University of Alberta

EE 552 Final Report

Driver's Ed

Raymond Sung, 345630
Patrick Chan, 225463
Jason Mah, 354665
Andrew Sung, 364189

December 6, 1999

Abstract

This project involved the design, implementation and test of a digital telemetry system. The completed system successfully gathers data from various sensors mounted on a remote controlled vehicle using a field programmable gate array. The FPGA packetized the data for synchronous transmission over radio frequency to a remote base station. The base station, which was implemented using a combination of two field programmable gate arrays, correctly recovered the transmitted data, did the necessary calculations and displayed the results in real-time on a VGA monitor. The RF transmission scheme was able to disregard invalid data and functioned correctly even when in close proximity to another system operating at an identical radio frequency. The collected telemetry data included relatively accurate measurements of the car's instantaneous velocity, instantaneous acceleration, heading and total distance traveled. Furthermore, the car's proximity to other objects could be detected at short distances and the results displayed on the VGA screen. The base station was able to interface to a 4x4 keypad and display user keypresses on the monitor. The keypad interface allows future expansion of the system for direction and heading control.

Current Feature Set Implemented

The feature set implemented as determined by VHDL simulation and observation of hardware behavior include: (electrical verification involves measurements with electronics test equipment)

Priority = High is an essential feature.
Priority = Medium/Low are features that can be deleted due to time or space constraints.

System Block	Feature	Priority
RC Car	Acceleration measurement	High
	Velocity measurement	High
	Distance measurement	High
	Heading measurement	High
	Proximity detection	Medium
	RF telemetry transmission	High
	RF movement instruction reception	High
Base station	Telemetry display on VGA monitor	High
	Advanced Telemetry Display with Visual Warnings and Graphics	Low
	Keypad Control interface	High
	Movement instruction / Telemetry storage (RAM)	High
	RF telemetry reception	High
	RF movement instruction transmission	High

All high priority items outline in the feature set are implemented according to the specifications. The proximity detection is the only medium priority item that is completed. Currently, the medium and low priority items are being design to incorporate into the system.

Sensor Data Collection and Telemetry Transmission

Implemented Features
Hardware

- Acceleration data acquisition using a micro-machined accelerometer as demonstrated by VHDL simulation and electrical verification
- Velocity and distance data acquisition using an optical encoder as demonstrated by VHDL simulation, electrical verification and mechanical mounting

- Proximity sensor data acquisition as demonstrated by VHDL simulation and electrical verification
- Data path multiplexer and bus logic, Multiple State Machine control logic for sensor interface, information packetization and data encoding. VHDL timing simulations complete, electrical verification complete.

Software

- Multiplexing of sensor data complete as demonstrated by VHDL simulation
- Encoding and packetization of multiplexed sensor data complete as demonstrated by VHDL simulation
- Acceleration calculation complete as determined by VHDL simulation
- Velocity and Distance calculation complete as determined by VHDL simulation
- Proximity detection system complete as determined by VHDL simulation
- Compass direction complete as determined by VHDL simulation
- Directional sensor data collection using digital compass
- Increase the range of detection for the proximity sensors with the addition of voltage comparator

Base Station Telemetry Reception and Data Processing

Implemented Features
Hardware

- Keypad recognition by VGA and debounce complete as demonstrated by functional verification
- RF data reception demonstrated through electrical verification

Software

- VGA text and graphics display as verified by direct observation
- Keypad interface complete as determined by VHDL simulation
- RF decoding and data recovery
- Data processing and real-time updates of telemetry data complete as demonstrated by VHDL simulation
- User interface in VHDL code complete, no VHDL simulation, verified by direct observation

Description of Operation

The purpose of the RC Car FPGA is to acquire telemetry data from the four different sensors mounted on the RC car and then send the data through an RF transmitter to the base station. The four sensors, an accelerometer, an optical encoder, four proximity detectors, and a compass are used to measure the instantaneous acceleration, the velocity, the distance, the magnetic heading, and the proximity to objects. There are two outputs from the accelerometer, one for x-acceleration and one for y-acceleration. Both outputs are pulse width modulated (PWM). A counter within the FPGA is used to count the width of the pulse. The optical encoder outputs 128 counts per revolution. A counter within the FPGA is used to count the number of pulses in a specified period, about 55ms. The proximity is detected using opto-electronic transmitter and receiver circuit. Each of the four proximity outputs is an input to the FPGA and stored in registers. The compass heading is detected using Hall effect sensors and each of the four outputs is an input to the FPGA as well.

The flow of signals within the FPGA is shown in the RC Car Flow Diagram found in the appendix. From this diagram, the data path can be seen. A finite state machine controls the sequence that the data is read into the packet encoder. The packet encoder then converts the 8-bit parallel data into a packetized serial bit stream suitable for RF transmission. The RF data will then be transmitted to the base station with monolithic RF transmitter.

RC Car Design

Accelerometer Design

To accurately measure the instantaneous acceleration of the RC vehicle, an Analog Devices, ADXL202 accelerometer with digital outputs is used. The advantages of this device include measurement of acceleration on both the x and y axes and digital outputs, eliminating the need for a A/D converter. Since the outputs of the accelerometer are pulse width modulated (PWM), a counter within the FPGA is used to count the width of the high pulse. The period for one complete cycle is set to 10ms using an external resistor as described in the accelerometer hardware section. The internal clock used to count the width of the pulse is 500kHz as recommended by the ADXL202 data sheet. Counting of the pulse width is accomplished by using the outputs of the accelerometer as an enable signal to the counter. Due to the restriction of the number of logic cells in the FPGA, the actual count of the pulse width is divided down by 32 using a 5-bit counter. The MSB is then used as the input to a 7-bit counter. The value of this counter is the value that is passed on to the base station for calculation. While this implementation reduces the accuracy of the accelerometer, the accuracy is only reduced by <2% and is thus deemed acceptable. For example, if the accelerometer is at rest the duty cycle is 50% which translates into a high pulse of 5ms. Using a 500kHz clock, this translates into 2500 counts. Dividing this number by 32 results in 78.125, and thus the 7-bit counter would only register 78, which when multiplied by 32 again results in 2496. This is a 0.16% error. Other values of acceleration will have more % error but all less than 2%.

To further conserve logic cells, the counter alternates between counting the x acceleration and y acceleration through a finite state machine controlling a 2-to-1 mux that selects between the two accelerations. For the base station to differentiate between the two accelerations, a bit is passed along with each set of acceleration data to the base station, thus making each set packet of data sent 8 bits wide. A state diagram in the appendix illustrates this state machine. Finally, before the data is sent to the data path multiplexer, the output is registered in a register with enable. The controller for the data path checks the value of the enable before reading the value of the counter, thus ensuring that the counter value is not changing when the controller is attempting to read the acceleration value.

Note that while the y acceleration is not currently used by the base station in the calculation of the acceleration, the y acceleration count is still passed onto the base station for possible future implementation.

The major design difficulty in designing the module for the accelerometer lies in the limited number of logic cells in the FPGA. Initially, both accelerations were counted simultaneously and passed onto the base station. However, the solution of dividing down the count value and using only one counter to count both accelerations reduces the logic considerably. The other design difficulty involved the different clock frequencies. This problem is solved using the method described above.

Optical Encoder Design

The optical encoder produces 128 counts per revolution (CPR). A small finite state machine (see appendix) was used to take the output from the encoder and pass it to the base station. The finite state machine has two counters. The first counter is a 8-bit counter used to count the number pulses that is outputted by the encoder. The second counter is a 4-bit counter that counts to 16 and then resets the first counter. The second counter is used because each packet is 3.72 ms, a interval too short to obtain any accurate data. For example, if the RC car were travelling at a rate of 20km/h (the approximate maximum speed of the RC car) and the circumference of the wheel that the optical encoder is mounted is 16cm. By performing some simple arithmetic, the estimated maximum number of counts per second is ~4444 (or 4.444 counts/ms). However, the car will probably never operate at this fast. A better operating speed would be about 13 km/h or about 3 counts/ms. The counter used within the FPGA to count the number of high pulses counts on the rising edge and the value of the counter is read approximately every ~3.5ms (corresponding to the period of one complete packet). This translates into ~10 counts/(packet period). At this rate, the calculation at the base station of the distance and velocity would not be meaningful. Thus, it was decided that the distance and velocity should only be calculated every 16 packets (~59.52ms). So when the counter value has been read from 16 times, the counter is reset to zero.

As stated above in the accelerometer design section, the biggest challenge in designing the interface for the optical encoder is the conflicting timing requirements of the two components.

