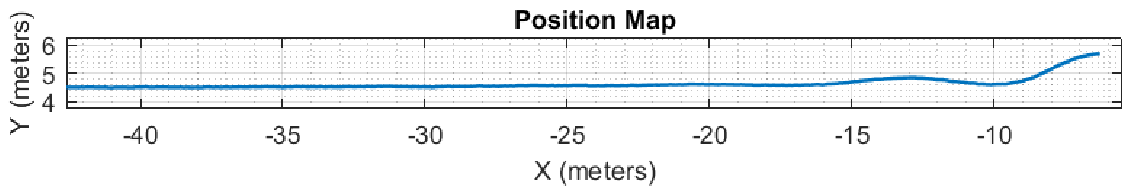


Figure 7.16 - NED coordinates of step response test

The mean of the cross track error was -2.7 cm during steady state, which was evident from distribution of the error (Figure 7.17) and in the steady state value of the error visible from (Figure 7.18). This offset could be reduced with a state integrator on an estimator, but would obscure presentation of the plant's available controllability from just noisy sensor measurements alone. Another step response is depicted I (Figure 7.19), where the elements of the gain matrix had about half the magnitude of the original controller due to LQR parameter adjustments. The step response is noticeably less aggressive.

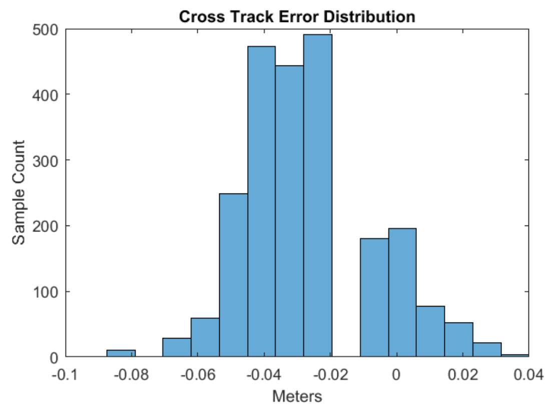


Figure 7.17 - Cross track error distribution

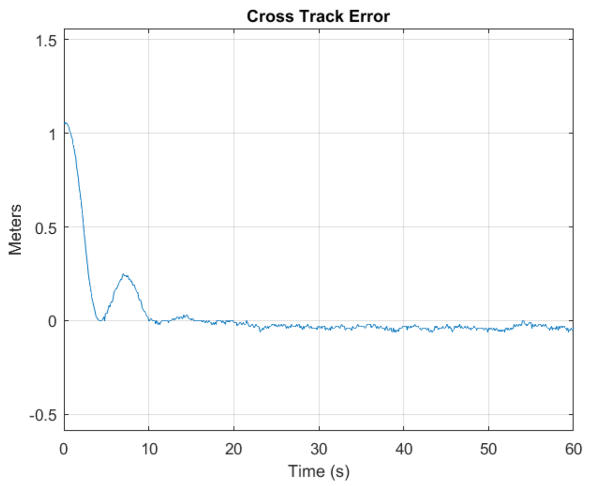


Figure 7.18 - Cross track error time domain response

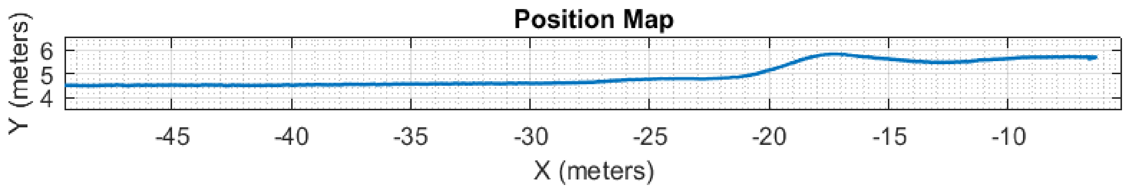


Figure 7.19 - Position control with less aggressive controller gains

To evaluate controller performance under less convenient initial conditions, the robot was posed facing Southeast and 5 meters off the waypoint line (Figure 7.20). The states were still found to converge after a period of actuator saturation (Figure 7.21). The steering wheels were commanded to steer as far as they could before the steering CAN node limited the actuator position, and the controller CAN node firmware saturated its own command output to the CAN bus. The UAGV made a turn as tightly as could be commanded and reached steady state regulation after 15 seconds of run time. The mean cross track error was -4.4 cm during steady state conditions (Figure 7.22).

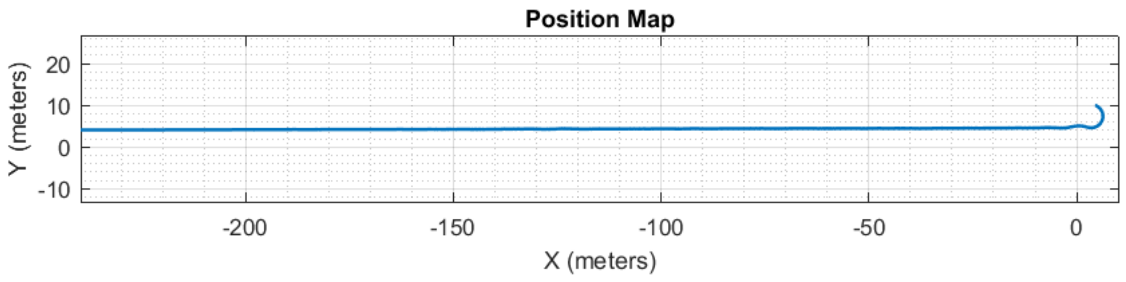


Figure 7.20 - Travel path from 'inconvenient' initial state

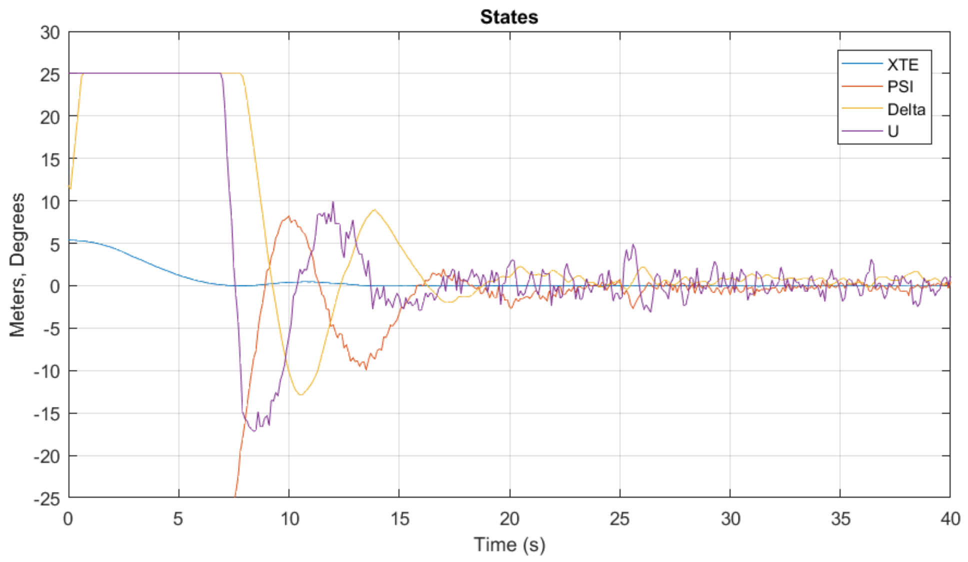


Figure 7.21 - State convergence from larger variety of state vectors

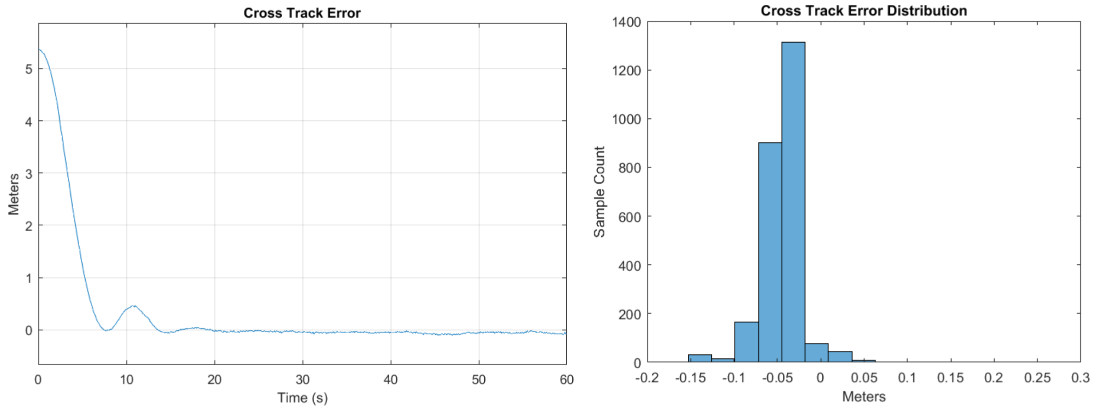


Figure 7.22 - Cross track error steady state was reached in 15 seconds of time, and had a very similar error distribution to the first test

