A Wireless Rover With External Video Feedback Control

7 June 2004
Daniel J. Gibson
Joanne Mikkelson

A design is presented for a wireless rover with a remote guidance system based on video feedback from an external camera. The design is modular at several nested levels and includes multiple FSMs. It was implemented using a combination of Verilog compiled to FPGAs and a wireless system with a microprocessor on the mobile side. Simulation at all levels from the smallest units to the two major subsystems resulted in predictable behavior of the hardware, at least to the degree that the hardware specifications were known. Better results were obtained by supplying additional detail in the Verilog code, rather than relying on the compiler to make the intended interpretations. The peripheral hardware presented surprising difficulties, notably noise and nonuniformity in the analog video signal and insufficient mechanical power in the wheel motors to drive the rover after adding custom hardware.

Contents

List of Figures

1 Overview

This project comprises a wireless remote controlled rover, and a fixed position (e.g. tripod mounted) video camera that provides visual feedback. The goal is to enable the user to point to a location on a video screen showing a large area of lab floor containing nothing but the rover, and instruct the rover to go to the location pointed to. The rover is made from a remote-controlled toy car, with both ends of its controller replaced. The car has a dark color to make it easily distinguishable from the white lab floor in the video image. Its direction of movement is calculated by taking the difference in its location at two points in time.

2 Video Analysis System (By Daniel J. Gibson)

2.1 Overview

The Video Analysis System's task is to convert an NTSC analog video signal to a digital representation, detect the position of the car in each frame, and generate error signals for output to the Car Control System. The error signals specify whether the car needs to be turned and if so in which direction, and when the car has arrived at the target position.

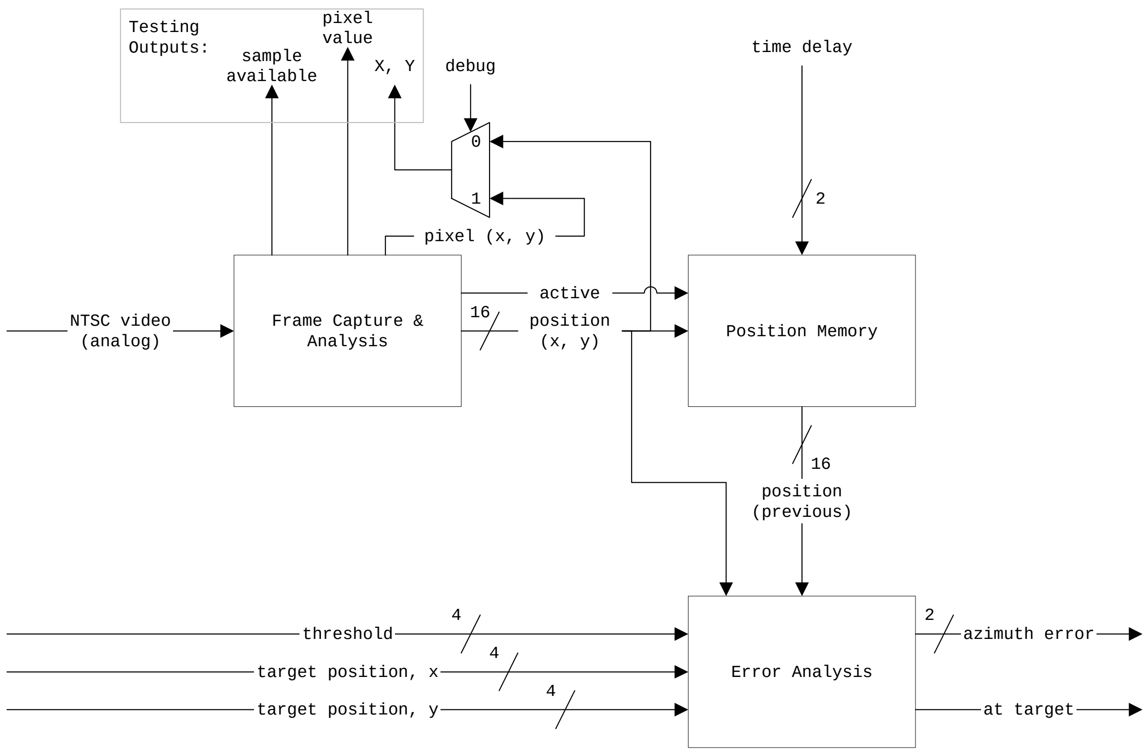


Figure 1: Video Analysis System Block Diagram

The overall structure of the system is shown in Figure 1. The Frame Capture & Analysis module performs the video conversion and position detection. Testing outputs are provided to give access to internal signals for debugging and for demonstration of functionality separately from the Car Control System. A mux makes it possible to examine two different 16-bit signals without consuming more than 16 output pins on the FPGA.

The Position Memory module is simply a 16-bit-wide four-position shift register that shifts at the beginning of each new frame. It accepts the 16-bit position signal produced by the Frame Capture & Analysis module, and it outputs a delayed 16-bit position signal. The delay is user-selectable to be one, two, three, or four frames to enable the system to work with different car speeds. It will not be discussed further here.

The Error Analysis module accepts the current and delayed car position coordinates, and user-specified 4-bit values for target position X and Y coordinates and a threshold that determines the precision with which the car's motion is controlled. It outputs the azimuth error of the car's velocity as a 2-bit value that indicates whether and in which direction the car must be turned in order to close on its target, and a 1-bit "at target" signal that indicates when the car should be stopped. The user-specified threshold affects both error outputs.

The Video Analysis System was implemented using an Altera Flex 10K70 FPGA and two separate analog chips for video conversion. System logic was specified in Verilog and compiled to the FPGA using Altera's Max+Plus II software.

2.2 Modules

2.2.1 Frame Capture & Analysis

Video Conversion The purpose of the Video Conversion module is to convert an NTSC analog video input into a 1-bit serial format together with an 8-bit X coordinate and an 8-bit Y coordinate for each pixel. High resolution is not required, and it so happens that when using a 10 MHz clock, the NTSC video format conveniently converts to 240 rows of 256 columns. This requires that the video signal be sampled once every other clock cycle, and only one of the two interlaced fields need be used. The present design is based on well-known methods that utilize a simple asynchronous 1-bit DAC and a Video Sampler FSM, and is described further in the Appendices.

Position Analyzer The Position Analyzer accepts the 1-bit video stream produced by the Video Conversion module, and calculates the X and Y coordinates of the car in each frame of video.

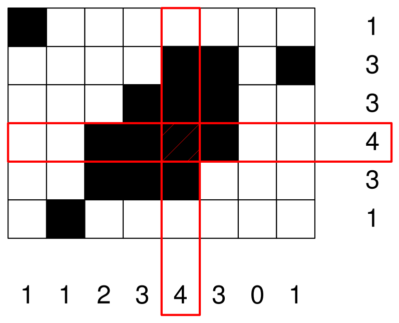


Figure 2: Determination of Position from Pixel Count Histograms

Position Analyzer Strategy The strategy is to construct histograms of the number of black pixels on each row and in each column of the video frame. While the histograms are being accumulated from the incoming video stream, a record is kept of the row or column number that has the largest count. Ties are resolved by using the smallest row or column number. When the entire video frame is finished, the car position is identified as the smallest row and column numbers with the maximum numbers of black pixels. This strategy is illustrated in Figure 2, where the highest numbers of black pixels were found on the fourth row and fifth column. It can be seen that this method provides good immunity from random noise in the image.