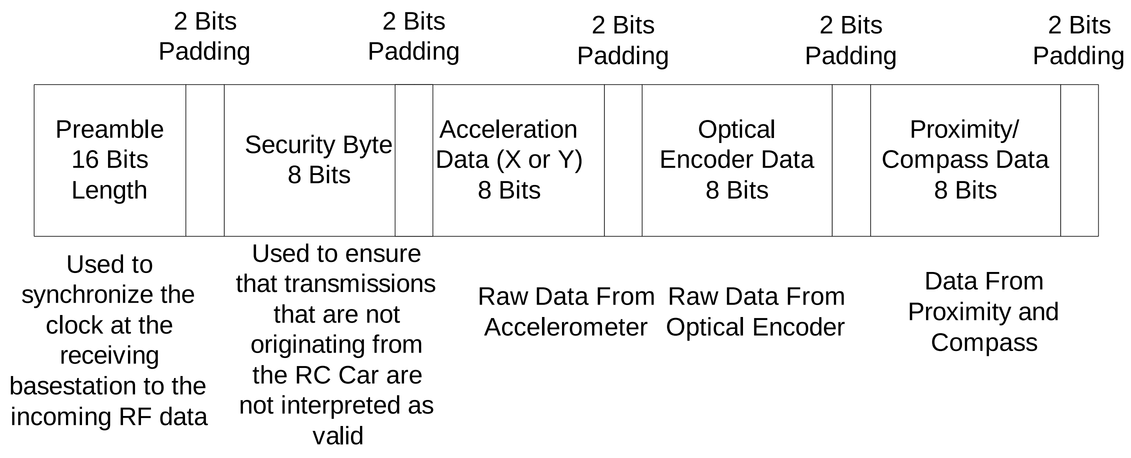
Data Path Controller

The data path controller controls a 4-to-1 mux, which selects the order that the telemetry data is sent to the packet encoder module. Each input of the 4-to-1 mux is a bus 8-bits wide. The accelerometer value and the encoder value are both eight bits wide while each of the proximity and compass values are four bits wide. Thus, the proximity and compass values are concatenated together to form an 8-bit bus. All three buses form three of the inputs to the 4-to-1 mux. The fourth input is a bus that is tied high and is used for sending all 1’s when an error occurs. An example of an error occurring is the controller attempting to read the accelerometer value when the accelerometer state machine is writing to the register. The data path state machine operates interactively with the packet encoder controller. Three flag signals are sent from the packet encoder controller indicating which of the three sets of data is needed. The state machine then selects the appropriate control signals. If an error occurs, the state machine transitions into an error state where eight 1’s are outputted.
The main design challenges for this module were the timing specifications of the optical encoder and the accelerometer. Since the accelerometer outputs are PWM, the time between successive cycles varies. This conflicts with the periodic sampling interval for the optical encoder. Periodic sampling is required for accurate speed calculation by the base station since speed is a time dependent measurement. Thus, a system had to be designed that could sample the encoder value periodically yet still sample the pulse width of every accelerometer cycle. The latter condition, although not essential to the correct calculation of the acceleration, is still desirable.

Packet Encoder

The RF data packet was formatted so that the receiver at the base station can synchronize with the incoming data stream using the 16-bit pre-amble. Furthermore, the 8 bit security codes must match on both ends of the RF link for that particular packet to be accepted. Otherwise, the base station will reject the current packet and wait for another 16-bit pre-amble. This data-encoding scheme is robust enough to ensure adequate system operation in the shared communications spectrum. As discussed earlier, the RF link was implemented using integrated FSK PCB mounting modules. According to the manufacturer's specifications, the transmitted data had to be run-length limited to 20 ms so that the RF receiver could accurately recover the bitstream. The run-length is the number of bit-periods that the data remains at a '0' or a '1.' The total packet length was 58 bits with each bit having a length of 64μs. This translated to a total packet length of 3.712 ms. Since, the transmissions were synchronous, the preamble "1010 …" stream would appear every 3.7 ms thus meeting the run-length requirement.

The design of the packet encoder involved three components. The finite state machine was used to control a data path consisting of a shift register to convert parallel bus data into a serial bitstream to the RF transmitter, a 5 to 1 multiplexer is used to select from the 5 data sources that needed to be shifted out and an asynchronous counter. The counter was necessary to allow the state machine to remain in a particular state until all of the data from the 8-bit bus was shifted out. The control signals Accel Chip Select, Encoder Chip Select and Prox Chip Select start the Accelerometer, Optical Encoder and Proximity/Compass Detector state machines when data from those particular sensors are required.


2 Bits Padding
2 Bits Padding
2 Bits Padding
2 Bits Padding
2 Bits Padding
Preamble 16 Bits Length
Security Byte 8 Bits
Acceleration Data (X or Y) 8 Bits
Optical Encoder Data 8 Bits
Proximity/ Compass Data 8 Bits
Used to synchronize the clock at the receiving basestation to the incoming RF data
Used to ensure that transmissions that are not originating from the RC Car are not interpreted as valid
Raw Data From Accelerometer
Raw Data From Optical Encoder
Data From Proximity and Compass

RF Packet Format

Challenges: It was critical to reduce the size of the state machine since the number of logic cells on the RC car FPGA was limited. Furthermore, it was determined after several design iterations that the shift register and state machine would not function properly unless it was coded as a strictly Moore Machine. Therefore the number of states had to be relatively large.

Proximity and Compass Design

The inputs from the four proximity circuits and the four compass inputs are non-periodic signals that do not need any control logic to acquire. The proximity sensors will output a high signal when the RC vehicle is not near to any objects and a low signal when an object is closer than 5cm. The compass will send a low signal on one or two of the input lines from the compass to the FPGA indicating which direction the RC vehicle is heading. Both the proximity and the compass inputs are inputted to registers where the data path controller can sample the signals as required. The registers are used to avoid unwanted signal spikes. There were no design difficulties in designing for these sensors.

Flashing LED's

After the complete design of the RC Car, about 10 logic cells remained. These 10 cells could not be used to implement any more features. Thus, a simple controller controlling a bank of LED's at the front of the RC Car is also added. This bank of LED's adds visual appeal to the RC Car. One benefit from implementing this feature, is the actual reduction of logic cells in the overall FPGA. Prior to the addition of the LED controller, different synthesis styles had not been attempted as the project fit on the first synthesis. After adding the LED controller the project no longer fit into the FPGA. Thus, different compile options were implemented and a smaller fit was eventually found. There were no design difficulties in designing the LED controller.

FPGA Choice

The UA7K development board with an EPM7128SLC84-10 FPGA was chosen over the UP1 development board with an EPF10K20RC240-4 FPGA for two reasons:

1. The MAX7k part is prom memory based rather than the static RAM based implementation of the FLEX10K20 part. This is important, as it is impractical to reprogram the FLEX10K20 FPGA after every power up since the FPGA is mounted on the RC car and powered from a battery pack. Due to the use of a battery pack, powering down the RC car whenever the car is not in use saves power and increases the time that the RC car and data acquisition controller can be used. If the FPGA had to remain powered even the car is not operated; the battery

pack would not last very long before needing a recharge. The need of a frequent recharge would be very time consuming.

2. The UA7K board is much smaller than the UP1 board. In addition to mounting the FPGA on the RC car, the four different sensor modules also must be mounted. Due to the size of the car, it was not practical to mount the sensors and the UP1 board onto the RC car at the same time.

While the FLEX10K part has many more logic cells, 1152 versus 128, the fact that the MAX7k is flash based and the smaller size of the UA7K board was deemed more important. Thus, the UA7K board was chosen over the UP1.

Clock Frequency

The clock frequency chosen for the system was 15.625kHz. This frequency is obtained by dividing down the 1MHz clock provided by the crystal oscillator on mounted on the UA7K board. This frequency was chosen for a number of reasons. Firstly, this frequency is divided down 64 times from the 1MHz oscillator. This is an advantage as 64 is a 2^n number, which means that this frequency can be directly obtained by taking the MSB of a 6-bit counter. Using this approach to divide down a clock saves logic cells, which is at a premium on the MAX7k128. Secondly, the frequency must be close to a frequency that can be obtained on the base station FPGA that uses a 25.175MHz crystal oscillator. By dividing down the 25.175MHz clock by 1611, a frequency of 15.627kHz can be obtained. The 2Hz difference between the two clocks was deemed close enough. There may be another pair of frequencies that is closer together but finding them would have been too time consuming and not worth the effort since the two frequencies found meet the requirements. A third parameter is that since the sensors are acquiring data in the order of milliseconds and in some cases, in seconds. Therefore, the clock frequency cannot be too fast. On the other hand, a clock that is too slow would mean slower data transmission and thus slower data processing by the base station. A fourth parameter is the maximum transmission rate of the RF transmitter, which is 20kbps. This translates into a maximum frequency of 40kHz. The selected frequency of 15.625kHz is well under that specification. While a rate that is faster could have been used (say 31.25kHz) and still be under the maximum transmission rate, 31.25kHz would not meet the second or third design parameters discussed above.

Base Station Design

The design is broken into three major modules. This was done to provide a framework to start from and to make it easier to divide the tasks. The three modules consists of the RF decode, data analysis and VGA. The following design description will also be broken into these modules.

RF decode

Decoding the received RF data involves: 1) using the preamble word to synchronize the incoming data rate with the clock used to receive the data; 2) checking the security byte for data packet validity; 3) separating the different bytes of data and storing the three different bytes in separate registers where they can be read by the data analysis modules.

Preamble Word (preamble_fsm.vhd)

This finite state machine (see appendix for state diagram) checks the first fourteen bits of the preamble word ("1010_1010_1010_10") to determine the start of the packet. The preamble word is also used to synchronize the receiving clock with the incoming data rate. This is necessary for two reasons. Firstly, when the system is initially powered up or after a global reset, the receive clock is at an unknown phase with respect to the incoming data. This is undesirable due to possible setup and hold timing violations in registers. This could cause some of the bits of the packet to be lost. Secondly, the incoming data rate 7.8126kbps while the receive clock is operating at 15.62694kHz. Ideally, the receive clock should be double the incoming data rate to prevent clock drift. So, the ideal receive clock frequency should be 15.625kHz. Thus, there is a 1.94Hz clock difference between the actual clock and the ideal clock. This small difference will cause the clock to drift to the left 8ns with respect to the incoming data every bit that is received. Eventually, when the clock drifts close enough to a transition in the incoming data the setup time for storing the bit into a register will be violated. To solve these problems, the receive clock must be synchronized to the incoming data stream on every packet. Thus, for one complete packet the clock will drift only ~460ns to the left.

Security Check (security_check.vhd)
This state machine (see appendix for state diagram) verifies whether the security byte matches with expected data from the RC car. The selected pattern is "10001000." If the security code does not match, the Security_Corrupted Flag will be set high for the duration of the packet. The flag will continue to be high until it receives a valid preamble from the RC car in which case the flag will be reset to low.