For a third variety of verification tests, the UAGV was loaded with a filled water sprayer on the hitch to informally identify any obvious model parameter sensitivities (Figure 7.23 and Figure 7.24). The model was kinematic, not accounting for mass distributions, accelerations, or forces. Identifying any major model sensitivities to parameters other than velocity and sensor positioning would indicative of very poor modeling. LQR parameters were adjusted to favor the cross track error state in an attempt to reduce the steady state cross track error quickly. The complete cancellation of steady state error was unlikely without an integrator state added to a state estimator. The new gains commanded the actuator harder during steady state and likely required far more energy to satisfy regulation (Figure 7.25 and Figure 7.26). The mean cross track error for the loaded test of 1 cm during steady state was indicative of better steady state performance from previous tests (Figure 7.27 and Figure 7.28).

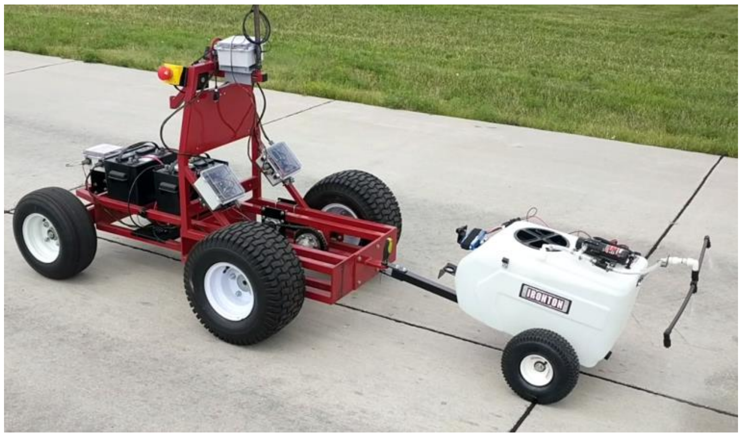


Figure 7.23 - Test setup for loaded UAGV runs

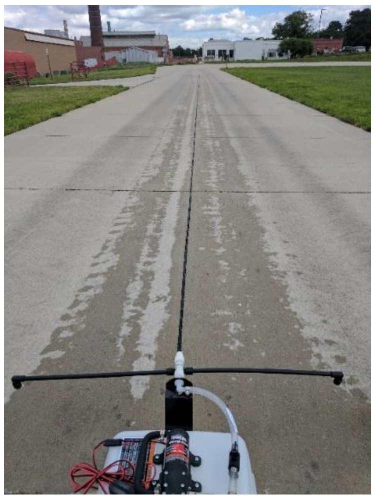


Figure 7.24 - Spray pattern from straight line navigation control

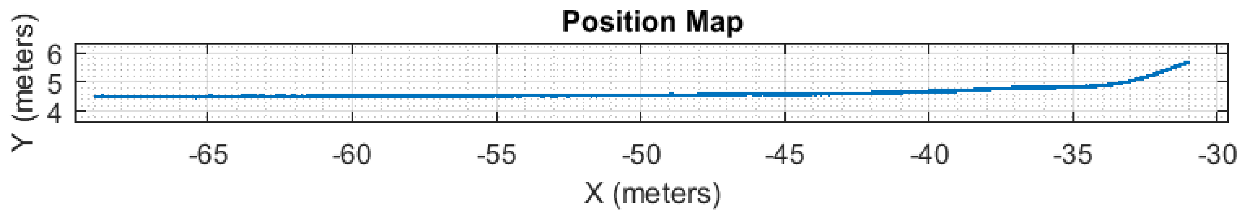


Figure 7.25 - The UAGV position still tracked quickly while load

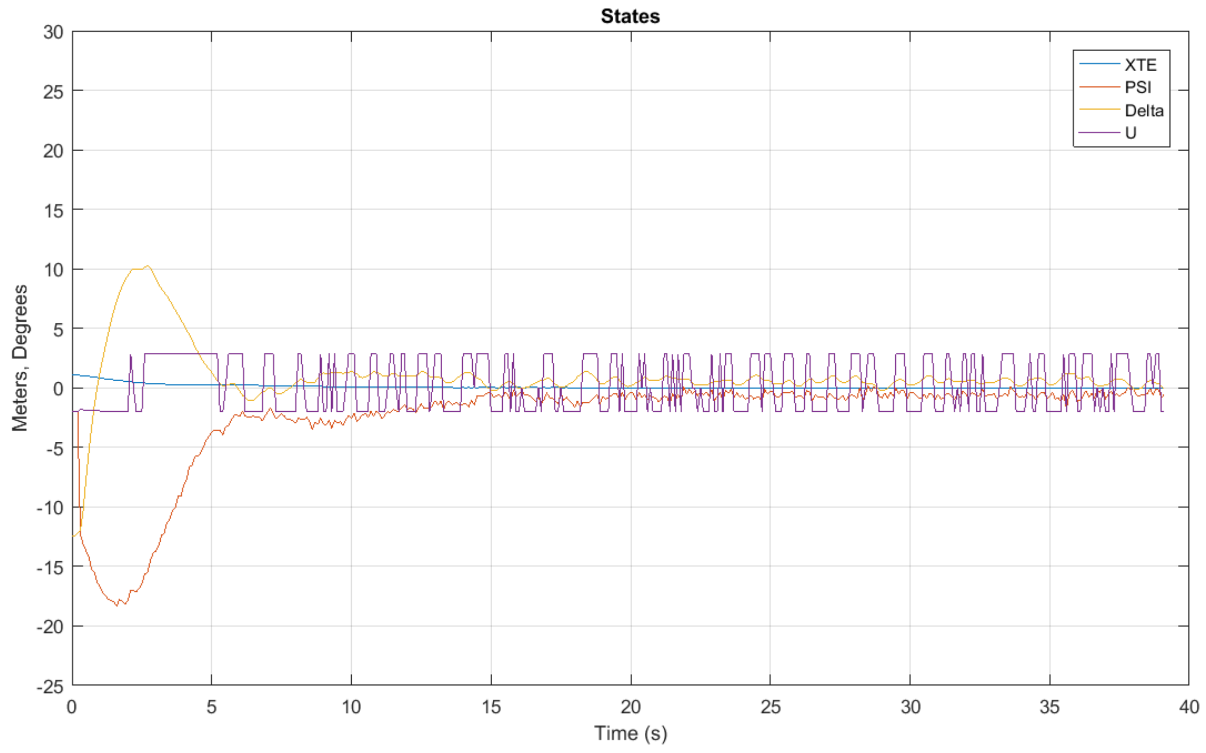


Figure 7.26 - Small errors still caused bang-bang-like control efforts because of the deadband compensation built into the steering actuator command calibration