The Position Analyzer thus comprises two identical Counts Accumulator modules for computing the histograms, plus an FSM to control the process of clearing the accumulated histograms in between frames.

Counts Accumulator The purpose of the Counts Accumulator module is to calculate a pixel count histogram along one dimension of the image. The module has the following inputs: pixel coordinate (8 bits), pixel value (1 bit), accumulate enable (1 bit), sample available (1 bit), and clear (1 bit). It produces one output, the 8-bit coordinate where the largest number of black pixels occurs. During the active video stream, the clear input is held low, the accumulate enable input is held high, and a two-cycle process updates the histogram. On the first cycle, `sample_avail` goes high and the pixel address may change (which it in fact does in the case of the X coordinate Counts Accumulator). Also, a new value for the accumulated count is computed from the old value and the current pixel value. On the next clock edge, the new accumulated count value is stored in a working register and written from that register to permanent storage.

Because Max+Plus II does not implement Verilog arrays, it was necessary to base the permanent storage on RAM implemented with Altera's LPM_RAM_DQ library module. The unregistered version was chosen to allow the designer to control the process timing at single clock cycle resolution. This choice was motivated by the fact that the video data rate was one pixel every two clock cycles, making a three-cycle write process impossible.

Given the temporal structure of the incoming video stream with pixel coordinates supplied, the histogram calculation is sufficiently straightforward not to require an FSM. However, the accumulators that hold the histogram counts must be re-zeroed in between frames, and an FSM was designed for that purpose.

Position Analyzer FSM Figure 3 shows the state transition diagram for the Position Analyzer FSM. The variable names illustrate a helpful design practice, which is to make the distinction between registered values and combinational values clear through the variable names, where the suffixes "_int" and "_reg" are used to denote the combinational (or "internal") value and the registered value respectively. This naming convention makes it immediately clear from looking at the state transition diagram when a new value becomes available relative to when it changes in the state diagram, i.e. the registered value gets updated one clock cycle after entry to the state that changes the combinational value. In the case of `sample_avail`, this is an unregistered value and so does not require any suffix, and any changes in its value are effective immediately.

On `reset`, the FSM clears the accumulators by counting through all the possible accumulator addresses and clearing each one. The signal `passthrough_reg` controls a mux that selects between pixel coordinates and "sample available" values generated internally by the FSM, or values supplied externally by the Video Sampler module. It is set to 0 at the start of the clearing process, thus selecting the internally generated signals. Each location is cleared in both Counts Accumulators by setting the "sample available" and "clear" inputs both high on both Counts Accumulators. When the last location has been cleared, "sample available" goes low, and the FSM either goes directly to the `WAIT_ACTIVE` state to wait for the start of the next frame, or if there is currently active video coming in from the Video Sampler, then the FSM waits for the current frame to finish and then goes to `WAIT_ACTIVE`. The Position Analyzer is now cleared and ready to analyze the next frame. When that next frame finishes, and the position for that frame has been calculated, the clear process begins again to prepare for analysis of the following frame.

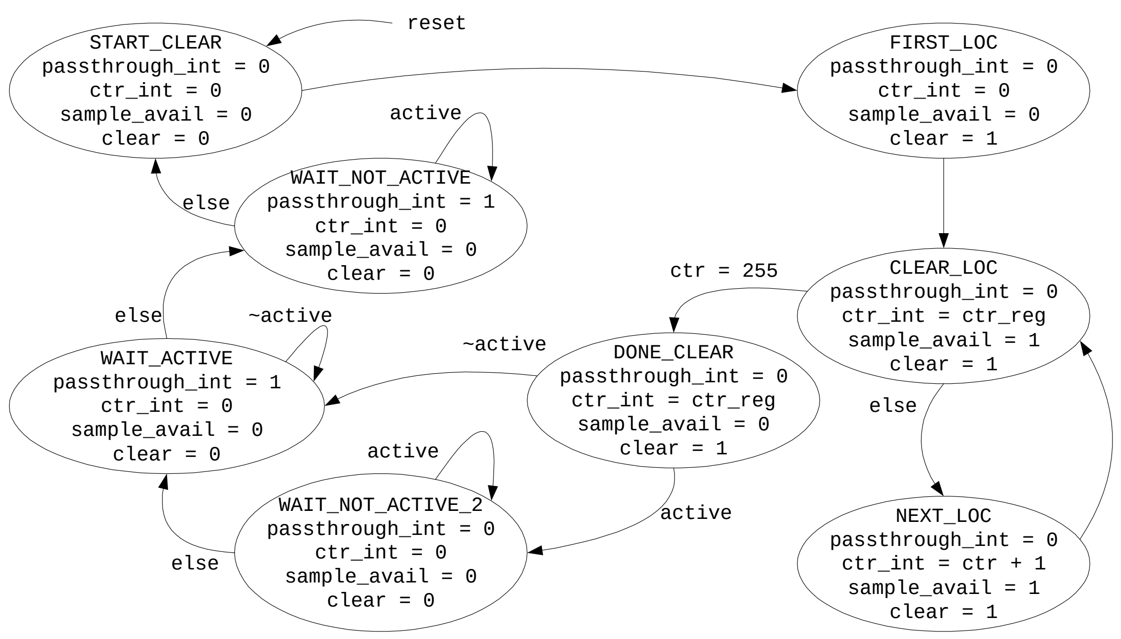


Figure 3: Position Analyzer FSM State Transition Diagram

2.2.2 Error Analysis

This module computes the error between the car's position and its intended position, and between its direction of motion and its intended direction. Since the car's steering mechanism can only be set to one of three discrete position for turning left, going straight, and turning right, a 2-bit output is sufficient for reporting the error in its direction. Similarly, the only use made of position error is to shut off the car's motor when it arrives at the target, and so one bit is sufficient for that purpose.

The Error Analysis module is conceptually much simpler than the Position Analyzer or Video Sampler, but it consumes more logic blocks than either of those modules, owing to its use of four multipliers with 16-bit or 20-bit outputs. When it was first compiled for unit testing, it did not fit even by itself in the Flex 10K10 FPGA, prompting the move to the larger Flex 10K70.

To compute its "at target" and "azimuth error" outputs, this module computes velocity by taking the difference between current position and the previous position provided by the Position Memory module. It also converts target coordinates (which are 4-bit values) to screen coordinates (which are 8-bit values) and computes the vector position of the target relative to the car. When the car is sufficiently close to the specified target position, then it is considered to be "at target". Azimuth error, which is the error in the direction of motion to the right or left of the target, is computed as the vector cross product of the velocity and relative target position. The X and Y components of this cross product are necessarily always zero, so the cross product may be treated as a signed scalar value. If the sign is positive, then the velocity is pointing to right of the target and the car should turn left, and if the sign is negative, then the car should turn right. "Azimuth error" is correspondingly reported as plus or minus one. However, if the magnitude of the result is sufficiently small, then the car should be allowed to go straight, and in this case "azimuth error" should be reported as zero. Strictly speaking, one would want the threshold for reporting zero azimuth error to be proportional to the speed of the car (see Appendix E). However, a constant value is used in this project in order to avoid computing a square root. For the same reason, the car position is defined to be "at target" if both the X and Y coordinates are within threshold of the target coordinates. A single 4-bit user-specified number determines both thresholds.