Data Deocder (data_decoder.vhd)
The entity *data_decoder* (see appendix for state diagram) is used to transform 8 bits serial data into an 8 bit vector data. It uses a state machine with a clock frequency of 64 us to accomplish the modification of data. The state machine starts when the *packet_decoder* entity sends a signal to the *data_decoder*. This signal is represents the security signal has been received and the following serial data is valid. When this occurs, the *data_decoder* will wait for 2 clock cycles. This two clock cycles is required to ensure that the two extra of '1' generated by the *packet_encoder* during the transmission of data over the RF are not read by the *data_decoder*. The *packet_decoder* will then shift the next 8 bits of serial data into a register. This is accomplished by the use of lpm_shiftreg. The shift register converts the 8 bits serial data into an 8 bits vector data. This 8 bit vector data is the acceleration data. The next 2 clock cycles will not be read by the *data_decoder*. These two bits are the extra bits generated by the *packet_encoder*. Once again, the next 8 serial bits will be shifted into an 8-bit vector by the lpm_shiftreg. This new 8-bit vector is the distance data. The following next 2 bits are ignore by the *packet_decoder* (extra bits). The next 8 bits of serial data will then be converted into a 8 bits vector by the lpm_shiftreg. This is the direction and proximity warning data. The last two bits are data are ignored by the *packet_decoder* (extra bits).

Demux_top (*demux_top.vhd*)
The *demux_top.vhd* is used to separate the data collected from the package decoder and pass them into their respective entities for data calculation. The data collected from the package decoder contains 3 types of information. The first series of data is a counter, which counts for the length of a pulse width from the accelerometer. The second set of data is a counter that counts for the distance travel from optical encoder. The final set of data contains information about the proximity detection and heading direction.

The package decoder will output 8 bits of data into one register and then after 2 clock cycles, 8 bits of new data will be shifted into the same register. Therefore, the *demux_top* entity has only 1 clock cycle to read from the register. When the package decoder is shifting data into the register (using lpm_shiftreg), an enable signal is set to high. During the time, the enable signal will prevent the *demux_top* entity from reading the register. When the enable signal goes low, the *demux_top* entity will read from the shift register and transfer the data into either acceleration, distance or direction-proximity register. A state machine controls the *demux_top* entity. As a reset signal is received, the state machine is set to the *start* state, on the next clock cycle; the state machine will go to the *security_state*. While inside this state, the package decoder will be reading in the preamble signals and the security bits. The state machine will enter the next state only when a high enable signal is detected. As a result the package decoder has now finished reading the preamble and security bits and is shifting the acceleration bits into a register. The state machine is now in the *accel_state*. When 8 clock cycle has passed (the package encoder has shift in 8 bits), the enable signal will go low which allows the *demux_top* to transfer the data from the shift register into the acceleration register. This is the *read_accel* state and will last only 1 clock cycle. The next state is the *temp_state1*. The purpose of this state is to allow the package decoder to read in the 2nd intermediate bits (there are two bits separating the acceleration, distance and direction-proximity bits). The state following this is the *dis_state*. This state is similar to the *accel_state* with the exception of the data being shifted is the data from the optical decoder. The next state is *read_dis*, which transfers the data from the shift register into the distance register in the same manner as the acceleration data, is written into the acceleration register into the *read_accel*. The *temp_state2* is used to read the 2nd intermediate bit. The next set of states (*dirprox_state*, *read_dirprox*, and *temp_state3*) function similarly to the *accel_state*, *read_accel*, and *temp_state1*, with the exception that the data is read into the direction-proximity register.

The most important aspect of the *demux_top* is the timing of when the data is read. The shift register must not be accessed by the *demux_top* when the package decoder is writing data to the register. Otherwise, the data read will be invalid.

Data Analysis

On board the RC car there are four sensors that output data to the Base station. These are the accelerometer, optical encoder, compass and proximity sensors. The data collected from these sensors will be used to calculate the acceleration, velocity, distance, direction and collision detection.

Acceleration (accel.vhd)
The entity *accel* will calculate the acceleration of the RC car. The input to this entity is the clock (40ns), the reset signal, an enable and the data input from the packet decoder. The data input from the packet decoder is first written into an acceleration register every 3.72 ms. Because the register is update so fast, the acceleration calculation is done every 30 packets (132.6 ms). This way, once the calculation is done, the VGA monitor can be updated. Whenever the packet decoder writes data into this register, an enable signal will increment a counter. When the counter reaches 15 (15 because there are two sets of acceleration data: x and y axis), the *accel* entity will calculate the acceleration of the RC car, and the counter is reset to 0. While the counter is incrementing, the old value of the acceleration will be used to display onto the VGA monitor.

The data receive by the entity *accel* from the packet decoder is a counter value. This value represents the length of T1 from the accelerometer. The calculation of the acceleration is based on the formula:

$$Acceleration = \frac{\frac{T1}{T2} - 50\%}{12.5\%} x 9.81 m/s^2$$

where T1 = the length of the pulse width
T2 = the period of the pulse

First of all, the value obtained from the packet decoder is T1/32. This division is required because of the process how the data are packaged (as explained in the accelerometer's data collection). Also, because MAXPLUSII only allows multiplication or division by a number of based 2 (lpm_division and lpm_multi are not chosen to reduce the number total logic cells used), the acceleration value obtained by *accel* entity is an approximation of the actual acceleration of the RC car. With T2 set to 10.0 ms, and the clock frequency of the RC car set to 500kHz (2ns), this means that the counter value for T2 is 5000. Thus, the acceleration formula becomes:

$$Acceleration = \frac{\frac{32xT1}{5000} - 50\%}{12.5\%} x 9.81 m/s^2 \approx (5T1 - 390) x 10^{-1} m/s^2$$

where T1 = the counter from packet decoder

If T1 is 74, the acceleration is equal to zero. If T1 is greater than 74, the acceleration is positive, and if less than 74, the acceleration is negative. The zero mark is set at T1 = 74 and not (5000/2/32 = 78) is because the accelerometer requires an offset. When the RC car not moving, the accelerometer is transmitting an impulse width of 4.6 ms, which translate to 74 in the counter.

The data obtained from the acceleration register is an 8 bit std_logic_vector. The first bit (most significant bit) indicated whether the data represents the x-axis acceleration or y-axis acceleration. A one represents y-axis acceleration, and a zero represents the x-axis acceleration. The remaining bits represent the acceleration of the RC car. An error in acceleration data collection occurs when the remaining 7 bits are "1111111". In this case, the data is ignored, and the old value of the acceleration is displayed on the monitor.

Once the acceleration of the RC car has been determined, the value of the acceleration has to be divided (using lpm_divide and divide by 10) into two digits (the ones digit and the tenth digit, i.e. 24 is divided into 2 and 4). These two digits are then passed into the VGA.vhd to be displayed on the monitor.

Attempts have been to reduce the total logic cells used. The original design used lpm_divide and lpm_multi to calculate the acceleration, but it used too many logic cells. The calculation is then changed to converting the input signal into integers, computing the acceleration and then converting the answer back to type signal. This method reduced the total logic cells used.

Challenges: The MAXPLUSII won't allow the division of any number that is not base 2, if lpm_divide is not used. Lpm_divide is not used to reduce the number of total logic cell used. Because of this problem, square root function was not able to execute. Also, the timing of the reading the data is critical. When the lpm_shiftreg is writing into the acceleration register, it is important for the accel entity not to read the shift register. The problem of refreshing and displaying on the data onto the VGA monitor provides a concern initially. The original plan is to update the acceleration data on the VGA monitor whenever there is a change. This will occur every two packets (7.42 ms). This is too fast for the human eyes to read the data. Thus, the counter of 30 is added.

Distance (*distance.vhd)*
The data is an 8 bit std_logic_vector from the optical encoder is used to calculate the distance traveled by the RC car. The optical encoder itself will generate 128 pulses per one revolution. Since the packets are sent every 3.72 ms, only the sixteenth packet is used in the calculation. Sixteen was picked, since it was easy to implement a 2^n-bit counter. To accomplish this, a finite state machine is used. The data is continually sent with each packet but a 4-bit counter holds the process in a counting state until it reaches "1111" then proceeds to the next state. The new value is then added to the previous value, since the total distance traveled is the desired result. The sum is then divided by 128 to find the total number of revolutions. A wheel is physically attached to the encoder; this wheel has a circumference of 16 cm. To obtain the distance, the circumference of the wheel is multiplied with the total number of revolutions.

$$Dis\tan ce = \frac{\#\,pulses}{128\,pulses/rev} * 16cm + (previous\_dis\tan ce\_value)$$

A register was implemented to hold the previous distance value to calculate the total distance traveled after an initial reset. Once the distance of the RC car has been determined, the value is divided by 100 to obtain a hundredth digit and the remainder is divided by 10 to obtain a tenth and ones digit. E.g. (345 is divided by 100 = 3 and remainder 45, then 45 is divided by 10 = 4 remainder 5). The three digits are then passed to the *VGA.vhd* to be displayed on the monitor. An error in distance data collection occurs when the remaining 7 bits are "1111111". In this case, the data is ignored, and the old value of the acceleration is displayed on the monitor.

One of the difficulties in the design of the distance was to make the system wait for the valid data and then complete the calculation. Much of the initial code had to be redesign since all the calculations were done with lpm_mult and lpm_divide. Using these functions required twice as many logic cells and extremely long compile and simulation times. Now the data is converted to integer before the calculation and then back to std_logic_vectors after. This reduces the number of logic cells need and simulation time.

Velocity (*velocity.vhd)*

The instantaneous velocity is also obtained from the optical encoder. From the distance design, it was stated that only the sixteenth packet had the valid optical encoder data. In the same distance finite state machine the output signal is sent to the velocity entity with the valid encoder data. Once the data is sent the velocity calculation can proceed. First, the data is divided by 128 to obtain the number of revolutions, and then it is multiplied by the circumference of the wheel that is 16 cm. This gives the instantaneous distance, to calculate the velocity; a reference time value is needed. The time is taken from packet transmission time of 3.72 ms, which is then multiplied by 16 for every valid data. So the total time is 3.72ms*16 = 0.05952, take the reciprocal and approximate to 16 1/s. The final velocity is the distance calculated multiplied by 16 and the result is in cm/s.

$$Velocity = \frac{\left(\frac{\#\ pulses}{128\ pulses\ /\ rev} * 16\ cm\right)}{0.0390\ \sec * \ 16\ packets}$$

Again, as in distance, once the velocity of the RC car has been determined, the result is divided by 10 to obtain a tenth and ones digit. E.g. (45 is divided by 10 = 4 remainder 5). The two digits are then passed to the *VGA.vhd* to be displayed on the monitor.