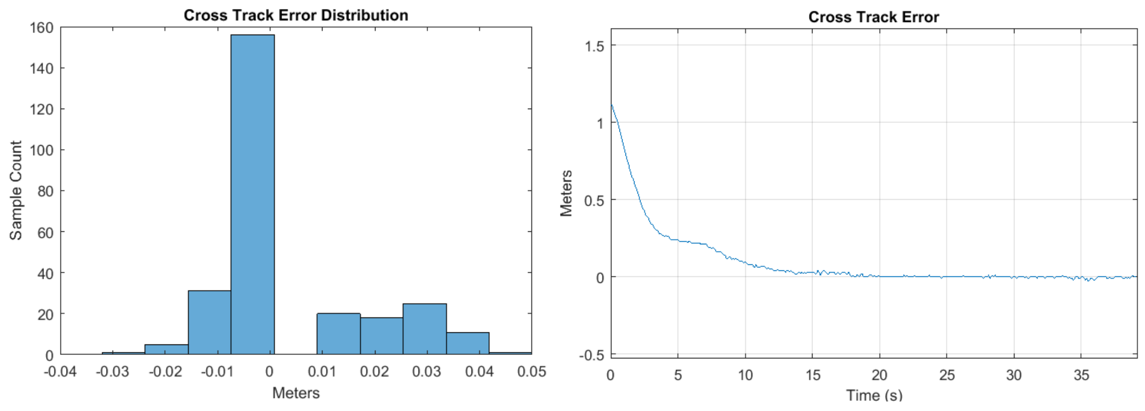


Figure 7.27 - Cross track error performance for the entire run

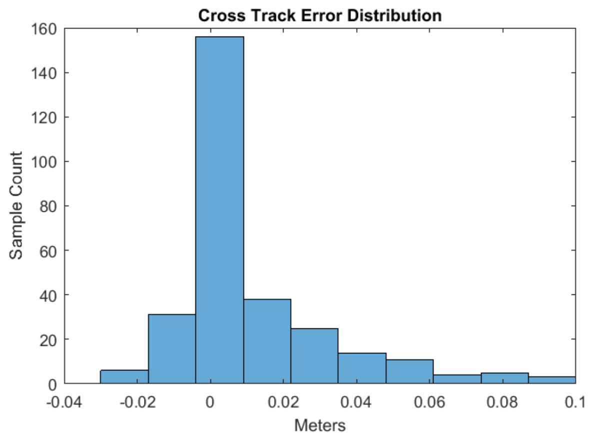


Figure 7.28 - Cross track error distribution during steady state was close to a normal distribution

7.5 Conclusions

None of the controllers developed were demonstrably optimal, only optimal as far as the system model was accurate. Performance of the controller could easily be enhanced with an active state estimator on the controller CAN node to help alleviate some of the sensor noise, and estimate system parameters. Augmenting the model in the estimator with a cross track error integrator state would improve steady state cross-track error without having to increase controller gain, risking actuator saturation. While the state evolution curves of the UAGV look similar to those of the simulated model, further scrutiny would likely reveal that most of the track tests started with initial conditions outside of the intended linearly-valid operating point of the model. Modeling from sampled state data and an assumed model structure could yield better models for controller and estimator design than what was used straight from literature. Though, this solution is not guaranteed to be fully controllable or fully observable. Better modeling of the steering actuator could also lead to improved performance, but should be considered a desperate means towards navigation precision since the steering actuators of the same manufacturer model appeared to vary in performance from each other. The demonstrated controllability of the UAGV on the digital time-triggered controller architecture is an encouraging step towards widespread use of a robust and deterministic controller communication standard for unmanned agricultural ground vehicles in research and commercial settings.

Chapter 8 Contributions and Innovations

The widespread adoption of unmanned agricultural ground vehicles in production operations both large and small will occur after two important measures are taken: The long process of defining and testing system architectures for inexpensive and modular machine design, and an arduous period of development and adoption of infrastructure standards provably appropriate for unmanned real-time networking and control of agricultural machinery components. The deeply ingrained practices of event-triggered ISOBUS implementations for increasingly autonomous heavy agricultural machinery will need to be replaced with one of determinism and safety in mind. This shift should take place or fully autonomous machines will be ever-slower to contribute to the increasingly large production processes that make or break growing agricultural economies.

8.1 Contributions

The front to back implementation of a complete controlled machine acts as a resource for researchers upon which a new time-triggered standard can be developed and tested for autonomous agricultural machinery and hitched implements. This work shows the system design considerations starting with the architecture's intent and philosophy, and finishing with a demonstration of a time-triggered network that carries system states and events that allow for modern control techniques as a common use case. The board designs and firmware descriptions serve as a reference design of a minimalistic time-triggered networking experimentation platform, and is intended to be a starting point and not a final implementation of a proposed industry standard.

8.2 Innovations

This work is a call to the agricultural machinery industry to begin the development of a controller network that supports the safety and determinism requirements for autonomous field machines. A reformation of the agricultural machinery system design standards is imminent as autonomous machinery becomes common place. Knowledge of these deterministic system design techniques has accumulated from decades of work done in the automotive and aerospace industries, but is largely unconsidered by the agricultural machinery industry. The benefits remain untapped by a sector of automation that most stands to profit. This work demonstrates the applied benefit of modern control techniques on an electric test platform that is made possible by an inexpensive and modular implementation of a time-triggered network with more determinism than a modern commercially-implemented event-triggered CAN bus. This base layer of network time-awareness is demonstrated to be relatively simple to implement on a basic microcontroller platform. This is a strong case for the continued experimentation with link layers (CAN-FD, Flexray) for the development of deterministic agricultural machinery networking standards (ISOBUS-TT).

Chapter 9 References

Ahmed, S. H., Kim, G., and Kim, D. (2013). Cyber Physical System: Architecture, applications and research challenges. In *Wireless Days (WD), 2013 IFIP* (pp. 1–5). IEEE. Retrieved from http://ieeexplore.ieee.org/abstract/document/6686528/

Albert, A. (2004). Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. *Embedded World*, *2004*, 235–252.

Almeida, L., and Fonseca, J. A. (2000). FTT-CAN: A Network-Centric Approach for CAN-Based Distributed Systems. *IFAC Proceedings Volumes*, *33*(25), 233–238. https://doi.org/http://dx.doi.org/10.1016/S1474-6670(17)39344-8

Amir, M., and Pont, M. J. (2013). Improving flexibility and fault-management in CAN-based “Shared-Clock” architectures. *Microprocessors and Microsystems*, *37*(1), 9–23. https://doi.org/10.1016/j.micpro.2012.09.011

Ataide, F. H., Carvalho, F. C., Pereira, C. E., and Wehrmeister, M. A. (2006). A comparative study of embedded protocols for safety-critical control applications. *IFAC Proceedings Volumes*, *39*(3), 87–94.

Bakker, T., van Asselt, K., Bontsema, J., Müller, J., and van Straten, G. (2010). A path following algorithm for mobile robots. *Autonomous Robots*, *29*(1), 85–97. https://doi.org/10.1007/s10514-010-9182-3

Bell, T., O’Connor, M., Jones, V. K., Rekow, A., Elkaim, G., and Parkinson, B. (1998). Realistic autofarming closed-loop tractor control over irregular paths using kinematic GPS. *The Journal of Navigation*, *51*(3), 327–335.

Biber, P., Weiss, U., Dorna, M., and Albert, A. (2012). Navigation system of the autonomous agricultural robot Bonirob. In *Workshop on Agricultural Robotics:*

Enabling Safe, Efficient, and Affordable Robots for Food Production (Collocated with IROS 2012), Vilamoura, Portugal. Retrieved from https://www.cs.cmu.edu/~mbergerm/agrobotics2012/01Biber.pdf

Billman, R. W., Pitla, S., Klopfenstein, A. A., Kuenzli, I., and Shearer, S. A. (2012, July). *Historical Perspective on the Growth in Tractor Power, Vehicle Weight, Tire Size, and Fuel Efficiency*. Presented at the ASABE Annual International Meeting, Dalla, TX.