2.3 Testing and Debugging Notes

The turning radius of the car determines the minimum size of the field that the car can work in, because if the field is too small then the car will normally end up driving in a circle around the destination or going off the edge of the field. It was therefore necessary to use a wide-angle lens in order to fit a workably large field in the view of the video camera. However, the wide-angle lens caused another problem that was not visible to the human eye on the video monitor, but that became apparent during testing of Video Conversion. The brightness of the image formed by a lens decreases with distance from the center. In extreme cases where this effect is visible to the eye, it is called "vignetting". It is more severe in wider-angle lenses. Careful examination of the analog video signal showed that the brightness of the white background in the corners of the image was about the same as the brightness of the black car in the center of the image. It was therefore necessary to crop the image at the input to the Position Analyzer by setting the "accumulate enable" input false in the margins of the image where the white background appeared as black. This in turn made the process of converting target coordinates into screen coordinates more complex, because a simple 4-bit shift left had to be replaced with a multiplication and addition.

Simulations were carried out in Max+Plus II that were sufficiently detailed to test the major features of the system. In all cases where there was a discrepancy between the simulated behavior and the behavior of the hardware, a wiring error was found. Building up repetitive signals by copying and pasting progressively larger repeating chunks made it unnecessary to hand-enter every signal detail individually. For example, the video data in one simulation (see Appendix B) was created by first creating an entire field of 242 empty (all white) video lines, and then modifying a few lines' worth of data to create black areas in various positions in the image. This made it possible to trace internal signals that would have been difficult to access in the hardware, and it was thus possible to identify and correct a design error.

One bug was traced to the obscure fact that although Max+Plus II automatically implements signed multiplication for the Verilog "*" operator, only the low-order 8 bits of the result are valid, regardless of the widths of the inputs and outputs. It is therefore usually necessary to use the LPM_MULT library module for multiplication. Also, a comparison failed to work as expected when a decimal constant of unspecified bit width was given as one of the operands, whereas replacing the decimal constant with an 8-bit hexadecimal constant gave the desired result. These observations support the idea that in the absence of a comprehensive knowledge of the compiler's specifications, it is safest to specify explicitly everything that can be specified in the Verilog code.

2.4 Conclusions

It is helpful to write Verilog code in painstaking detail; a number of bugs in this project were solved that way. It is likely that other bugs were prevented by explicitly making all variables either registered or combinational, rather than letting the compiler decide. Testing as early as possible and at the lowest possible levels before integrating the components was crucial to the success of the iterative design-test-redesign process in this project, as was the use of test-only outputs to gain access to internal signals.

Noise and other unintended variations in the analog video signal turned out to be a surprisingly difficult and widespread problem. Perhaps that fact should not be surprising, since analog behavior is by definition more complicated and subtle than digital behavior.

The single design change that would probably yield the greatest overall improvement to this project would be to replace the 1-bit video converter with a multi-bit converter. It would then be possible to handle the vignetting problem with more sophisticated techniques that allow the threshold to vary with the average luminance of a local portion of the image. Such refinements would ultimately lead to the equivalent of edge detection, a commonly used first step in image analysis. Also, the azimuth error calculation should really be made independent of car speed and distance from target, either by multiplying the threshold by speed, or by dividing the cross product by speed. It would then be possible to calculate the azimuth error in such a way that it is zero only when straight-line motion guarantees that the car will pass within the distance threshold of the target.

3 Car Control System (By Joanne Mikkelson)

The car control system is responsible for the movement of the car. Controls from the video analysis system indicate the direction the car should move and when it should stop after reaching the target. A manual control starts the car moving towards the current target and a control to stop the car manually is provided in addition to the automatic stop from the video analysis system. The car control system also can change the speed of the car and measures the actual speed of the car. Additionally, for testing and demonstration, complete manual control is provided for the car's four control signals: forward, backward, left and right.

During automatic operation, the directional inputs, start, and the two stop controls are translated into the four control signals for the car. The automatic control system does not use backward movement, so this signal is always zero. Backward is provided for manual control and to allow its use in a more complex automatic control system. Stopping the car simply requires turning both forward and backward motion off. The four control signals and the desired speed are transmitted over a wireless link to the car. On board the car, this information is used to control the two motors in the car. An optical sensor counts wheel revolutions, and this count is transmitted back to the control system, where it is translated into a speed and displayed on the lab kit.

3.1 Design

Because the car is separated from the lab kit by the wireless link, two separate subsystems are required. The subsystem on the lab kit handles the external inputs and outputs and does all computation producing the directional signals and the speed information. The intent was to minimize the logic required on the car. The logic on board the car is only required to convert the directional signals to the proper voltages to control the motors and to collect raw speed information in the form of a counter. Figure 4 shows a view of the two parts of the car control system, and the controls and outputs of each.

The wireless kit provided for the project contains two small wireless modules and a larger board on which one of the smaller boards can be mounted. Each wireless module board holds the wireless chip, a Chipcon CC1010, its supporting hardware, and an antenna. The wireless boards are programmed in C, using macros and functions specific to the wireless chips. Once compiled, the code is loaded and runs on a small on-board processor. This allows the wireless board arbitrary C-coded functionality. The car-side subsystem takes advantage of this functionality so that it is not necessary to fit discrete logic and counters into the limited physical space on the car. Otherwise, the use of the programmability of the wireless boards was minimized, in favor of digital logic.

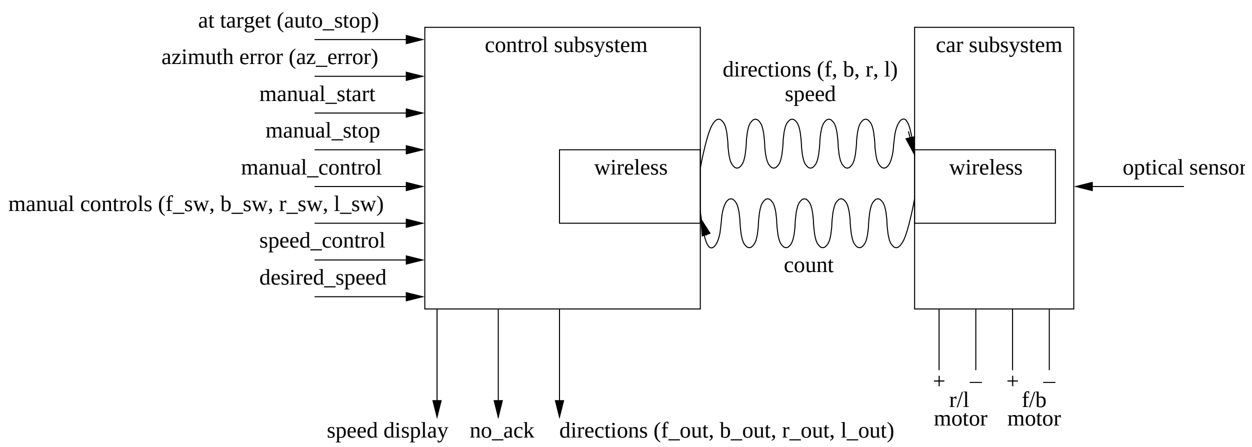


Figure 4: Two parts of the Car Control System

3.1.1 Control Subsystem

The control inputs to the car control system are all handled by the lab kit. Only the automatic stop, azimuth error, and manual start controls are used during automatic control of the car. Figure 5 is a detailed block diagram of the control system, implemented on the FPGA on the lab kit. The major components are the FSM responsible for the four car movement controls, the wireless control, the telemetry block and the display driver.