Many of the problems faced in the velocity calculation were carry over from the distance calculation since distance is an essential component of velocity. Again, the multiplication and division were initially implemented with LPM modules, but were change to reduce the number of logic cells.

Direction and Proximity Warning (direction.vhd)

The data input to the entity *direction* is an 8-bit std_logic_vector. This data is used to calculate both the heading of the RC and the proximity warnings. Both the heading direction and the proximity warnings are updated every packet (every signal from the packet decoder). The input to this entity is the clock (40ns), the reset signal, an enable signal and the data input from the package decoder. The data input from the package decoder is first written into an direction-proximity register every 3.72 ms. Whenever the package decoder writes data into this register, an enable signal will be set to disable the *direction* entity in determining the heading of the RC car. The old value of the acceleration will be used to display onto the VGA monitor. When the package decoder finishes writing to the direction-proximity register, the *direction* entity will obtain the data from the direction-proximity register and calculates the heading of the RC car. If the packet data contains all '1's, the data is discarded and the previous data is used to display on the VGA monitor. This is due to the data from the compass is logic high. Therefore, there is no possible way for the data to contain all '1's.

As mention before, the data stored in the direction-proximity register contains an 8-bit std_logic_vector. The bits that contain the direction are the last 4 bits. The order of bits is north, east, south, and west (with north in the most significant bits). An if - else statement is used to separate this 4 bits and converts them into a 4 bits std_logic_vector (000 = north, 001 = northeast, 010 = east, 011 = southeast, 100 = south, 101 = southwest, 110 = west, and 111 = northwest). This 3 bits vector will then be passed in to the VGA.vhd to be displayed onto the monitor.

The detection of proximity objects is done at the same time the direction heading of the RC is calculated. When the 8-bit vector is passed to the *direction* entity, the 4 most significant bits of the vector contains the information about the proximity objects. These four bits represent the proximity detection at the front, left, right, and back of the RC car. A zero represents an object is closed to the RC car, and a one means that the car is safe from crashing. When these 4 bits are separated, they will be passed into the VGA.vhd to be displayed onto the monitor.

VGA

Initially, the VGA must synchronize both the vertical and horizontal direction (*syncgen.vhd*)[1]. This is important because not all pixels on the VGA are allowed to be updated at the same time. The pixels on the VGA are updated one at a time from left to right and from top to bottom. This is accomplished by *count_xy.vhd*[1]. These two files essential drive the refresh sequence for the VGA. The rest of VGA is broken into two sections, static and dynamic objects. Static is an object that does not change and is continuously displayed on the monitor, while dynamic are objects that change.

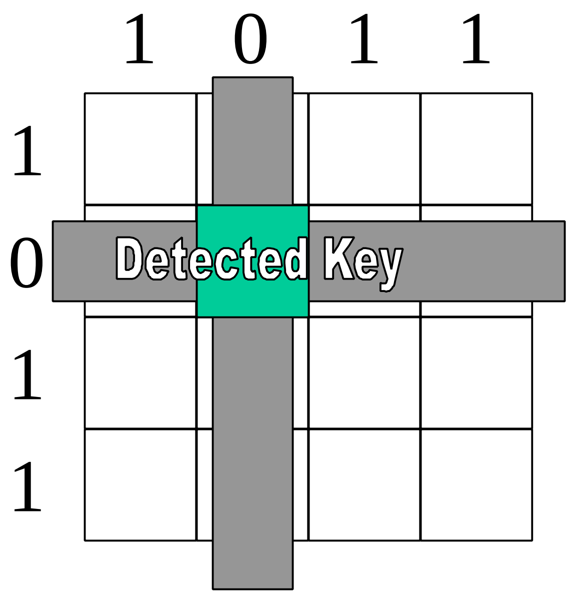
The static objects are implemented first, starting with a blue background (*bkgd.vhd*). This entity uses *count_xy.vhd* to cycle through all the pixels on the screen and sets the output blue pin high, while setting the green and red pins to low. To display letters on the screen (*display.vhd*), for example (Driver's Ed, our project name), two lpm_rom were used. The first lpm_rom holds a *char_set.mif* file, which the individual characters are in binary format. Second lpm_rom holds a different file called *d_text.mif*, which provides the ability to group individual characters together to form words. The *d_text.mif* contains 3 columns. The first column represents the binary address of the words within the .mif file. The second column uses the octel system to write the address of each characters I the *char_set.mif*. The last column is the actual written words. Again, the entity *count_xy.vhd* is used to cycle through all the pixels on the screen. A different process checks the current column address and row address to see if a match is made to output a certain string in the *d_text.mif* file. Once a match is found, then a specific message is selected and a color for the output is chosen.

Dynamic objects are implemented in the same manner as the static objects with minor adjustments to the input process (*combine.vhd*). Dynamic objects include such items as the proximity warning, keypad input, compass heading (N,E,S,W) and the digits for acceleration, velocity and distance. These objects are refreshed when the new data values are calculated and passed to the VGA component. Two counters cycle through the x and y position on the display and another process checks the current column and row address. Once the same column and row address is found, the new data from the requested calculation is taken and displayed on the monitor. The update of the dynamic characters is relatively straightforward. Since the data is passed in as a signal, the signal can be used indirectly to update the display. For example, if the input signal contains the number 4, an offset is added to the signal so that the new signal represents the location of the number 4 in the *char_set.mif*. Essentially, the *char_set.mif* becomes a lookup table for dynamic objects.

Keypad Interface Code

The keypad interface was designed from scratch using a strictly Moore Machine architecture. This is a radical departure from the keypad code from the application notes where the hybrid Moore/Mealy machines with falling edge or double edge clocking were implemented. These application notes were used as a starting point in the design although the implemented code is significantly different. This was because the hybrid architectures did not perform well when implemented on the Altera FPGAs.

The keydecoder used a common row and column-scanning algorithm. The state machine began by driving all of the columns to a low value and detecting if any of the rows, which are weakly pulled high with 4.7kΩ resistors, were driven to zero. If any one row was driven to zero, that would mean that a key was depressed. The state machine will then wait several clock cycles for the keypad-bouncing transients to die down and then proceed to determine which key was pressed. It does this by shifting a zero through each column while keeping the other columns high. When it detects a zero in a particular row, the intersection of the column driven to zero and the row that has a zero as its input will yield the desired key. The keyscanning routine then waits for the key to be released by driving all the columns to 0 and waiting for all of the rows to go to one. The keyscanning routine then starts from the beginning. The resulting keypress is latched to an output register where the value is held until another keypress is detected. Furthermore, the keyvalid signal will go high for 1 clock period following the detection of a valid keypress. The recorded keyvalue will be output to a binary to 7-segment decoder and displayed on the VGA screen as well.



A simplified Finite State Machine Diagram is as follows. The keydetection phase for Columns 2, 3 and 4 have been omitted since it is very similar to the key detection of Column 1.

The concepts for the implementation of the VGA took awhile to grasp. The student application notes provided a basis to start from but not enough to generate the desired output initially. Much of the testing was through trial and error. Adding to the problem, the VGA code would not compile and gave a pointer error on the computer stations in CEB 342. The VGA code would only compile on the workstation in CEB 540, therefore we were constantly running up and down to compile and view the results.

Estimation/ Measurement of Total Logic Cell used

Base Station

Total Logic Cells Available: 1152 Logic Cells (EPF10K20RC240-4)

All the essential features, as described in the achievements section, have been implemented and simulated. Due to logic cell constraints, the keypad functionality is moved to the other FPGA available on the UP1 development board. The total logic cell usage from the report generated by MAX+plus II is **1123/1152 (97%).**

Total Logic Cells Available: 128 Logic Cells (EPM7128SLC84-7)

This FPGA implements the design for user interface using a keypad. The total logic cell usage from the report generated by MAX+plusII is **60/128 (46%).**

RC Car

Total Logic Cells Available: 128

All the essential features on the RC Car FPGA have been designed and simulated in MAX+plus II. From the report generated, the number of logic cells used is **115/128 (89%)**. Some additional features that could not be implemented due to fitting issues include:

1) An error detection and correction (EDAC) scheme using a linear feedback shift register that randomly compares bits to check for errors
2) A more accurate accelerometer pulse width count. Currently, the acceleration count is divided down be 32 before it is sent to the base station. This introducing some round off error as the acceleration is only accurate to ±31 counts.

Maximum Speed

Base Station

The maximum clock frequency of the base station FPGA as measured by the Registered Performance Timing Analyzer in MAX+plus II was 3.56MHz. However, this appears to be incorrect as the system uses multiple clocks. The use of multiple clocks was necessary since the RF receiver receives the data at 7.1825kbps while the calculation and VGA display modules operate at the system clock of 25.175MHz. Simulation was completed with a clock frequency of 25.175MHz.

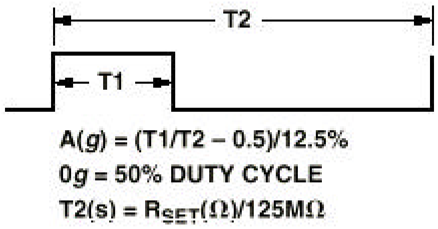
RC Car

The maximum clock frequency of the RC car FPGA as measured by the Registered Performance Timing Analyzer in MAX+plus II was 37.31MHz. This is much faster then the required system performance. There is no need to increase the system performance since the system operates at an acceptable frequency already.