Blackmore, Fountas, Gemtos, and Griepentrog. (2008). A specification for an autonomous crop production mechanization system. In *International Symposium on Application of Precision Agriculture for Fruits and Vegetables 824* (pp. 201–216). Retrieved from http://www.actahort.org/books/824/824_23.htm

Blackmore, Fountas, and Have. (2002). Proposed system architecture to enable behavioral control of an autonomous tractor. In *Automation Technology for Off-Road Equipment Proceedings of the 2002 Conference* (p. 13). American Society of Agricultural and Biological Engineers. Retrieved from https://elibrary.asabe.org/abstract.asp?aid=9990

Blackmore, Fountas, Tang, and Have. (2004). Systems requirements for a small autonomous tractor. Retrieved from https://ecommons.cornell.edu/handle/1813/10414

Blackmore, Have, and Fountas. (2002). Specification of behavioural requirements for an autonomous tractor. In *Automation Technology for Off-Road Equipment Proceedings of the 2002 Conference* (p. 33). American Society of Agricultural and Biological Engineers. Retrieved from https://elibrary.asabe.org/abstract.asp?aid=10067

Bosch GmbH, R. (1991). CAN Specification 2.0. Retrieved from http://www.ohio.edu/people/uijtdeha/can2spec.pdf

Brogan, W. L. (1991). *Modern Control Theory*. Prentice-Hall, Inc. Retrieved from https://www.amazon.com/Modern-Control-Theory-William-Brogan/dp/B0072VGN42?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B0072VGN42

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, *2*(1), 14–23.

Cena, G., Valenzano, A., and Vitturi, S. (2005). Advances in automotive digital communications. *Computer Standards & Interfaces*, *27*(6), 665–678. https://doi.org/10.1016/j.csi.2004.12.005

Chan, K. C., Katupitiya, J., Hanrahan, J. W., Jackson, C. J., and Rose, D. (2014). An emergency communication system for an agricultural autonomous vehicle. In *Proc. Int. Conf. Intelligent Agriculture, IPCBEE* (Vol. 63, pp. 76–82). Retrieved from http://www.ipcbee.com/vol63/013-ICOIA2014-IA0030.pdf

Darr. (2004). Development and evaluation of a controller area network based autonomous vehicle. Retrieved from http://uknowledge.uky.edu/gradschool_theses/192/

Darr. (2012). CAN bus technology enables advanced machinery management. *Resource Magazine*, *19*(5), 10–11.

Darr, Stombaugh, T. S., and Shearer, S. A. (2005). Controller area network based distributed control for autonomous vehicles. *Transactions of the ASAE, 48*(2), 479–490.

Di Natale, M., Zeng, H., Giusto, P., and Ghosal, A. (2012). *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*. Springer New York. Retrieved from https://books.google.com/books?id=AJn-vSOL_3EC

Dong, X., Vuran, M. C., and Irmak, S. (2013). Autonomous precision agriculture through integration of wireless underground sensor networks with center pivot irrigation systems. *Ad Hoc Networks*, *11*(7), 1975–1987. https://doi.org/10.1016/j.adhoc.2012.06.012

Eaton, R., Katupitiya, J., Cole, A., and Meyer, C. (2005). Architecture of an automated agricultural tractor: Hardware, software and control systems. In *Proceedings of the 2005 IFAC World Congress*. Retrieved from http://www.nt.ntnu.no/users/skoge/prost/proceedings/ifac2005/Fullpapers/04714.pdf

Ehrl, M., and Auernhammer, H. (2007). X-By-Wire via ISOBUS Communication Network. *Agricultural Engineering International: CIGR Journal*, *9*(Manuscript ATOE 07 002).

Elkaim, G., O’Connor, M., Bell, T., Parkinson, B. W., and others. (1997). System identification and robust control of a farm vehicle using CDGPS. In

PROCEEDINGS OF ION GPS (Vol. 10, pp. 1415–1426). INSTITUTE OF NAVIGATION. Retrieved from https://users.soe.ucsc.edu/~elkaim/Documents/sysid_robust_farm.pdf

Erbach, D. C., Wilkins, D. E., and Lovely, W. G. (1972). Relationships between furrow opener, corn plant spacing, and yield. *Agronomy Journal*, *64*(5), 702–704.

FAO's Director-General on How to Feed the World in 2050. (2009). *Population and Development Review*, *35*(4), 837–839. https://doi.org/10.1111/j.1728-4457.2009.00312.x

Fountas, Blackmore, Vougioukas, Tang, L., Sørensen, C. G., and Jørgensen, R. (2007). Decomposition of agricultural tasks into robotic behaviours. *Agricultural Engineering International: CIGR Journal*. Retrieved from http://www.cigrjournal.org/index.php/Ejounral/article/download/901/895

Franklin, G. F. et al. (1998). *Digital Control of Dynamic Systems*. Addison-wesley. Retrieved from https://www.amazon.com/Digital-Control-Dynamic-Systems-Franklin/dp/8177588281?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=8177588281

Fuster, S., Rodríguez, F., and Bonastre, A. (2005). Software-based EDF message scheduling on CAN networks. In *Embedded Software and Systems, 2005. Second International Conference on* (p. 6–pp). IEEE. Retrieved from http://ieeexplore.ieee.org/abstract/document/1609911/

Godoy, E. P., Tabile, R. A., Pereira, R. R., Tangerino, G. T., Porto, A. J., and Inamasu, R. Y. (2010). Design and implementation of an electronic architecture for an agricultural mobile robot. *Revista Brasileira de Engenharia Agrícola E Ambiental*, *14*(11), 1240–1247.

Hehenberger, P., Vogel-Heuser, B., Bradley, D., Eynard, B., Tomiyama, T., and Achiche, S. (2016). Design, modelling, simulation and integration of cyber physical systems: Methods and applications. *Computers in Industry*, *82*, 273–289. https://doi.org/10.1016/j.compind.2016.05.006

Herlitzius, T. (2017). Automation and Robotics - The Trend Towards Cyber Physical Systems in Agriculture Business. In *SAE Technical Paper*. SAE International.

Hofstee, J. W., and Goense, D. (1999). Simulation of a Controller Area Network-based Tractor — Implement Data Bus according to ISO 11783. *Journal of Agricultural Engineering Research*, *73*(4), 383–394. https://doi.org/http://dx.doi.org/10.1006/jaer.1999.0432

Hu, L., Xie, N., Kuang, Z., and Zhao, K. (2012). Review of Cyber-Physical System Architecture (pp. 25–30). IEEE. https://doi.org/10.1109/ISORCW.2012.15

Jacobs, G., Schlüter, F., Schröter, J., Feldermann, A., and Strassburger, F. (2017). Cyber-Physical Systems for Agricultural and Construction Machinery—Current Applications and Future Potential. In S. Jeschke, C. Brecher, H. Song, and D. B. Rawat (Eds.), *Industrial Internet of Things* (pp. 617–645). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-42559-7_26

Jadlovska, A., Jadlovska, S., and Voščček, D. (2016). Cyber-Physical System Implementation into the Distributed Control System. *IFAC-PapersOnLine*, *49*, 31–36.

Jensen, K., Larsen, M., Nielsen, S., Larsen, L., Olsen, K., and Jørgensen, R. (2014). Towards an Open Software Platform for Field Robots in Precision Agriculture. *Robotics*, *3*(2), 207–234. https://doi.org/10.3390/robotics3020207

Juanole, G., Mouney, G., Calmettes, C., and Peca, M. (2005). FUNDAMENTAL CONSIDERATIONS FOR IMPLEMENTING CONTROL SYSTEMS ON A CAN NETWORK. *IFAC Proceedings Volumes*, *38*(2), 79–86. https://doi.org/http://dx.doi.org/10.3182/20051114-2-MX-3901.00012

Katupitiya, J., Eaton, R., and Yaqub, T. (2007). Systems engineering approach to agricultural automation: new developments. In *Systems Conference, 2007 1st Annual IEEE* (pp. 1–7). IEEE. Retrieved from http://ieeexplore.ieee.org/abstract/document/4258893/

Klopfenstein, A. A. (2016). *An Empirical Model for Estimating Corn Yield Loss from Compaction Events with Tires vs. Tracks High Axle Loads*. The Ohio State University. Retrieved from http://rave.ohiolink.edu/etdc/view?acc_num=osu1461316924

Kopetz, H. (1995). A communication infrastructure for a fault-tolerant distributed real-time system. *Control Engineering Practice*, *3*(8), 1139–1146.