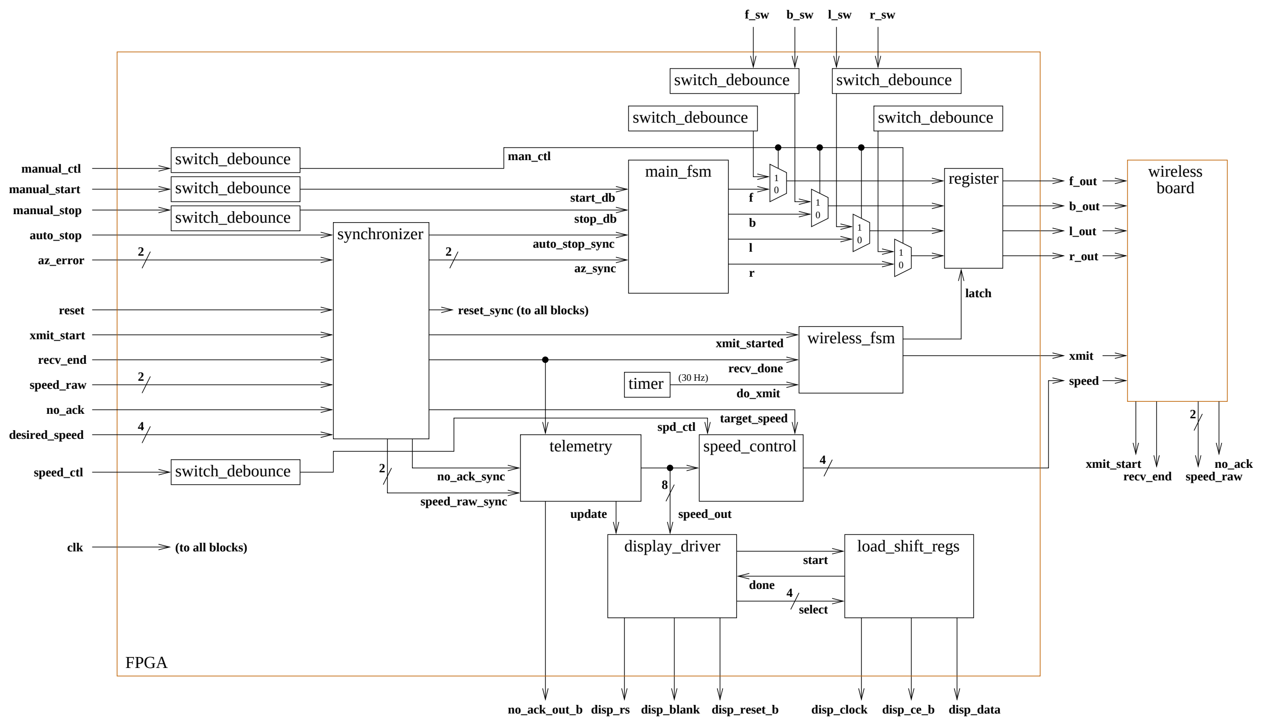


Figure 5: Detailed block diagram of Car Control

The `main_fsm` block accepts a two-bit azimuth error and the `auto_stop`, `manual_start` and manual_stop signals, all synchronized. The azimuth error and auto stop signals are generated by a separate lab kit, so must still be synchronized despite being automatically computed. Figure 6 is the state transition diagram. The FSM is used to maintain the current state of the control signals. This is especially important for stopping and starting, because once the car has been stopped, it should not move again until manual_start is asserted, even if the auto_stop signal is low, indicating that the car should be moving. This allows the target position to be changed after the car has stopped without causing the car to unexpectedly begin moving again. The azimuth error can be thought of as a signed-magnitude number, though since 10 is not a valid input, a twos complement representation is identical. An azimuth error of 00 means "go straight", while a negative azimuth error indicates that the car should turn right, and a positive error indicates "turn left." The azimuth error could require a low-pass filter but experimentation with the car under video control was necessary to determine if this was the case. Thus there is no such filter in the car control system; if one was needed, the FSM would receive a post-filtered version of the azimuth error.

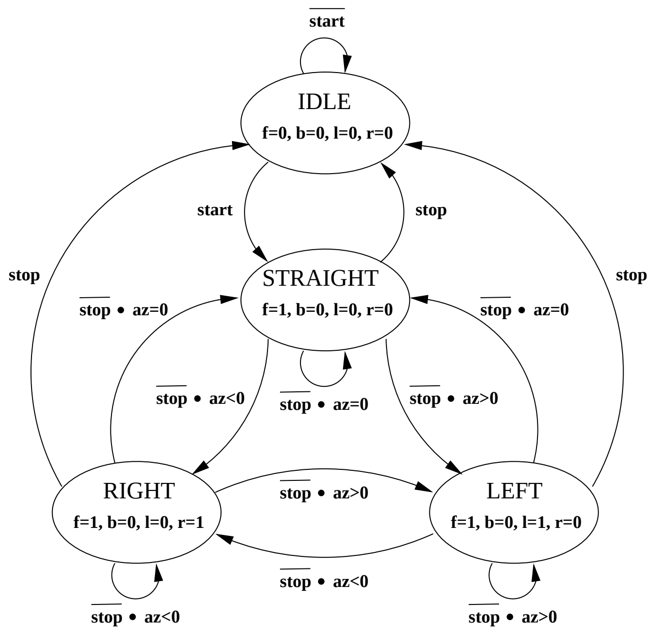

$\overline{\text{start}}$
IDLE
f=0, b=0, l=0, r=0
start
stop
stop
stop
STRAIGHT
f=1, b=0, l=0, r=0
$\overline{\text{stop}}$ • az=0
$\overline{\text{stop}}$ • az=0
$\overline{\text{stop}}$ • az<0
$\overline{\text{stop}}$ • az=0
$\overline{\text{stop}}$ • az>0
RIGHT
f=1, b=0, l=0, r=1
$\overline{\text{stop}}$ • az>0
LEFT
f=1, b=0, l=1, r=0
$\overline{\text{stop}}$ • az<0
$\overline{\text{stop}}$ • az<0
$\overline{\text{stop}}$ • az>0

Figure 6: FSM for the movement controls

The wireless board is controlled by the wireless_fsm block. An FSM is used to latch the outputs to the wireless board before the wireless board is instructed to read them. As the wireless board is not synchronized, the outputs (and the transmit signal itself) must not be allowed to change until it is known that the wireless board has read them; this is why the outputs are latched separately by the wireless FSM, rather than by the main FSM. A state transition diagram for the wireless FSM can be found in Appendix F. In addition to keeping track of when the wireless board has read the outputs, the wireless FSM keeps track of when the board has finished receiving data from the car. This proved to be very useful, as the transmissions were designed to be at 30 Hz, the frequency that the car's position would be sampled by the video analysis system. However, the wireless cards actually only managed about five round trips per second. The wireless FSM ensured, despite the lower transmission rate, that the directional outputs were not changed faster than the wireless card might read them. Given the time scales inherent in the motion of the car, 5 Hz was not an unreasonably slow transmission rate, so the system was allowed to remain unchanged.