Hardware Component Integration

Accelerometer Design

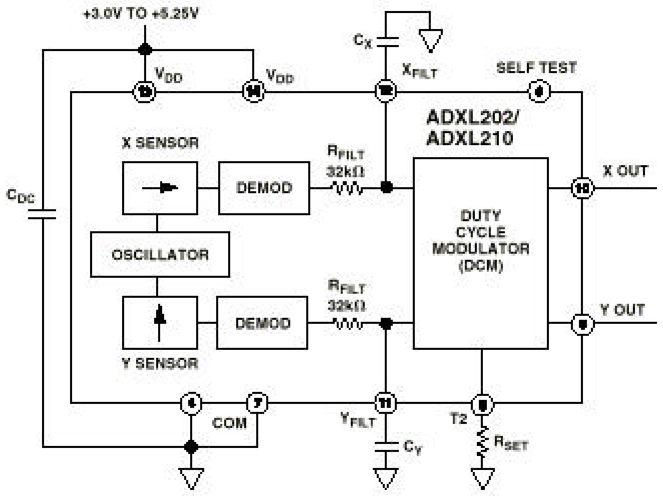
The ADXL202 is a dual axis accelerometer capable of measuring both positive and negative accelerations to a maximum level of ±2g. For each channel an output circuit converts the acquired analog acceleration measurement into a duty cycle modulated (or Pulse Width Modulated) output. When the accelerometer is at rest and completely horizontal to the earth, both axes should normally produce a 50% duty cycle. Due to processing considerations in the base station, to be discussed later, only the x-axis was used in this project. Thus only straight-line kinematic acceleration was displayed accurately on the VGA screen. An example of the accelerometer waveform is as follows:



The acceleration is proportional to the ratio of the pulse width and the period or T1/T2. The scale factor, as specified in the data sheet is 12.5% Duty Cycle change per g. Therefore, the maximum de-acceleration will be measured as a 25% duty cycle while the maximum acceleration will be measured as a 75% duty cycle.

Before the accelerometer could be used, it had to be configured for operation using several external components.

The functional diagram of the accelerometer from the Analog Devices data sheet is as follows:



First a small decoupling capacitor C_{DC} of 0.1uF was needed to decouple the IC from noise on the power supplies.

Next, the bandwidth of the accelerometer needed to be chosen. The bandwidth determines the response time of the accelerometer. Increasing the bandwidth decreases the response time but also increases noise the occurrence of white noise. According the data sheet the bandwidth is selected using an appropriate low-pass filtering capacitor. The equation that governs the bandwidth is simply

$$F_{-3dB} = \frac{5mf}{C_X}$$

Since the motion of the RC car was relatively slow, it was decided that using the lowest recommended bandwidth would be suitable for our application. Using a bandwidth of 10Hz resulted in a low-pass filtering capacitor value of 0.5uF. Choosing a standard value of 0.47uF met our needs.

The Duty Cycle Modulated Period (sample rate) was set by a single resistor R_{SET}. Since, as will be discussed later, the FPGA must gather data from different sensors and transmit the entire packet at a maximum rate of 20kbps; the period was set as high as possible. The equation governing the period was

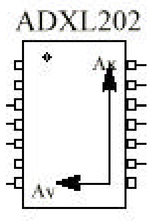
$$T2 = \frac{R_{SET}}{125M\Omega}$$

Since the maximum recommended period is 10 ms, we decided to use a 1.2MΩ resistor. This resulted in a theoretical period of 9.6ms and thus a sampling rate of 104Hz.

The accelerometer has characteristics of white Gaussian noise that contributes equally at all frequencies. The noise is proportional to the square root of the bandwidth of the accelerometer. Through statistical methods that are beyond the scope of this report, the resolution accuracy one should be able to achieve a resolution of approximately 1.9mg.

However, the resolution of the acceleration measurements are also limited by the frequency and number of bits of the counter. Because of logic cell limitations on the RC Car FPGA and the maximum speed of the RF transmitter, a counter frequency of 500 kHz was chosen. Furthermore, an effective 7-bit counter was used. In order to count a pulse that might last a maximum of 7.5ms, the FPGA will use a 12-bit counter. Then the counter value will be divided by 32 to be placed on the 8-bit data bus. The effective resolution is 51.2mg.

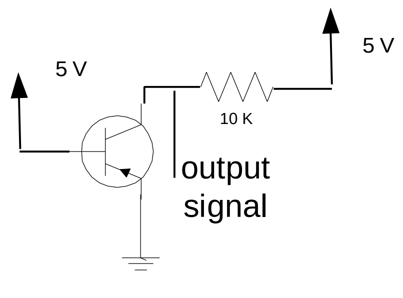
The accelerometer was mounted on the RC Car with the following orientation:
Front of Car



However, as will be discussed in the IC test measurements section, the accelerometer was mounted at a tilt.

Compass

The digital sensor No. 1490 is used as the compass sensor for the RC car. The sensor magnetically indicates the four Cardinal (N, E, S, W) directions, and by overlapping the four Cardinal directions, shows the four intermediate (NE, NW, SE, SW) direction. These signals are generated by four Hall effect IC’s. The following circuit is attached to each Hall effect.

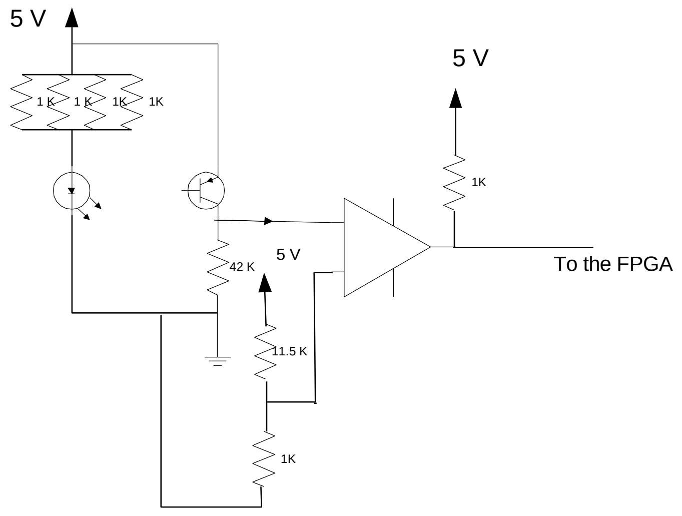


The output signal of the hall-effect sensor is a logic low signal. When the particular hall-effect sensor is not detecting the magnetic north, the output signal is high. When it is detecting the magnetic north, the signal will go low.

Challenges: The biggest challenges related to the heading direction are using the compass sensor. The pins layer of the compass sensor is not in the format of an IC chip. It is oriented in the 4 sides with 3 pins a side (total of 12 pins). This requires bending of some pins. The power and ground pins of the sensor also poses some problems. The ground is always right beside the power pin. If bending of the pins is required, extreme cautious must be taken to ensure the ground and power pins do not come in contact and short circuit the sensor. The mounting of the sensor on the RC is major obstacle. At first, the sensor is mounted on the back of the RC car. When the sensor is being tested, it was discovered that the sensor always displays one direction. Later, it was found that the inductor coil from the RC motor which lies directly below the compass sensor creates a magnetic field which interfere with the sensor. The next location of the sensor to be tested is at the front of the RC car. This location is also proven to be inappropriate because it too close to the rest of the other circuits (accelerometer, RF transmitter etc). The current produces by the power and ground lines at the side of the circuit produces a magnetic field, which also interfere with the compass sensor. The final solution is to elevate the sensor above the RC car thus above the interference of the magnetic fields produce by the motor and the other circuitry.

Proximity Detection

The proximity detection sensor (QRD 1114 or OMRON EE-SF5) is make up of both the photo reflective sensors and the voltage comparator. The output signal of the combine sensor is logic low. Normally the photo reflective sensor will emit light. This light is bounced back into the photo reflective sensor if an object is in front of the sensor. The reflected light will cause the output signal to go low. The voltage comparator is used to drive the signal low after a threshold point. The threshold point in the voltage comparator is set to 0.6 V to increase the sensitivity of the proximity detection. This means that whenever the input voltage drops from 5 V to 4.4 V, the voltage comparator will output a 0 V signal. Two resistors, 11.5kΩ and 1kΩ, controls the threshold point. If the 11.5kΩ is reduce, the threshold point of the voltage comparator will increase.



Challenge: The only difficulty for the proximity warning is the distance of detection. At first, the sensors can only detect objects if they are less than 1 cm from the object. After purchasing a relatively more expensive ($1 compared to $6) sensors, and adding a voltage comparator, the detection range of the sensors increase to 5 to 6 cm. The new sensors play a significant role in the detection range. The front, left and right sensors are the more expensive sensors, while the back sensor is the original sensor. Although all 4 sensors have the same voltage regulator, the detection ranges are quite difference. The new sensors have a range of 5 to 6 cm while the original sensor has only 2 to 3 cm range (which have a significant improvement with the voltage regulator). An even more expensive sensor (~$60), with an amplifier can improve the detection range even further. But for the purpose of this project, we feel that the present sensors are justified.

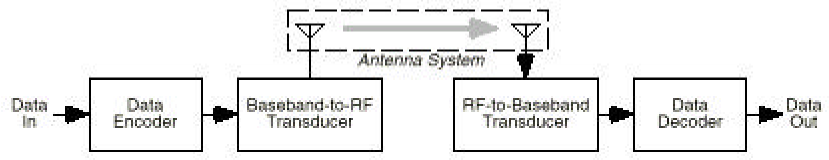
Optical Encoder

The optical encoder used was the Clarostat-Optical Rotary Encoder 600E, which outputs 128 pulses per revolution. It has an input power of 5V @ 30 mA and an operating speed range of 300RPM to 3000RPM.

Challenge: Mounting the optical encoder was also a problem. The optical encoder can in a non-standard package and was difficult to mount onto the RC car. The solution to this problem was to use Lego^{TM} to make a housing for the optical encoder and implemented a gear and chain system to turn a wheel. Still, we experienced some slipping with the optical encoder and wheel. Also, was mounted on the rear of the car creating a problem when backing up. With more time a better design could be implemented so that backing up the car would not be a problem.