Kopetz, H. (2011). *Real-Time Systems*. Boston, MA: Springer US. https://doi.org/10.1007/978-1-4419-8237-7

Krall, J. M., Esechie, H. A., Raney, R. J., Clark, S., TenEyck, G., Lundquist, M., Humburg, N. E., Axthelm, L. S., Dayton, A. D., and Vanderlip, R. L. (1977). Influence of within-row variability in plant spacing on corn grain yield. *Agronomy Journal*, *69*(5), 797–799.

Lari, V., Dehbashi, M., Miremadi, S. G., and Amiri, M. (2007). Evaluation of babbling idiot failures in FlexRay-based networkes. *IFAC Proceedings Volumes*, *40*(22), 399–406.

Lauer, J. G., and Rankin, M. (2004). Corn response to within row plant spacing variation. *Agronomy Journal*, *96*(5), 1464–1468.

Lawrenz, W. (1997). *Can System Engineering: From Theory to Practical Applications* (1st ed.). Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Lee, J., Bagheri, B., and Kao, H.-A. (2015). A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. *Manufacturing Letters*, *3*, 18–23. https://doi.org/10.1016/j.mfglet.2014.12.001

Leen, G., and Heffernan, D. (2002). TTCAN: a new time-triggered controller area network. *Microprocessors and Microsystems*, *26*(2), 77–94.

Martin, K., Raun, W., and Solie, J. (2012). By-plant prediction of corn grain yield using optical sensor readings and measured plant height. *Journal of Plant Nutrition*, *35*(9), 1429–1439. https://doi.org/10.1080/01904167.2012.684133

Marx, S. E., Luck, J. D., Hoy, R. M., Pitla, S., Blankenship, E. E., and Darr, M. J. (2015). Validation of machine CAN bus J1939 fuel rate accuracy using Nebraska Tractor Test Laboratory fuel rate data. *Computers and Electronics in Agriculture*, *118*, 179–185. https://doi.org/10.1016/j.compag.2015.08.032

Marx, S. E., Luck, J. D., Pitla, S., and Hoy, R. M. (2016). Comparing various hardware/software solutions and conversion methods for Controller Area Network (CAN) bus data collection. *Computers and Electronics in Agriculture*, *128*, 141–148. https://doi.org/10.1016/j.compag.2016.09.001

McKee, K. D., Formwalt, C. W., Benneweis, R. K., and Stone, M. L. (1999). ISO 11783: An Electronic Communications Protocol for Agricultural Equipment. Retrieved from https://elibrary.asabe.org/data/pdf/6/ddp2002/lecture23.pdf

Mead, R. (1966). A relationship between individual plant-spacing and yield. *Annals of Botany*, *30*(2), 301–309.

Meschi, A., Di Natale, M., and Spuri, M. (1996). Earliest deadline message scheduling with limited priority inversion. In *Parallel and Distributed Real-Time Systems, 1996. Proceedings of the 4th International Workshop on* (pp. 87–94). IEEE. Retrieved from http://ieeexplore.ieee.org/abstract/document/557464/

Moran, M., Inoue, Y., and Barnes, E. (1997). Opportunities and limitations for image-based remote sensing in precision crop mangement. *Remote Sensing of Environment*, *61*, 319–346.

Nolte, T., Hansson, H., and Norstrom, C. (2002). Minimizing CAN response-time jitter by message manipulation. In *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE* (pp. 197–206). IEEE. Retrieved from http://ieeexplore.ieee.org/abstract/document/1137394/

O’Conner, M. L. (1998). *Carrier-phase differential GPS for automatic control of land vehicles*. Stanford University. Retrieved from http://web.stanford.edu/group/scpnt/gpslab/pubs/theses/MichaelOConnorThesis98.pdf

Oksanen, T., Öhman, M., Miettinen, M., and Visala, A. (2005). ISO 11783–Standard and its implementation. *IFAC Proceedings Volumes*, *38*(1), 69–74.

Pedreiras, P., and Almeida, L. (2002). EDF message scheduling on controller area network. *Computing & Control Engineering Journal*, *13*(4), 163–170.

Pitla, S. (2012). *Development of Control Architectures for Multi-robot Agricultural Field Production Systems*. University of Kentucky, Lexington, KY, USA.

Pitla, S., Lin, N., Shearer, S. A., and Luck, J. D. (2014). Use of controller area network (CAN) data to determine field efficiencies of agricultural machinery. *Applied Engineering in Agriculture*, *30*(6), 829–839.

Pitla, S., Luck, J. D., and Shearer, S. A. (2008). Automatic Guidance System Development Using Low Cost Ranging Devices. In *2008 Providence, Rhode Island, June 29–July 2, 2008* (p. 1). American Society of Agricultural and Biological Engineers. Retrieved from https://elibrary.asabe.org/abstract.asp?aid=36211

Pitla, S., Luck, J. D., Werner, J., Lin, N., and Shearer, S. A. (2016). In-field fuel use and load states of agricultural field machinery. *Computers and Electronics in Agriculture*, *121*, 290–300. https://doi.org/10.1016/j.compag.2015.12.023

Rad, C.-R., Hancu, O., Takacs, I.-A., and Olteanu, G. (2015). Smart Monitoring of Potato Crop: A Cyber-Physical System Architecture Model in the Field of Precision Agriculture. *Agriculture and Agricultural Science Procedia*, *6*, 73–79. https://doi.org/10.1016/j.aaspro.2015.08.041

Rodríguez-Navas, G., and Proenza, J. (2003). Analyzing atomic broadcast in TTCAN networks. *IFAC Proceedings Volumes*, *36*(13), 147–150. https://doi.org/http://dx.doi.org/10.1016/S1474-6670(17)32477-1

Saito, M., Tamaki, K., Nishiwaki, K., Nagasaka, Y., and Motobayashi, K. (2013). Development of Robot Combine Harvester for Beans using CAN Bus Network. *IFAC Proceedings Volumes*, *46*(18), 148–153. https://doi.org/10.3182/20130828-2-SF-3019.00058

Schmidt, K., and Schmidt, E. G. (2007). Systematic Message Schedule Construction for Time-Triggered CAN. *IEEE Transactions on Vehicular Technology*, *56*(6), 3431–3441. https://doi.org/10.1109/TVT.2007.906413

Shi, Y., Wang, N., Taylor, R. K., Raun, W. R., and Hardin, J. A. (2013). Automatic corn plant location and spacing measurement using laser line-scan technique. *Precision Agriculture*, *14*(5), 478–494. https://doi.org/10.1007/s11119-013-9311-z

Short, M., Sheikh, I., and Rizvi, S. A. I. (2016). A transmission window technique for CAN networks. *Journal of Systems Architecture*, *69*, 15–28. https://doi.org/10.1016/j.sysarc.2016.07.001

Shoukry, Y., Shokry, H., and Hammad, S. (2011). Distributed Dynamic Scheduling of Controller Area Network Messages for Networked Embedded Control Systems. *IFAC Proceedings Volumes*, *44*(1), 1959–1964. https://doi.org/10.3182/20110828-6-IT-1002.03620

Silva, A., and Vuran, M. (2010). (CPS)^ 2: integration of center pivot systems with wireless underground sensor networks for autonomous precision agriculture. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems* (pp. 79–88). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=1795206

Silva, V., Fonseca, J. A., Nunes, U., and Maia, R. (2006). Communications Requirements for Autonomous Mobile Robots: Analysis and Examples. In *Fieldbus Systems and Their Applications 2005* (pp. 91–98). Elsevier. https://doi.org/10.1016/B978-008045364-4/50053-8

Soman, P., Jayachandran, R., and Bidinger, F. R. (1987). Uneven variation in plant-to-plant spacing in pearl millet. *Agronomy Journal*, *79*(5), 891–895.