The lowered transmission rate had more of an effect on the `telemetry` block. This module updates the speed and the no-ack outputs after the `recv_done` signal from the wireless board is asserted. The no-ack output is an LED, which is lit when the wireless board reports that it did not receive a response from the car within a 20 ms timeout. As this was expected to change at up to 30 Hz, the telemetry module has a two-state FSM, which counts `recv_done` signals to ensure that the LED stays lit for half a second. The speed display is also updated only once per second. The telemetry block accumulates the speed counter until a second has passed, then computes the speed from this. The FSM initially counted to 30 before updating the no-ack and speed displays, but had to be changed to update after 5 counts.

The speed computation is based on wheel revolutions. A wheel on the car travels nearly 6.5 inches per revolution. It seemed that a speed in feet per minute was most appropriate. The telemetry module accumulates the wheel revolution counts from the wireless board, and every second must compute $\text{count} * (6.5\ in) * (60\ s/min) \div (12\ in/ft)$. This works out to $(\text{count} * 32.5)$ and to avoid needing a divide or floating point computations, the final computation used is $(\text{count} * 65) >> 1$. The speed is passed to the `speed_control` block, but is not presently used to compute the desired speed transmitted to the car.

The last major part of the control subsystem is the `display_driver` block. The speed was initially to be displayed on the hex LEDs on the old lab kit. With the change to the new lab kit, hex LEDs were no longer available; instead a 16-character display was present. Each character requires serially loading 40 bits into the display. Thus the display driver, added towards the end of the design process, includes two large FSMs. The first FSM accepts the speed computed by the telemetry block, converts it into base 10, and loads 8 characters into the display (the other 8 are left blank). The speed displayed is a three-digit speed followed by "f/m". Each 40-bit character is loaded by a second FSM, the `load_shift_regs` block. The display can only be clocked every third FPGA clock cycle at the 27 MHz clock readily available, so the `load_shift_regs` FSM has many wait states. Perhaps the most difficult part of the driver was in figuring out that four 8-bit control words have to be loaded at once, one word for every four characters, rather than two control words, one for each display of eight characters.

3.1.2 Car Subsystem

As described at the beginning of section 3.1, a minimal amount of logic is performed on board the car. The wireless board waits for a transmission to arrive, at which point it extracts the four directional signals and the speed control value. The speed control value is an 8-bit number representing the duty cycle of the pulse-width modulation (PWM) output provided by the wireless board. With pulse-width modulation, motor speed is a function of this duty cycle. A duty cycle of 100% (255 on the wireless board) is full speed. Reducing the duty cycle, so that the motor is pulsed on and off, reduces the effective voltage and thus the speed of the motor. The wireless board outputs the PWM signal in addition to the four directional signals.

The wireless board is mounted on a small perforated board holding the remaining components necessary to drive the car. An LM317 linear voltage regulator provides a 3.3 V source to the wireless board from a 4.5 V battery output. On the board, the forward and backward signals are each ANDed with the PWM. The resulting signals are used as inputs to an LM18293 motor driver. The two outputs of the LM18293 should be passed through an RC filter to smooth the signal to the motor. The right and left signals are used directly as inputs to the LM18293 to drive the right/left motor attached to the front wheels.

After receiving a transmission and the outputs are changed, the wireless board transmits an acknowledgment back to the control subsystem. The number of wheel revolutions counted since the last transmission is included in this acknowledgment for computation of the speed of the car. The wheel revolutions are counted by a “reflective object sensor” attached to the car. A white spot on the tire of one of the back wheels is detected by a phototransistor on the sensor. The transition time of this signal is very long, 8 μs, and the signal resulting from this is low when a reflective object is detected, so the output of the sensor is passed through an inverter constructed from a transistor and a pullup resistor, and this signal is counted by the wireless board.

3.2 Implementation and Testing

The implementation of the project was primarily in Verilog on the FPGA in the lab kit. Most of the Verilog modules were unit tested in simulation before the entire system was simulated. After simulation, only the display driver required much testing on the lab kit. This was because determining that the display worked in simulation required knowing exactly what signals the display required. Misunderstandings there were reflected in the simulations just as in the code under test.

The two wireless boards were programmed and compiled in C. Because the actions performed by the wireless boards were very simple, testing the wireless by observing the control signals to and from the wireless board was sufficient. The no-ack LED was very good for denoting when the wireless link itself was not working, which usually turned out to be due to a lack of power on the car side.

The remaining implementation of the system was in the components on the board in the car. Each component of the board was tested with connections on the breadboard on the lab kit before being soldered onto the board. The optical sensor required two resistors, and the sensor and its inverter were demonstrated to work with the kit. A potentiometer was soldered on the board instead of a 3 kΩ resistor for the pullup resistor for the phototransistor so that the sensitivity could be tuned after the car was assembled.

At this point, everything was assembled onto the car for final testing and to tune the optical sensor. Unfortunately, while the control of the motors over the link from the kit worked, the motors were not strong enough to move the heavily-laden car more than a few inches. The car originally operated on two AA batteries; with the addition of the wireless board requiring three batteries (two batteries generating 3 V not being quite enough) and a separate set of three batteries for the motors to avoid having to worry about motor noise, the car was substantially heavier. An obvious solution was to remove one of the sets of batteries, relying on the voltage regulator to make sure no dangerous spikes from the motor reached the wireless board. Doing so resulted in the car not moving at all; the motors were not able to draw enough power from the batteries. Thus the second set of batteries was returned to the car.

As a result of the power problems, the final car assembly could not travel very far. Additionally, turning on the speed control reduced the speed to zero. The high-order bits of the eight-bit PWM duty cycle were transmitted from the kit; perhaps if these were all set high and the low-order bits were transmitted, a small speed difference could be achieved. The pulse-width modulated speed of the wheels was zero even when the car was lifted from the ground. However, the dramatic reduction in the speed of the wheels in the case when the car is turned right or left implies that using only the low-order bits of the PWM duty cycle probably would not be enough to allow speed control and turning at the same time. Fixing these power problems will require using different battery technology. Perhaps smaller batteries, AAAs, would be light enough. Possibly using a 9 V battery for the motors would be better. More drastic measures are probably not necessary.