RF Hardware Design

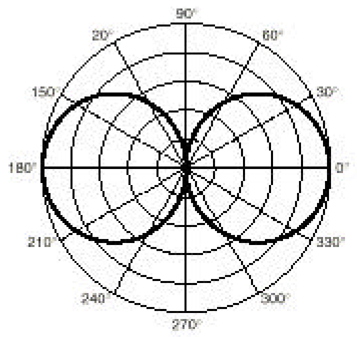
The RF Link was accomplished through the use of PCB mounting modules for both the RF transmitter and receiver. The RF transmitter module, the ABACOM TXM-418-F was an RF module that accepted serial digital data and modulated the incoming bitstream using Frequency Shift Keying at 418 MHz. The receiver module used was an ABACOM SILRX-418-A FM Superheterodyne Receiver which demodulated the FSK transmission and converted it into a baseband digital signal. A block diagram of the RF link can be represented as follows:



The RF transmitter/receiver pair was tested for correct function using the pulse width modulated outputs from the digital accelerometer. Oscilloscopes were connected to both ends of the RF link and it was observed whether a change of pulse width on the transmitted side was reflected in the same change in pulse-width on the received side.

Antenna Design

The simplest antenna type was used in this design. The antenna type used was the quarter-wave monopole cut from 24-gauge wire. The radiation pattern of a quarter wave monopole antenna is as follows.

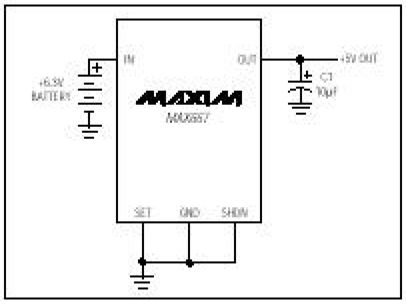


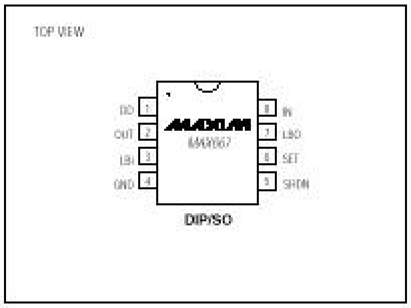
Using the information gained from several antenna design tutorials it was determined that resonance of a quarter-wavelength monopole occurs when its length is slightly less than a quarter-wavelength. The following equation was used to determine the required length of wire:

$$length = {}^{2808}\!/_{frequency(MHz)} = {}^{2808}\!/_{418} = 6.72 inches$$

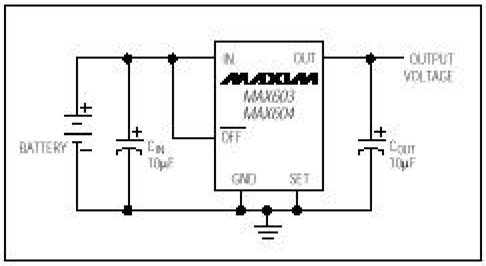
Voltage Regulators

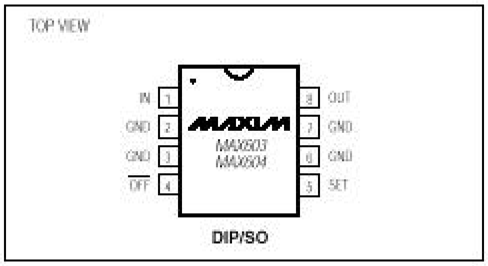
Two different 5 V Voltage Regulators were used. The first voltage regulator is the MAXIM MAX667 which supplies up to 250 mA of output current. Was used on the base station to regulate the 5V for the Keypad and RF receiver.





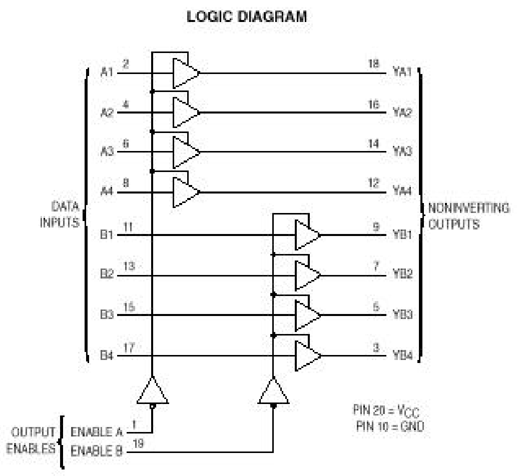
The second is the MAXIM MAX 603 which supplies up to 500 mA of output current. A 9.6V RC car battery powered the system on the RC car. This battery powered all the sensor components and the MAX 7000. This is reason we chose a different voltage regulator.

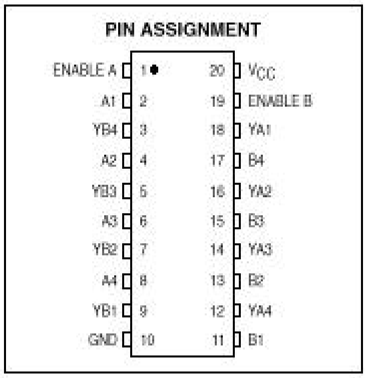


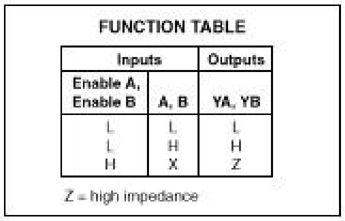


Buffers

A Octal 3-State Noninverting Buffer was used to protect the FPGA from static discharge and human wiring errors. The buffers were also used to provide a 5-volt drive needed since the FPGA could not only output 3.156V. The MC54/74HC244A buffers were used.





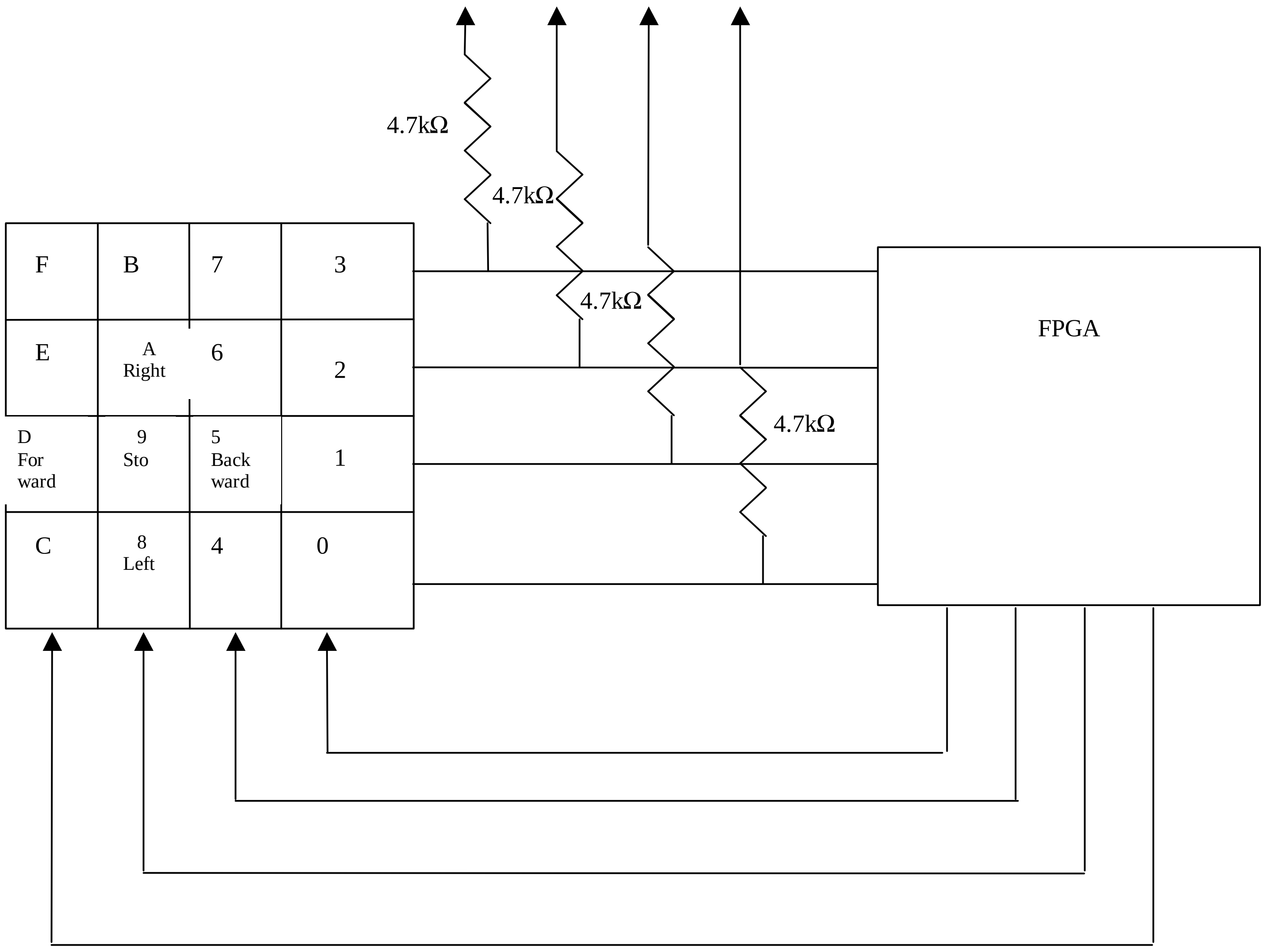

FUNCTION TABLE

Inputs		Outputs
Enable A, Enable B	A, B	YA, YB
L	L	L
L	H	H
H	X	Z

Z = high impedance

Keypad[2]

The keypad is a 4 by 4 Grayhill with 4.7kΩ pull-up resistors on the rows. A diagram below will illustrate how the keypad was wired:



Results of Experiments

Acceleration Experiment

Using a turntable, the acceleration was tested. The accelerometer circuit and the RC car's FPGA were placed on the turntable. When the turntable is turned on, the acceleration of the accelerometer displayed on the VGA monitor was recorded. The reason measuring the acceleration on a turntable is to measure a constant acceleration. When an object is rotating around a fixed point at a constant speed and a constant distance from the fixed point (radius), the object experiences two accelerations, the normal acceleration and the tangential acceleration. If all the required conditions (constant radius, constant speed around the fixed point) are true, the normal acceleration will be constant. Thus, the normal acceleration can be calculated by:

$$Acceleration = w^2 r$$

where ω = angular velocity in rad/s2
r = distance from the fixed point (radius)

This test will accurately test the acceleration of the accelerometer (since only one acceleration axis is used to display on the monitor). When the accelerometer is not moving, the acceleration display on the monitor is 0.3 m/s^2. Thus, there is an offset of 0.3 m/s^2.