Sowers, K., Wang, N., Taylor, R., Raun, W., and Hardin, J. (1994). Nitrogen use efficiency of split nitrogen applications in soft white winter wheat. *Agronomy Journal*, *86*, 942–948.

Speckmann, H., and Jahns, G. (1999). Development and application of an agricultural BUS for data transfer. *Computers and Electronics in Agriculture*, *23*(3), 219–237.

Tenruh, M. (2011). Message scheduling with reduced matrix cycle and evenly distributed sparse allocation for time-triggered CAN. *Journal of Network and Computer Applications*, *34*(4), 1240–1251. https://doi.org/10.1016/j.jnca.2011.01.009

Thompson, H. A., Benítez-Pérez, H., Lee, D., Ramos-Hernandez, D. N., Fleming, P. J., and Legge, C. G. (1999). A CANbus-based safety-critical distributed aeroengine control systems architecture demonstrator. *Microprocessors and Microsystems*, *23*(6), 345–355.

Tillett, N. D., Hague, T., and Marchant, J. A. (1998). A robotic system for plant-scale husbandry. *Journal of Agricultural Engineering Research*, *69*(2), 169–178.

U.N. Food and Agriculture Organization. (2009, October 12). How_to_Feed_the_World_in_2050.pdf. Retrieved from http://www.fao.org/fileadmin/templates/wsfs/docs/expert_paper/How_to_Feed_the_World_in_2050.pdf

Wei, J., Zhang, N., Wang, N., Oard, D., Stoll, Q., Lenhert, D., Neilsen, M., Mizuno, M., and Sing, G. (1998). Design of an embedded weed-control system using Controller Area Network (CAN). In *2001 ASAE Annual Meeting* (p. 1). American Society of Agricultural and Biological Engineers. Retrieved from https://elibrary.asabe.org/azdez.asp?AID=7414&T=2

Weidong, L., Li'e, G., Yilin, D., and Jianning, X. (2006). Communication scheduling for CAN bus autonomous underwater vehicles. In *Mechatronics and Automation, Proceedings of the 2006 IEEE International Conference on* (pp. 379–383). IEEE. Retrieved from http://ieeexplore.ieee.org/abstract/document/4026112/

Will, J. D., Stombaugh, T. S., Benson, E. R., Noguchi, N., and Reid, J. F. (1998). Development of a flexible platform for agricultural automatic guidance research. *ASAE Paper*, *983202*. Retrieved from https://pdfs.semanticscholar.org/6ed2/c411c298a0f6ae5a125f06722020b803655c.pdf

Wise, T. A. (2013). Can we feed the world in 2050. *A Scoping Paper to Assess the Evidence. Global Development and Environment Institute Working Paper*, (13-04). Retrieved from http://ase.tufts.edu/gdae/Pubs/wp/13-04WiseFeedWorld2050.pdf

Xia, J., Zhang, C., Bai, R., and Xue, L. (2013). Real-time and reliability analysis of time-triggered CAN-bus. *Chinese Journal of Aeronautics*, *26*(1), 171–178. https://doi.org/10.1016/j.cja.2012.12.017

Zhang, Q., and Pierce, F. J. (2016). *Agricultural Automation: Fundamentals and Practices*. CRC Press. Retrieved from https://www.amazon.com/Agricultural-Automation-Fundamentals-Qin-Zhang-ebook/dp/B00CPCKDZI?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B00CPCKDZI