A Appendix: Check-off Sheet

At completion:

☐ Demonstrate that manual start and stop operate correctly

☐ Demonstrate manual control of forward, back, left, and right with switches

☐ Show that telemetry is operating (speed and no-ack are displayed on the kit)

☐ Turn on speed control and observe the speed of the car

☐ Demonstrate (with manual inputs if possible) that the car responds correctly to azimuth error and auto stop inputs

☐ Display Position Analyzer X and Y outputs on hex LEDs, verify that they respond appropriately to various positions of manually positioned dark object on light background in video image.

☐ Display Azimuth Error output on hex LEDs, verify that it shows correct values for manually controlled dark object moving towards arbitrary target position and at 90°, and 270° to target position.

☐ Connect output of position error threshold discriminator to an LED and verify that it goes on when manually controlled dark object reaches target position.

☐ Demonstrate operation at full speed (i.e. system-level operation)

B Appendix: Handcrafted Test Data for Simulation

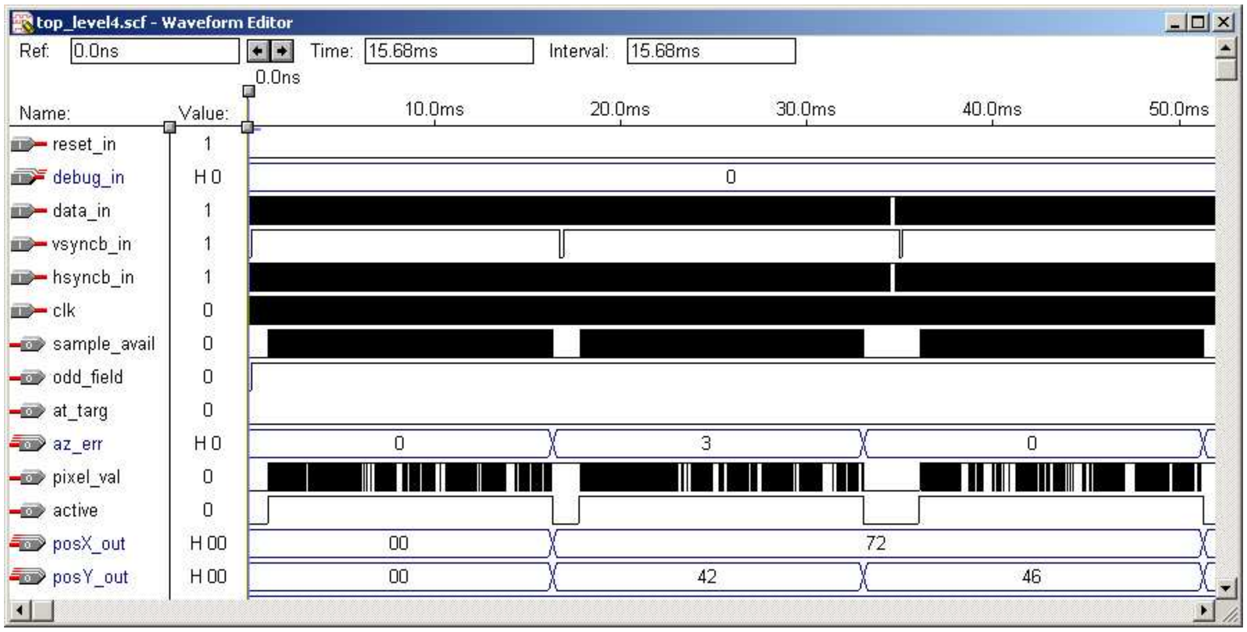


Figure 7: Simulation of three video frames, odd fields only

C Appendix: Analog to Digital Video Converter

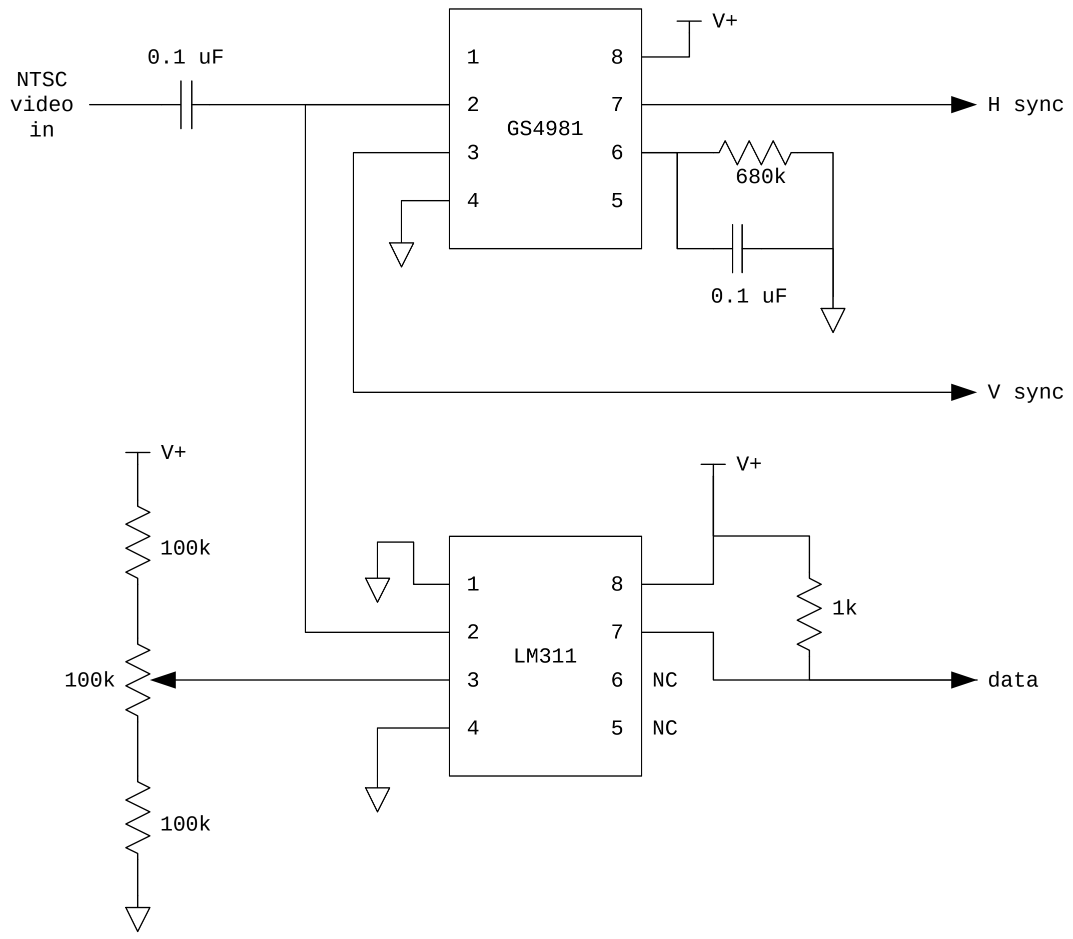


Figure 8: ADC Circuit Diagram

The 100 kΩ trimmer potentiometer is used to set the black/white threshold. The resulting binary image data appear at the data output. `H sync` and `V sync` are conventional video sync signals, i.e. they are normally high and briefly go low for sync pulses.