RPM of turntable	**Radius**	*Angular velocity rad/s*	*Theoretic Acceleration m/s^2*	**Actual** *Acceleration (on monitor) m/s^2*	*Actual Acceleration (after offset) m/s^2*	*% error*
33	14 cm	3.45576	1.7	1.9	1.6	5.8
45	14 cm	4.7124	3.1	2.9	2.6	12.9

The % error may seem a little high, but the RPM may be too accurate. According to the turntable, the RPM can be set at 33 and 45. But when the FPGA and the accelerometer circuit are placed on the turntable, the RPM may be slower than stated. The combine weight of the FPGA and the accelerometer is heavier than the weight of a record, thus, this will cause the RPM to decrease, which will result in a smaller value for the angular velocity and the theoretic acceleration and in turn reduces the percentage error. Thus, the data from the accelerometer can be considered valid.

Distance Experiments

The optical encoder was used for both distance and velocity calculations. Testing the accuracy of the optical encoder to perform the function of calculating distance was done through the following series of tests.

Tests
1. Starting from rest, then proceed to 2 meters and stop, take reading from VGA
2. Starting from a constant velocity, then proceed to 2 meters and stop, take reading from VGA
3. Accelerating to starting point, then proceed to 2 meters and stop, take reading from VGA

These tests will accurately test the optical encoder for all possible conditions. The tests were conducted on a carpet surface to prevent the wheel, which the encoder is attached to, from slipping.

Test Number	*# of Trials*	*Actual Distance Traveled (m)*	*Expected Distance Traveled (m)*	*% error*

1	1	1.79	2.00	11.7
	2	1.66	2.00	20.5
	3	1.56	2.00	28.2
	4	1.89	2.00	5.8
	5	1.88	2.00	6.4
	Avg	1.76	2.00	13.6
2	1	1.89	2.00	5.8
	2	1.65	2.00	21.2
	3	1.99	2.00	0.5
	4	2.07	2.00	3.4
	5	1.97	2.00	1.5
	Avg	1.91	2.00	4.7
3	1	1.66	2.00	20.5
	2	1.89	2.00	5.8
	3	1.94	2.00	3.1
	4	1.88	2.00	6.4
	5	1.72	2.00	16.3
	Avg	1.81	2.00	10.5

From the results it can be concluded that the second test proved to be the most accurate for measuring distance. In the first test the car started from rest. The optical encoder seem to behave poorly under slow speeds (>0.5 m/s). Also some slipping could have occurred to produce the higher percent error. In the accelerating tests, the car was a little hard to control and to keep in a straight line. This was the main cause of the error. Finally, the test at constant velocity provided the least amount of error since the car was in motion and the encoder was already moving it gave better results. Still, some error was due to keeping the car at a constant speed and keeping going in a straight line.

In a separate test, a turntable was used to calculate the distance traveled. The wheel was placed on the edge of the turntable and was 14 cm away from the center. Rotating the turntable 5 times resulted in a total distance traveled of 4.40 meters.

$$\text{Distance} = 5 * (2 * \pi * 0.14 \text{ m}) = 4.40 \text{ meters}$$

The following is the results of this test:

Trial Number	*Actual Distance Traveled (m)*	*Expected Distance Traveled (m)*	*% error*
1	4.32	4.40	1.9
2	4.42	4.40	0.5
3	4.37	4.40	0.7
4	4.27	4.40	3.0
5	4.51	4.40	2.4
Avg	4.38	4.40	0.5

These results demonstrated the accuracy of the optical encoder without the added human error of driving the RC car in a straight line.

Velocity Experiment

The velocity was tested in conjunction with the distance in test 2, where we tried to keep the velocity constant. The following data was obtain from the test:

Trial Number	*Actual Distance Traveled (m)*	*Expected Distance Traveled (m)*	*Time (sec)*	*Actual Velocity (m/s)*	*Expected Velocity*	*% error*
1	1.89	2.00	1.85	1.02	1.00	7.4
2	1.65	2.00	1.94	0.85	1.00	17.6
3	1.99	2.00	1.97	1.01	1.00	1.0
4	2.07	2.00	1.99	1.04	1.00	3.8
5	1.97	2.00	1.98	0.99	1.00	1.0
Avg	**1.91**	**2.00**	**1.95**	**0.98**	**1.00**	**2.0**

The overall data gather from the optical encoder was quite accurate with little error. The error in the calculation can be accounted for by the slipping of the wheel, keeping the car straight, and human reaction time for the stopwatch. The percent error is still within reason and therefore the data is valid.

Proximity Experiment

The proximity detection measurement is done by placing a ruler on the table underneath the sensor and measuring how far the object is before the sensor detects the object and display the warning message on the monitor. 3 tests (white surface, reflective surface and non-reflective surface) are done with the room's lights on, and one test (white surface) is done when the room is dark.

	White Surface	*Reflective Surface*	*Non-Reflective Surface*	*White Surface*
Front Proximity	5 cm	5.5 cm	3.5 cm	4.5 cm
Left Proximity	5 cm	5.5 cm	3.5 cm	4.5 cm
Right Proximity	5 cm	5.5 cm	3.5 cm	4.5 cm
Rear Proximity	3 cm	3 cm	1 cm	2 cm

Direction Experiment

Comparing the display on the monitor with an actual compass does the test of the compass sensor. The results show that the compass sensor is working properly.

VGA Experiment

The VGA testing is done by trail and error. The VGA program is downloaded onto the UP1 and display onto the monitor. The location of the words and pictures are then viewed and adjusted if necessary.

Base station Experiments

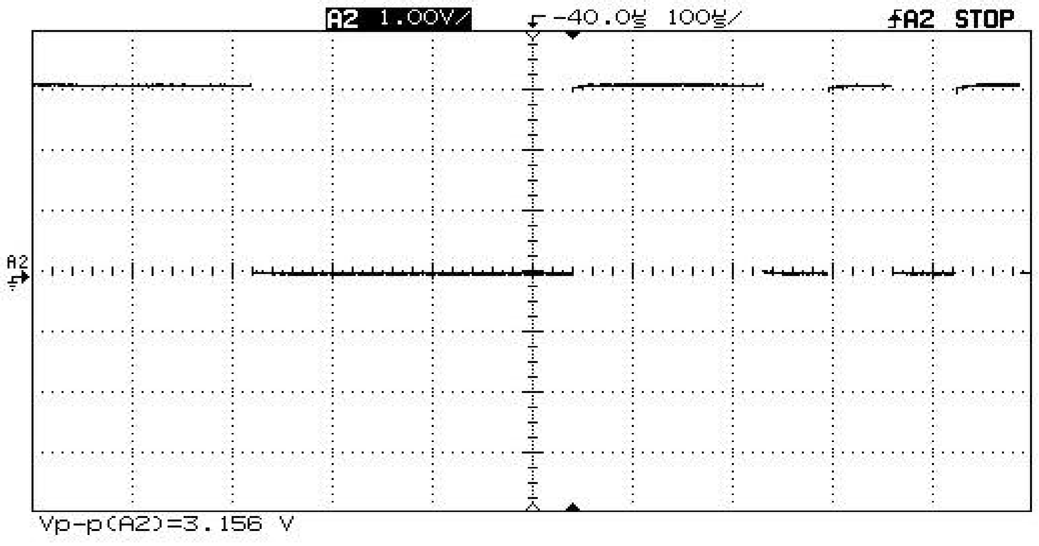
Any changes to the default setting in synthesis of speed vs. area trade-offs resulted in over 100% total cell usage and therefore would not fit into the Flex10K chip. Different settings of global synthesis style also resulted in over usage of cells. Therefore, no changes were made to the settings. The critical path is limited by the external hardware, which are the RF receiver and the VGA monitor. Therefore, no formal calculation could be done.

RC Car Experiments

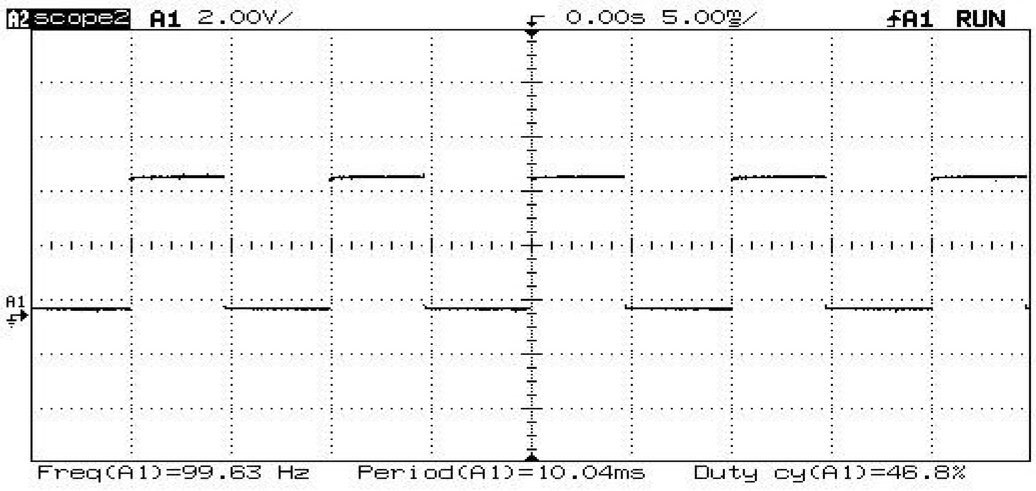
A noticeable reduction in the usage of logic cells was observed when adjusting the synthesis options. Initially, the speed vs. area optimization level was set to 0 (for most optimized area synthesis). When normal synthesis style is selected, the project would not compile. When fast was selected the project compiled but with about 92% logic usage. When the speed vs. area optimization level was set to 2 the project blocks was reduced to 89% usage. Other levels were attempted but did not yield improved results. The main objective for using the different compile options was for the most efficient optimization of area rather than attempting to achieve maximum speed. This is mainly due to the limited number of logic cells available on the MAX7K. The maximum speed was not a great concern as fast throughput was not required. Furthermore, external components such as the RF transmitter with a maximum transmission rate of 20kbps set a limit to the maximum throughput of the FPGA.