Chapter 10 Appendix

10.1 Schematic Designs

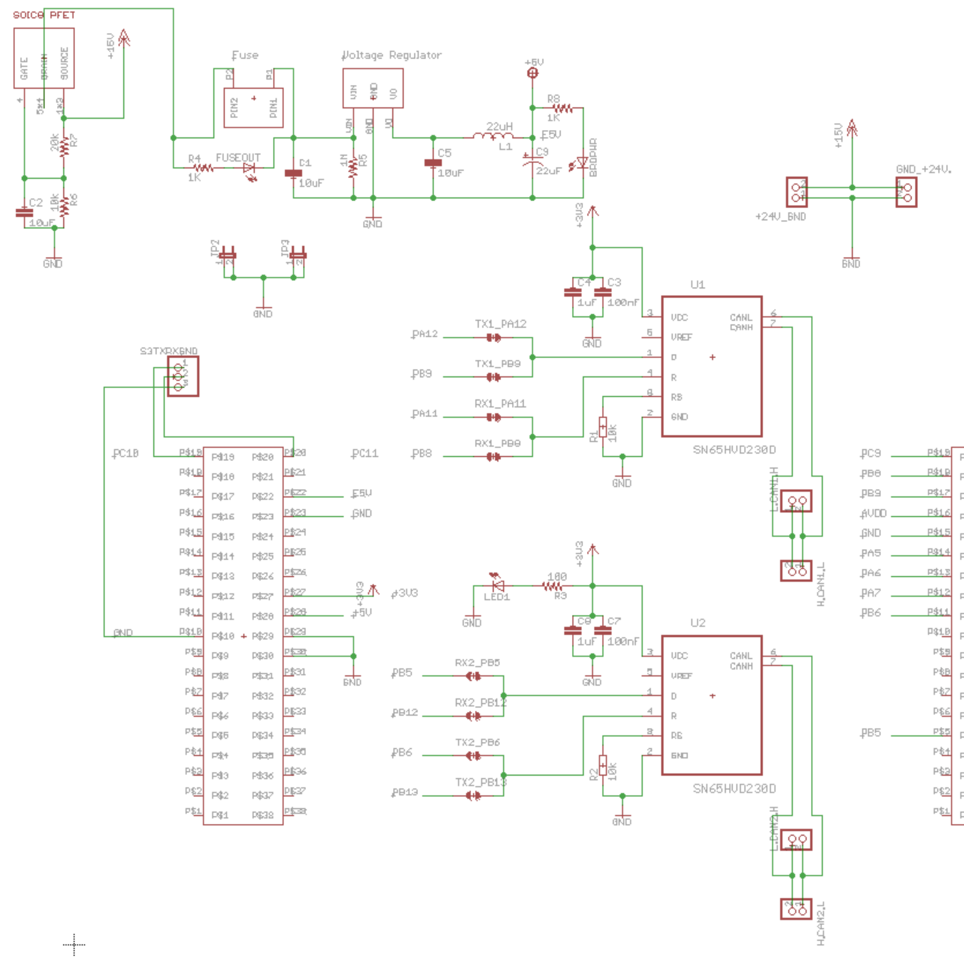


Figure 10.1 - CAN sled for F446

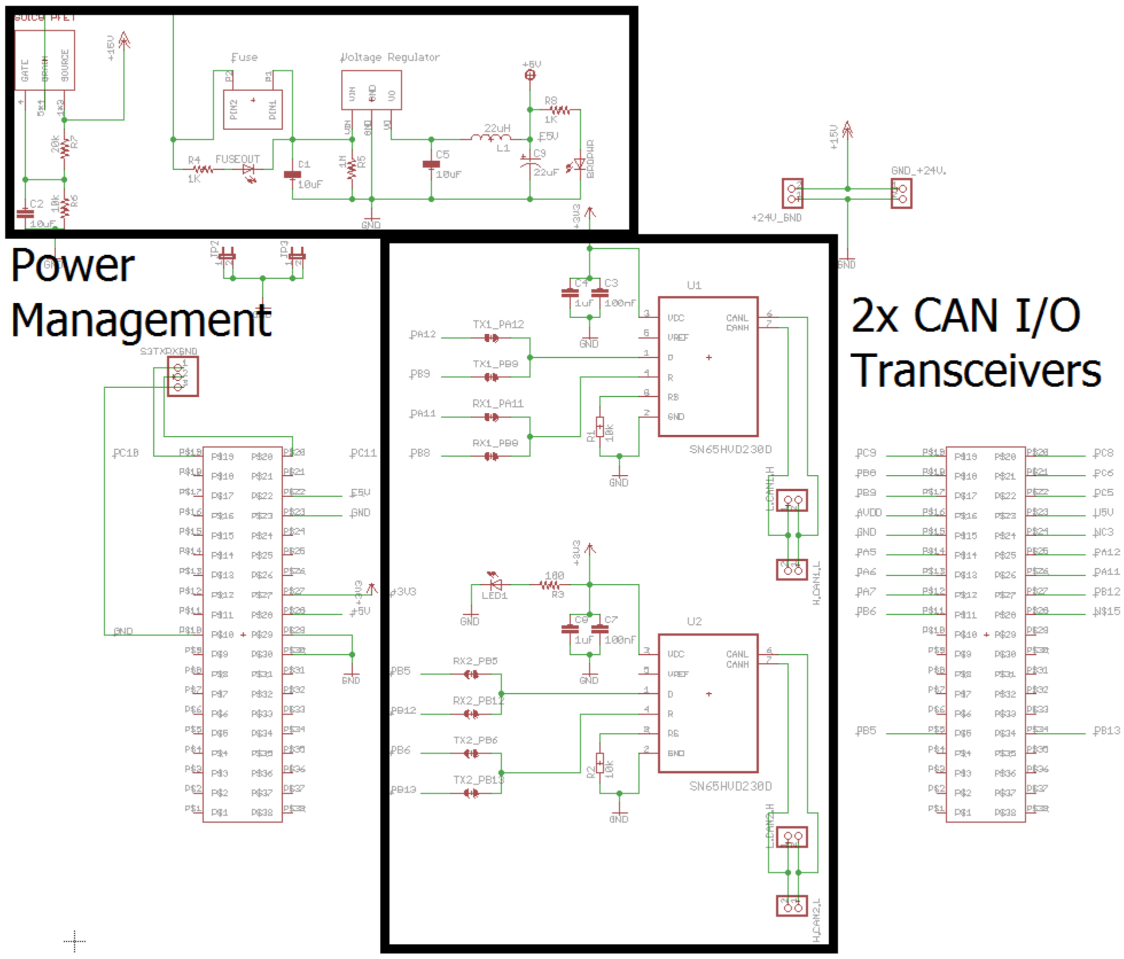


Figure 10.2 - Simple sub-system description of CAN interface

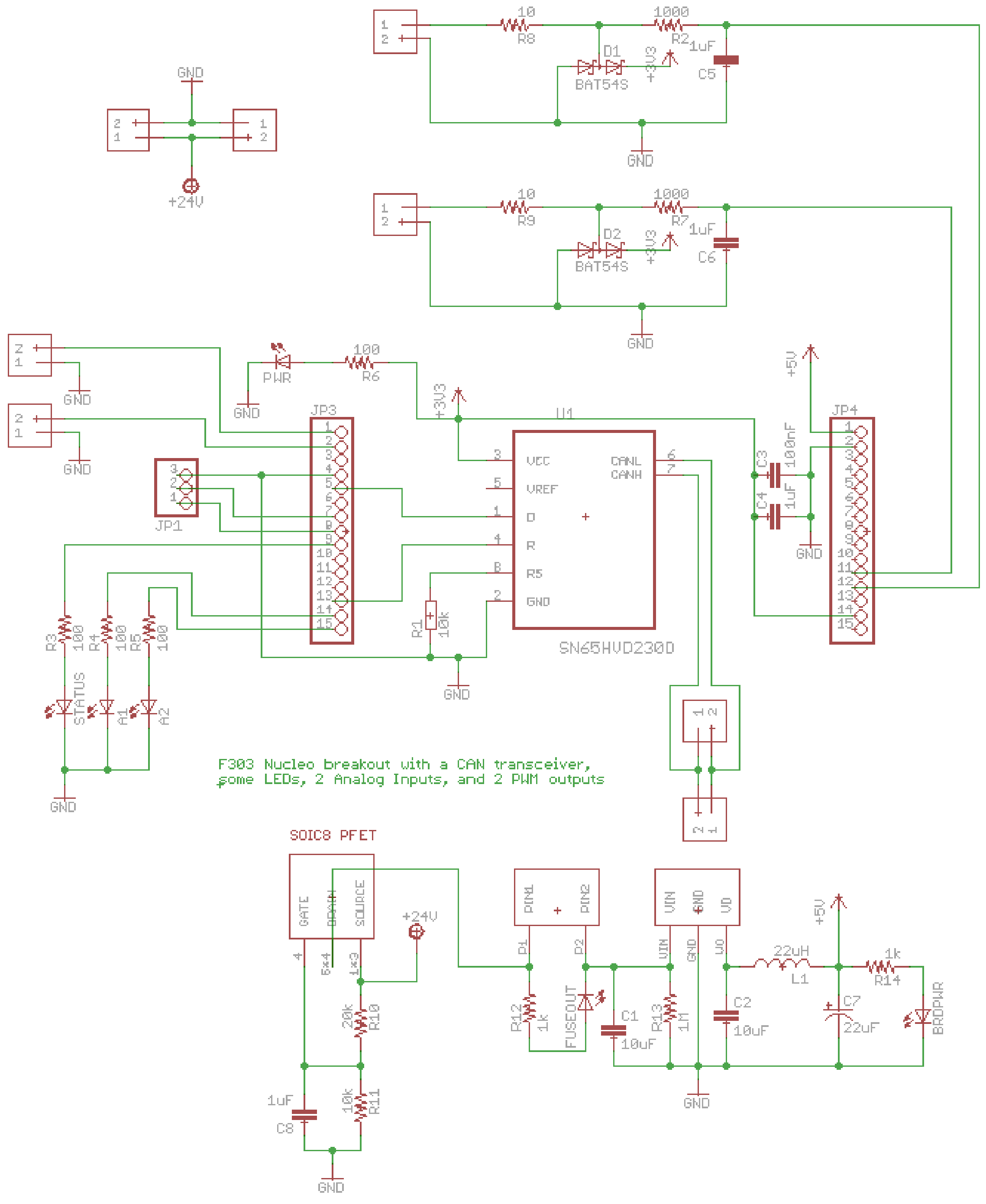


Figure 10.3 - F303 CAN sled schematic