D Appendix: Video Sampler FSM

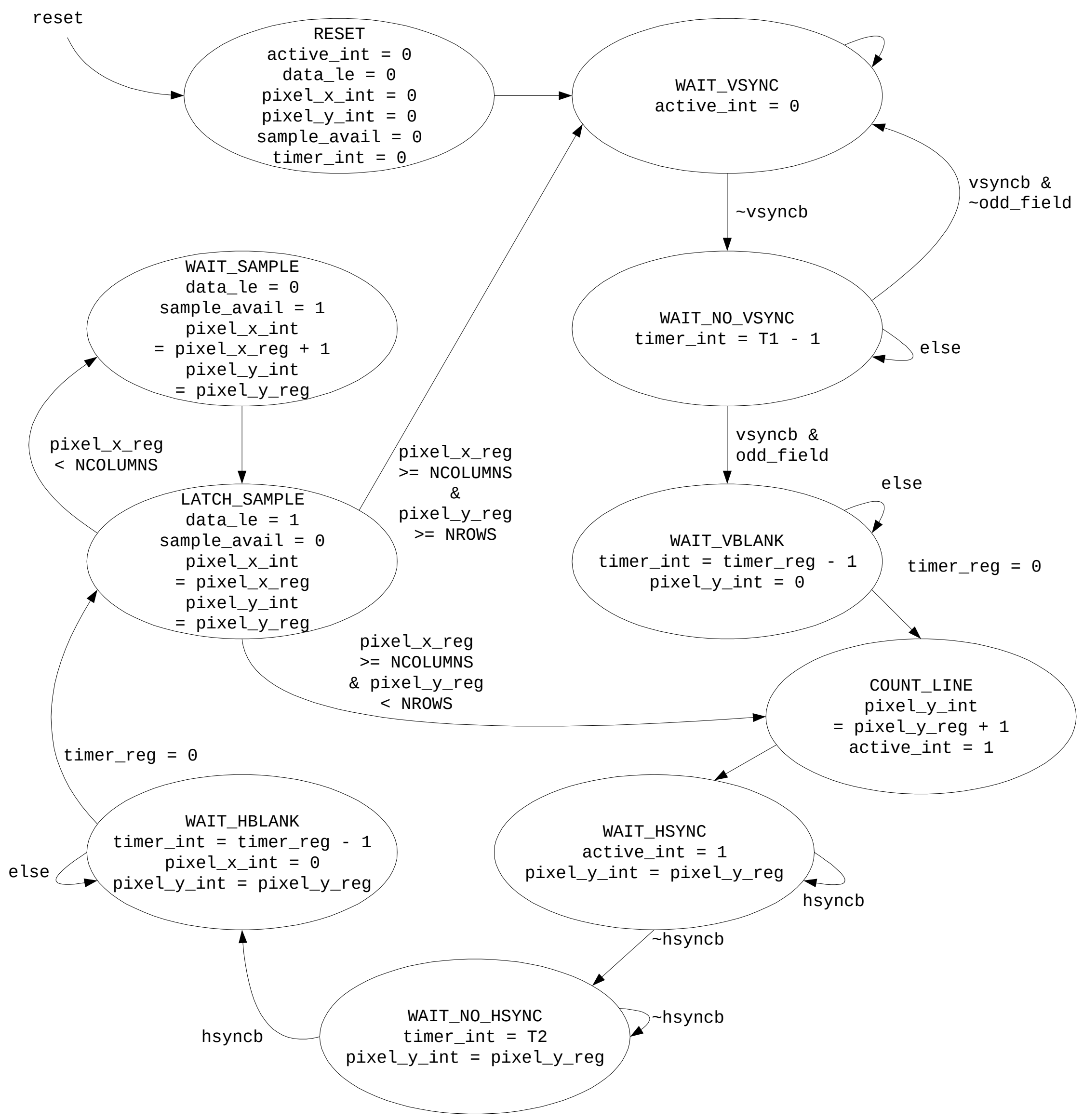


Figure 9: Video Sampler FSM State Transition Diagram

In the state transition diagram of Figure 9, the signals `vsyncb` and `hsyncb` refer to synchronized versions of the `V sync` and `H sync` outputs of the ADC circuit in Figure 8. An important feature not apparent from Figure 9 is a register that stores the value of a synchronized version of the ADC's `data` output when `data_le` is high on a positive clock edge. That register's output is thus the pixel value output of the Video Conversion module, and `pixel_x_reg` and `pixel_y_reg` are the pixel coordinate outputs.

E Appendix: Cross Product Computation of Azimuth Error

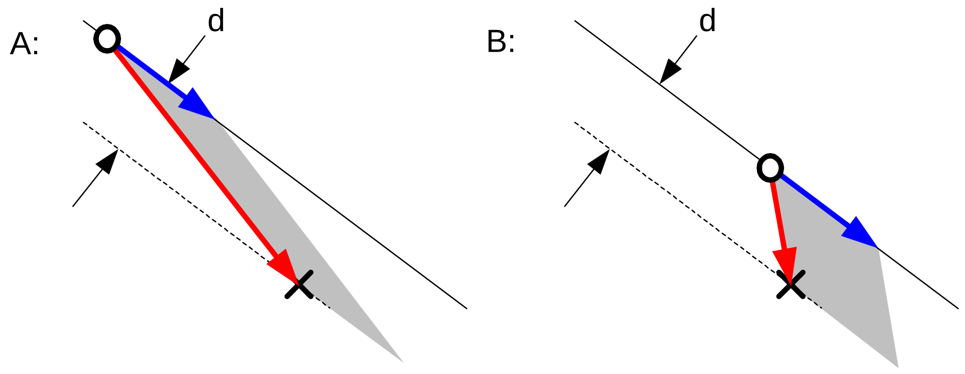


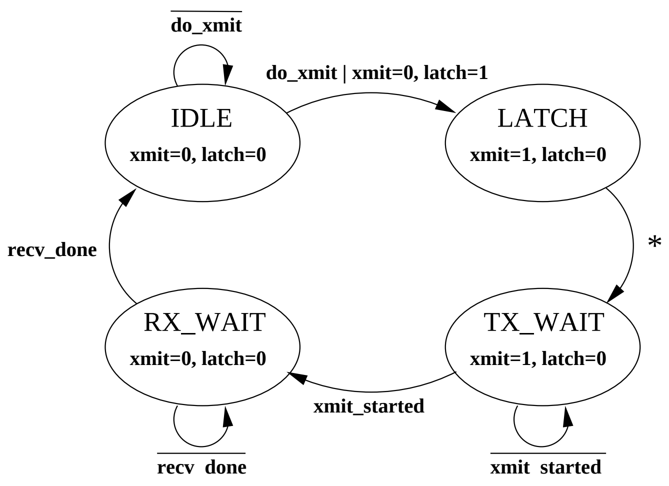
Figure 10: Geometry of Cross Product Computation at Two Points in Time

Figure 10 illustrates the motion of an object (circle) relative to a target (cross) at two points in time. Figure 10A shows one point in time, and Figure 10B shows a later point in time. The object is traveling at constant velocity (blue arrow), so it follows a straight line (solid line). The cross product of velocity with relative target position (red arrow) has a magnitude equal to the area of the gray parallelogram, and its sign depends on whether the line of travel passes to the right or to the left of the target. Using $\vec{\mathbf{v}}$ to denote the velocity vector and d to denote the minimum scalar distance from the line of travel to the target (see diagram), the area of the parallelogram is $|\vec{\mathbf{v}}| * d$. Notating the relative target position as $\vec{\mathbf{p}}$, we can solve for d:

$$d = \frac{|\vec{\mathbf{v}} \times \vec{\mathbf{p}}|}{|\vec{\mathbf{v}}|}$$

If d is less than the distance threshold for considering the car to be “at target”, then it is not necessary to adjust the car’s direction of motion.