IC Measurements

The following diagram shows how the FPGA cannot drive up to the 5V rail when sourcing current to even a single CMOS load. Therefore a current buffer was deemed necessary. The peak to peak voltage is 3.156V

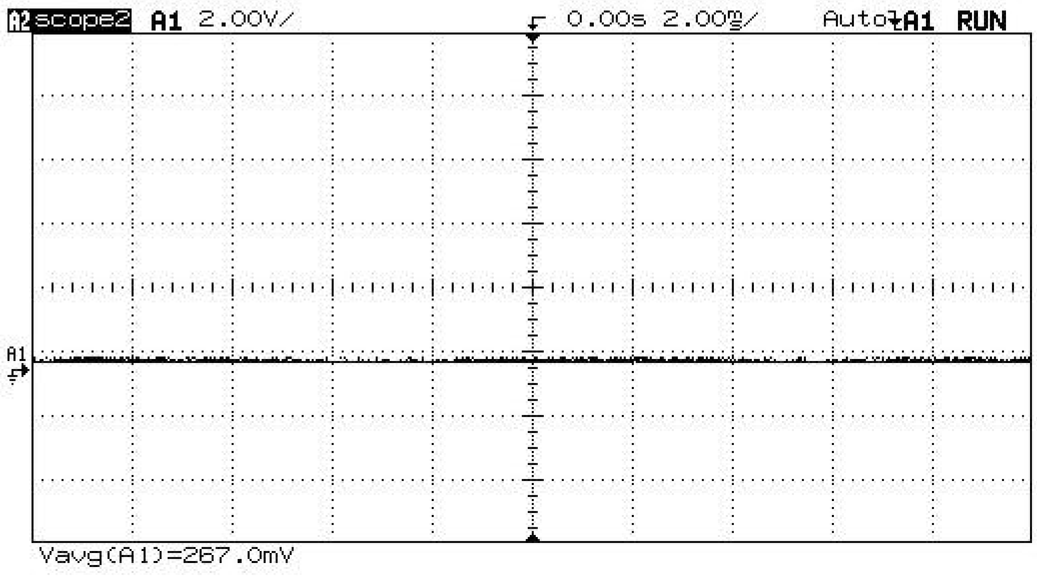


The next measurement taken is the acceleration in the x direction. This measurement provides us with insight into the operation of the accelerometer.

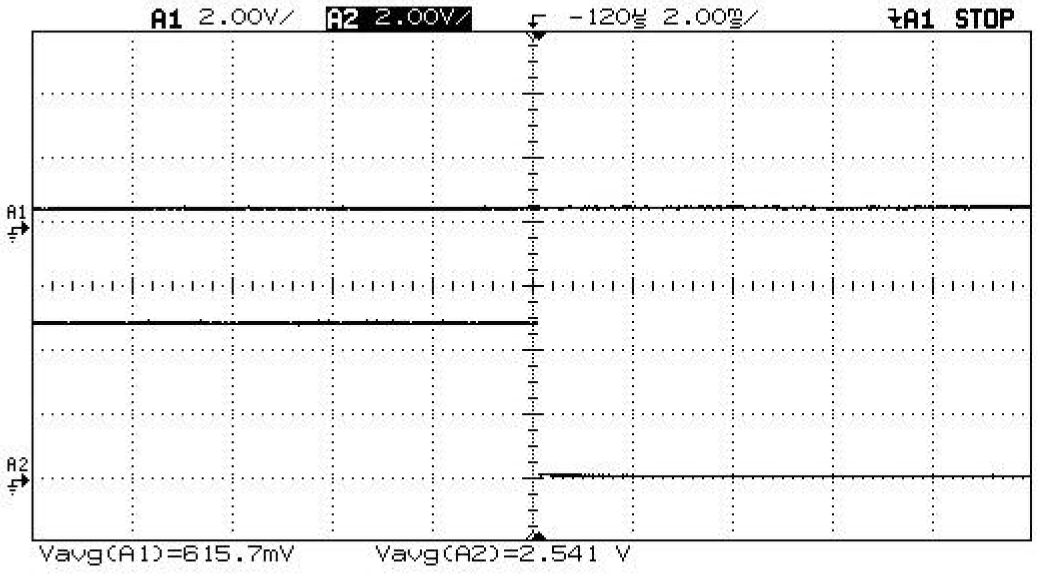


The sampling rate of the accelerometer was 99.63 Hz with a corresponding Duty Cycle period of 10.04 ms. The duty cycle, length of the high pules, at rest was 46.8%. This deviated from the theoretical 50% duty cycle for an accelerometer at rest because of the angle at which the accelerometer was mounted. The frequency and period can be altered with a change in resistor value. This is explained in the Hardware Design section.

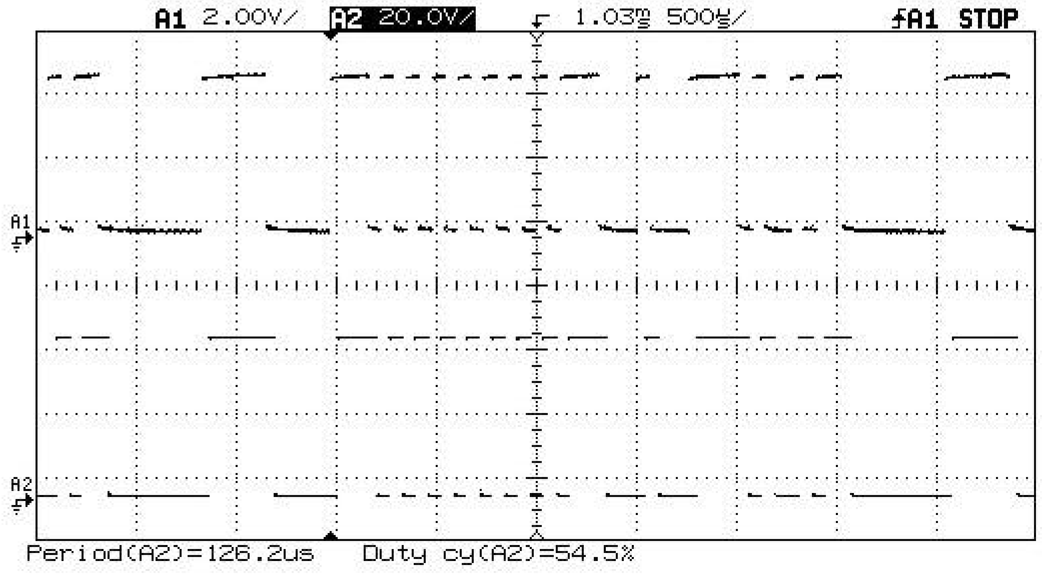
The following two diagrams show the operation of the proximity sensors and voltage comparators. The first diagram illustrates the operation of the proximity sensor when no object is detected. The effects of the fluorescent lights cause the deviation from 0V. As described elsewhere in this document, the voltage level increases when the infrared light reflects off of an object and is detected by the phototransistor circuit. However, the voltage will not reach the threshold needed by the FPGA to detect a logic high until the object was very close to the proximity sensor circuit.



The voltage comparators where used to increase the effective range of the proximity sensors. With the resistor divider network as shown in the Hardware Section, the threshold trip point where the comparator would go from high to low and signal to the FPGA that an object was detected is at 615.7mV. This corresponded to an object being within 5cm of the proximity sensor.



The final diagram shows the how the packet that is transmitted from the RC car is received by the base



station.

From this diagram we can see that the received signal is slightly delayed when compared to the transmitted signal. However, the bitstream was recovered exactly. This test was run with both the transmitter and receiver within 1m of each other.

References:

Student Application notes:
1. VGA: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/98w/dicerace_video_display
2. Keypad: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/99w/keypad/keydecoder.html

3. http://www.ee.ualberta.ca/~ee480/hardware/UA7K_UsersManual.pdf
4. http://www.ee.ualberta.ca/~ee480/hardware/UA7K_Helpsheet.pdf
   Written by Steven Sutankayo and Curtis Wickman

Datasheet

5. Abacom Technologies:	TXM-4xx-F (5V, 20000bps Transmitter)
6. Abacom Technologies:	SILRX Series UHF FM Superhet Receiver Module
7. Analog Devices:	ADXL202/210, Accelerometers with Digital Output
8. Clarostat:	Optical Rotary Encoders Series: 600, 601H, 601V
9. Dinsmore:	Digital Compass Sensor
10. Maxim:	MAX667, +5V/Programmable Low-Dropout Voltage Regulator
11. Maxim:	MAX603, +5V/Programmable Low-Dropout Voltage Regulator
12. National Semiconductor:	LMC6772 Dual Micropower Rail-to-Rail Input CMOS Comparator With Open Drain Output
13. Motorola:	MC54/74HC244A Octal 3 State non inverting Buffer/Line Driver/ Line Receiver
14. Grayhill:	12/16 Button Keyboards

15. Optoelectronics:	ORD1113 Reflective Object Sensor
16. Micrel:	MICRF001 Antenna Design Tutorial
17. Omron:	EE-SF5-B Compact Reflective Phototransistor Output Insusceptible to External Interference Light
18. RFM:	Application note on encoding and decoding RF transmissions and packetizing information

Appendix

(not included in web copy)