10.2 Board Designs

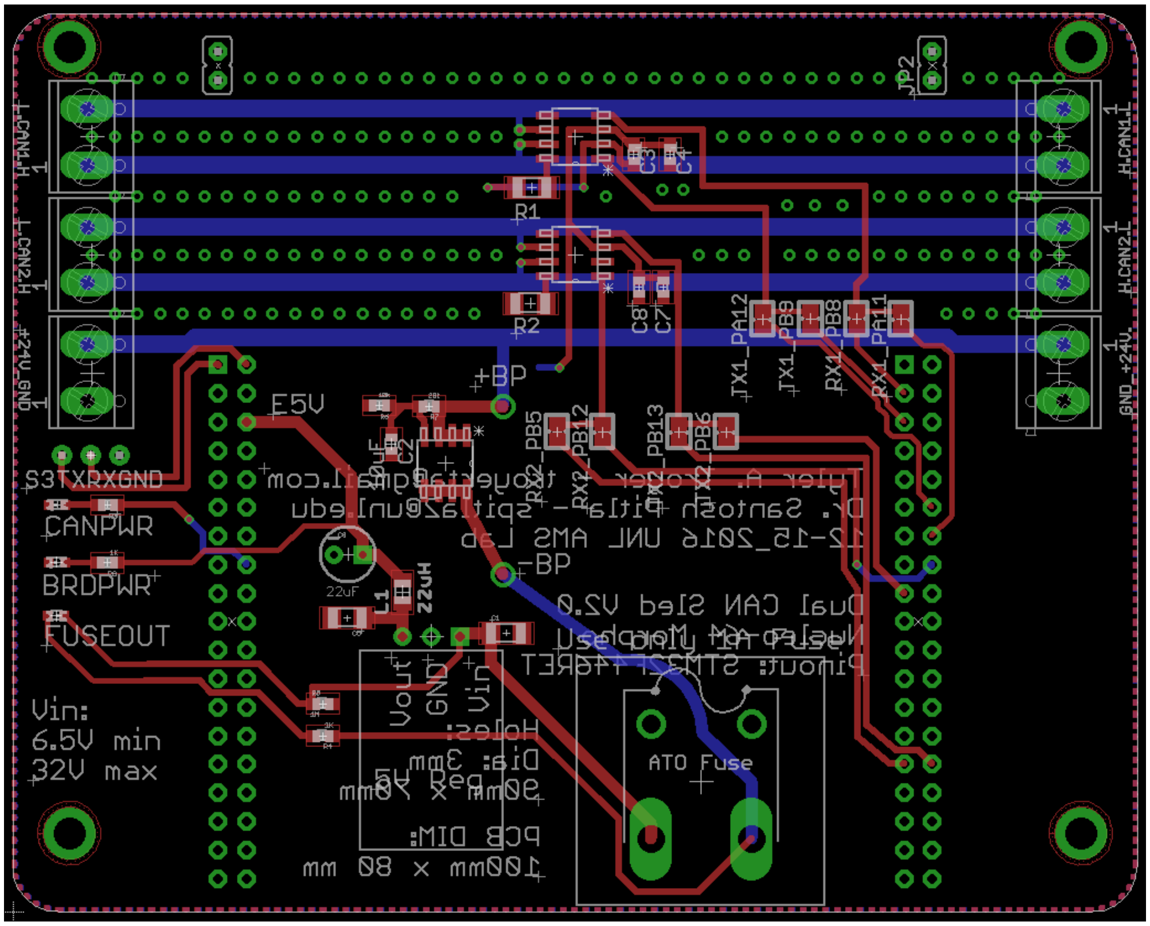


Figure 10.4 - Board layout for F446 CAN sled

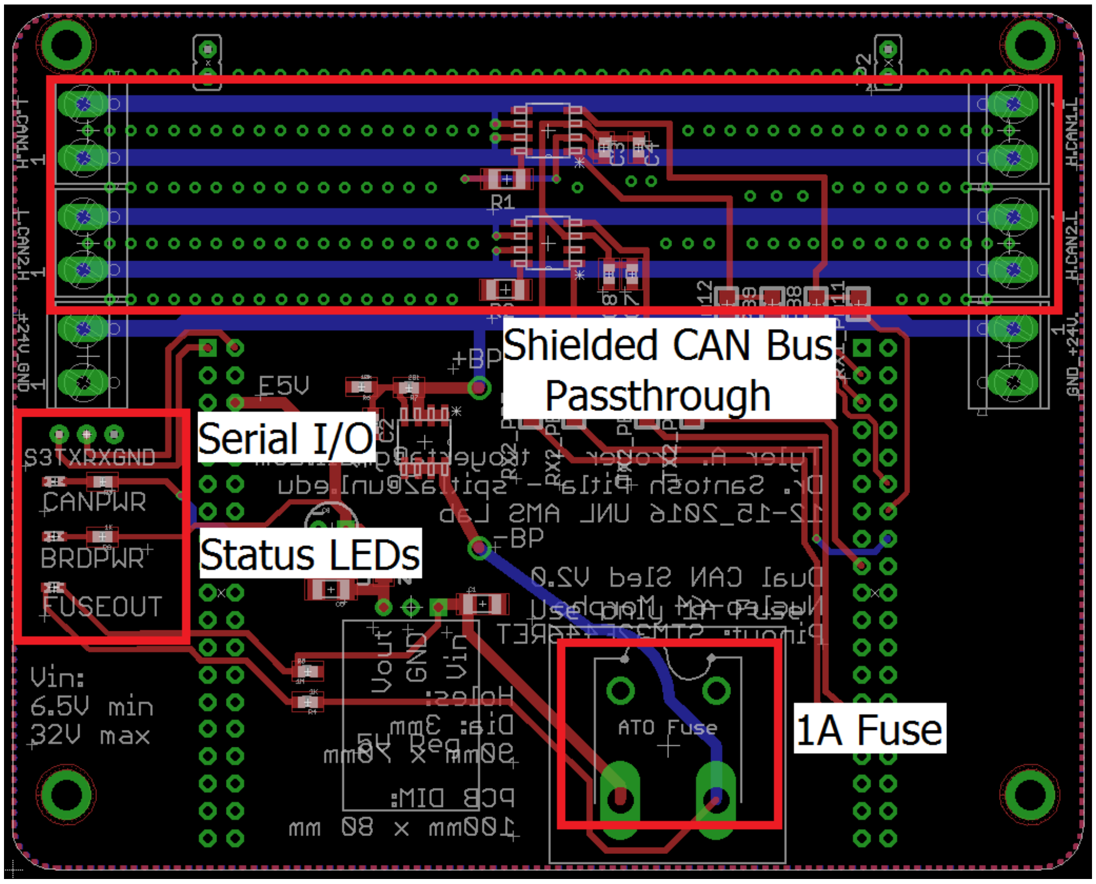


Figure 10.5 – Board component description of F446 CAN interface board

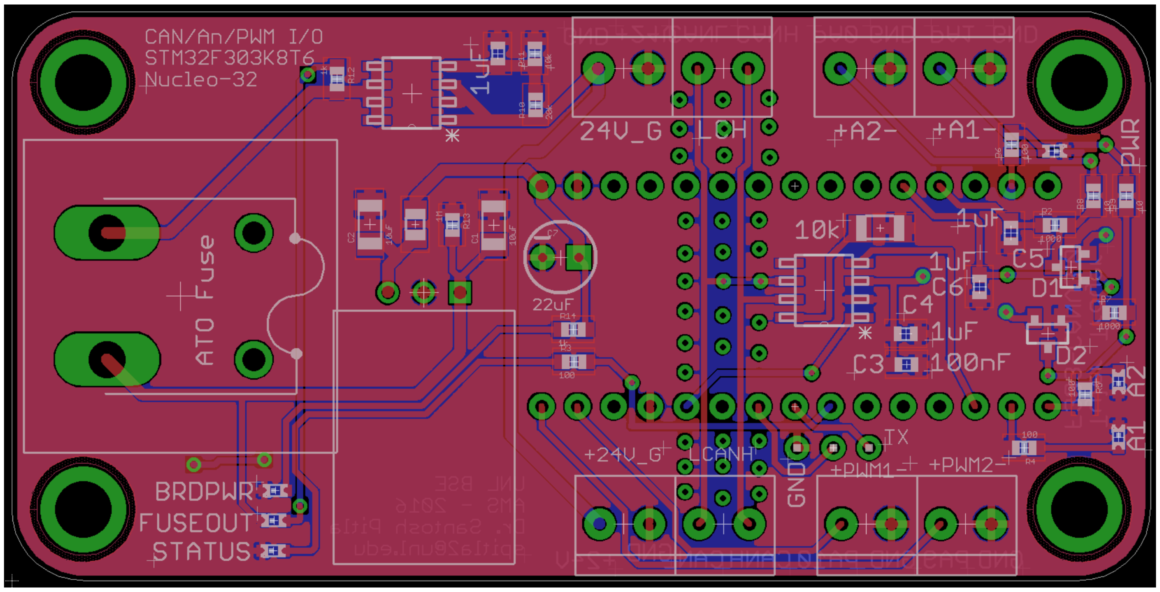


Figure 10.6 - F303 CAN sled board layout