F Appendix: Wireless Control FSM



$\overline{\text{do\_xmit}}$
do_xmit | xmit=0, latch=1
IDLE
xmit=0, latch=0
LATCH
xmit=1, latch=0
recv_done
*
RX_WAIT
xmit=0, latch=0
TX_WAIT
xmit=1, latch=0
xmit_started
$\overline{\text{recv\_done}}$
$\overline{\text{xmit\_started}}$

Figure 11: Wireless Control FSM State Transition Diagram

G Appendix: Car-side Wireless Board Code

This code uses the PWM output, a timer to stop the car should transmissions from the control subsystem cease, and the wheel revolution counter. The kit-side wireless code is similar, but without the three features listed above.

```
// main code for car side of wireless link
#include <chipcon/hal.h>

#define PREAMBLES 7
// the packet frequency is 30 Hz, or 33333 microseconds
#define TIMEOUT 83000 // timer in microseconds
#define CC1010EB_CLKFREQ 14746

#define RIGHT PORTBIT(1,6)    // pin 19
#define LEFT PORTBIT(0,3)     // pin 23
#define FORWARD PORTBIT(3,0)  // pin 27
#define BACKWARD PORTBIT(2,4) // pin 35
// sensor in is on pin 31, PORTBIT(3,2)
// PWM out is on pin 18, PORTBIT(3,4)

// counter for speed measurement
byte count;
// "modulo" argument for timer adjustment -- appears to be required
word modulo;

int main() {
  RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x15,    // Modem 0, 1 and 2
    0xAA, 0x80, 0x00,    // Freq A
    0x5C, 0xF4, 0x02,    // Freq B
    0x01, 0xAB,          // FSEP 1 and 0
    0x58,                // PLL_RX
    0x30,                // PLL_TX
    0x6C,                // CURRENT_RX
    0xF3,                // CURRENT_TX
    0x32,                // FREND
    0xFF,                // PA_POW
    0x00,                // MATCH
    0x00,                // PRESCALER
    };
    // Calibration data
    RF_RXTXPAIR_CALDATA xdata RF_CALDATA;
  int pwm_duty_cycle = 255; // full on
  byte len;
  byte received[2];
  byte sending[2];

  // Disable watchdog timer
  WDT_ENABLE(FALSE);
  // Startup macros for speed and low power consumption
  MEM_NO_WAIT_STATES();
  FLASH_SET_POWER_MODE(FLASH_STANDBY_BETWEEN_READS);

  // Set up ports; default is output
  // P0.3 = l
  PORTDIR(0) = PORT_OUT(3);
  // P1.6 = r, P1.0 and P1.2 = accidental Vcc input
  PORTDIR(1) = PORT_OUT(6) | PORT_IN(0) | PORT_IN(2);
  // P2.4 = b
  PORTDIR(2) = PORT_OUT(4);
  // P3.0 = f, P3.4 = PWM, P3.2 = optical sensor output
  PORTDIR(3) = PORT_OUT(0) | PORT_IN(2) | PORT_OUT(4);

  // Set default values
  count = 0;
  PORT(0) = 0;
  PORT(1) = 0;
  PORT(2) = 0;
  PORT(3) = 0;

  // Set up counter interrupt
  INT_EXTERNAL0_TRIGGER_ON_LEVEL();
  INT_PRIORITY(INUM_EXTERNAL0, INT_LOW);
  INT_ENABLE(INUM_EXTERNAL0, INT_ON);

  // Set up rx watchdog timer/interrupt
  halConfigTimer01(TIMER1 | TIMER01_INT_TIMER, TIMEOUT,
                   CC1010EB_CLKFREQ, &modulo);
  INT_ENABLE(INUM_TIMER1, INT_ON);
  TIMER1_RUN(TRUE);

  // Enable interrupts
  INT_GLOBAL_ENABLE(TRUE);

  // Configure timer 2 as a PWM timer
  halConfigTimer23(TIMER2 | TIMER23_PWM, 0, CC1010EB_CLKFREQ);
  PWM2_SET_PERIOD(1); /// 62 is about 1/2 second
  PWM2_SET_DUTY_CYCLE(pwm_duty_cycle);

  // Set up RF
  halRFCalib(&RF_SETTINGS, &RF_CALDATA);

  while (1) {
    // Start by receiving
    halRFSetRxTxOff(RF_RX, &RF_SETTINGS, &RF_CALDATA);
    len = halRFReceivePacket(0, received, 2, NULL, CC1010EB_CLKFREQ);
    if (len != 2) {
      // CRC error
      sending[0] = 1;
    }
    else {
      // restart watchdog timer
      ISR_TIMER1_ADJUST(modulo);
      TIMER1_RUN(TRUE); // in case it wasn't running
      sending[0] = 0;
    }
    // NOTE this can lose data if the interrupt comes at just
    // the wrong time but I need mutexes to do better.
```

```
    sending[1] = count;
    count = 0;

    halRFSetRxTxOff(RF_TX, &RF_SETTINGS, &RF_CALDATA);
    halRFSendPacket(PREAMBLES, sending, 2);

    // now, effect changes
    if (len != 2) {
      continue;
    }
    if (received[0]) {
      PWM2_SET_DUTY_CYCLE(received[1]);
      TIMER2_RUN(TRUE);
      if (received[0] & 0x1) {
        LEFT = 0; RIGHT = 1;
      }
      else if (received[0] & 0x2) {
        RIGHT = 0; LEFT = 1;
      }
      else {
        RIGHT = 0; LEFT = 0;
      }
      if (received[0] & 0x4) {
        BACKWARD = 0; FORWARD = 1;
      }
      else if (received[0] & 0x8) {
        FORWARD = 0; BACKWARD = 1;
      }
      else {
        FORWARD = 0; BACKWARD = 0;
      }
    }
    else {
      FORWARD = 0; BACKWARD = 0;
      RIGHT = 0; LEFT = 0;
      TIMER2_RUN(FALSE);
    }
  } // while (1)
}

void isr_extint0() interrupt INUM_EXTERNAL0 {
  count++;
}

void TIMER1_ISR() interrupt INUM_TIMER1 {
  // haven't heard from controller; stop car
  TIMER2_RUN(FALSE);
  FORWARD = 0; BACKWARD = 0;
  RIGHT = 0; LEFT = 0;
  TIMER1_RUN(FALSE);
}
```