TYPE R

Robot A, Navigation Subsystem

Jonathan Andrews
Matthew Artelt
David Catanach
Ryan Kruse

New Mexico Institute of Mining and Technology
Electrical Engineering Department

Prepared For:
Dr. William Rison
Dr. Kevin Wedeward

May 6, 2002

Abstract

One requirement to obtain the Bachelors of Science in Electrical Engineering at New Mexico Tech is that students complete "Introduction to Design," EE382. This semester's task was to build a golfing robot. The class built two robots, A and B, with each robot divided into four subsystems: Navigation, Ball/Hole Location, Smart Chassis, and Communications. We were assigned to the Navigation Module of Robot A.

The following report will outline the design and equipment used to complete the Navigation Module. The requirements of the project stipulate that the module use a GPS to determine the robot's location and a compass to determine the heading. This information will then be used to calculate the angles and distance the robot must move to find the ball and hole on a green. The bulk of the module was finished few weeks prior to the end of the term, and the remaining time was spent assisting other modules and testing the communications protocols.

The reference section of the report (Appendix D) contains links to the data sheets and handbooks that go with the hardware that we used. They were indispensable in setting up much of the hardware.

Why "Type R"?

Why "Type R?" This comes from the automotive world, specifically Import Tuning. A sub-culture, if you would, of Import Tuning is called "Rice." One popular aspect of Rice is adding stickers, emblems, etc. of faster cars to "normal" cars such as a Honda Civic. Among Ricers, as people who perform such acts are called, of the most popular "rices" is putting a Type R sticker off a race version Acura Integra onto their cars. This is done to intimidate others.

Golf, like auto racing and sport tuning, is a sport of mental challenges, and the more distractions and intimidations that a player can give his opponent, the worse the opponent will do, in theory. Thus, we chose "Type R" to intimidate the opposing robot player, and improve our chances of playing a better game.

The actual "Type R" on the back of the module was not originally intended to be painted onto the Robot. However, one long night in the lab resulted in high stress levels and a need to release that stress. Thus, "Type R" was born.

Table of Contents

Index of Appendices

Index of Figures and Tables

Figures:

Tables:

Introduction

Scope

The overall goal of this project was to design and build a robot capable of finding a golf ball given GPS coordinates, transport the ball to a hole, given GPS coordinates and drop the ball into the hole. Our team has been tasked with building the navigation module of Robot A. The requirements for the Navigation Module are as follows:

1. Determine the current location of the robot using a GPS
2. Calculate the distance to travel to the ball/hole
3. Determine the current heading of the robot
4. Calculate the angle of rotation for moving to the ball/hole
5. Interface with the other modules to send/receive commands and robot status updates

We added the following additional criteria to the requirements to make the design more robust and compact:

1. Use a digital compass to determine the heading
2. Use a single HC12 for the module
3. Implement tilt correction for 2-axis compass errors

Purpose:

The purpose of the paper is that given the appropriate prerequisites, an engineer could accurately reproduce or debug our Navigation Module. Prerequisites include:

1. In-depth knowledge of the Motorola 68HC12 micro-controller
2. Digital and Analog circuit design principles
3. Hardware design knowledge
4. Knowledge of the C programming language and Cosmic Compiler

Module Overview

The Navigation module is the center of overall robot design. As such, there are no direction connections between the other modules, except for the regulated power supplied to all by the Chassis module. Figure 1, below, shows this design.

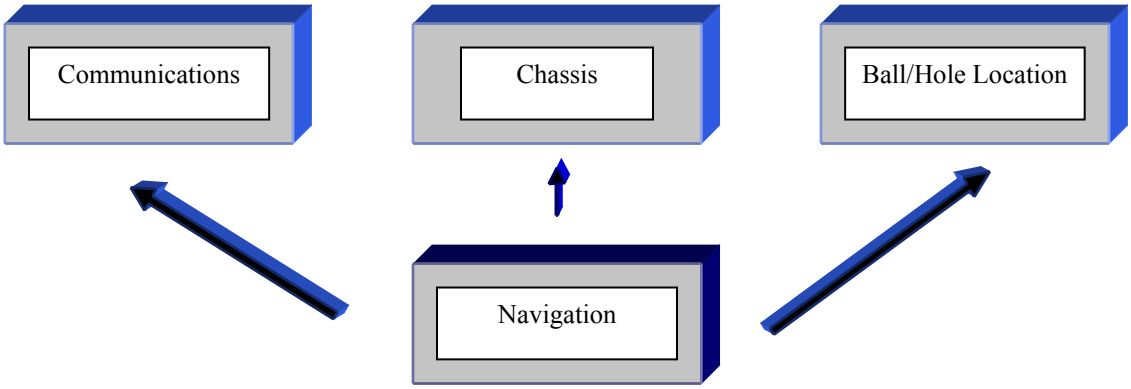


Figure 1: Block Diagram of Robot Modules

The Navigation Module itself is centered on a Motorola 68HC12 microcontroller and evaluation board. The Navigation overview is shown below in figure 2.

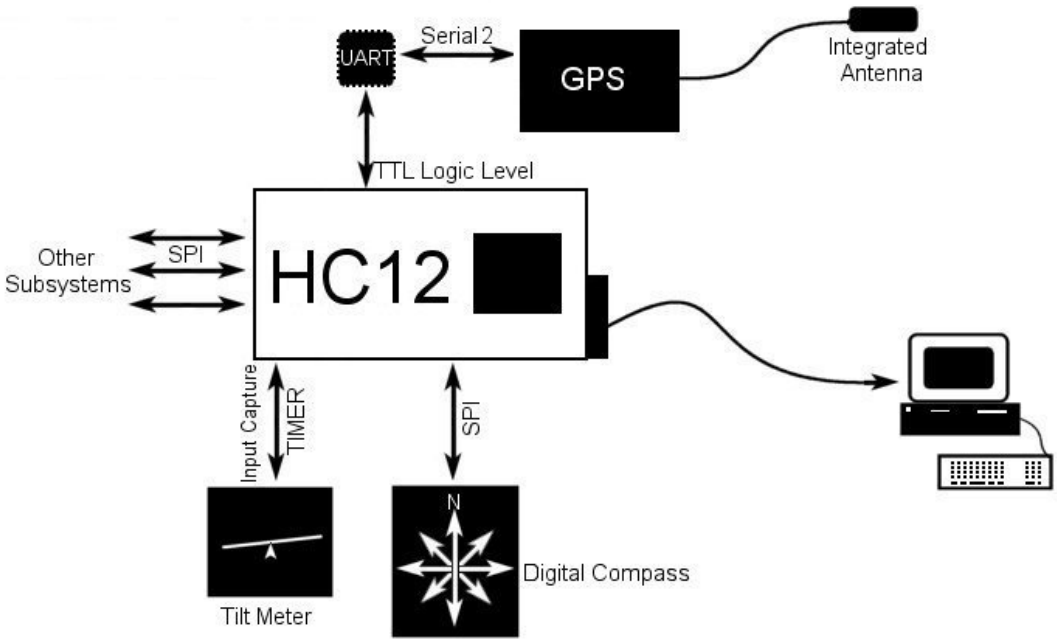


Figure 2: Navigation Block Diagram

Hardware Design

As with any robot, the choice of hardware is critical to proper operation and implementation of the given design. We had the following hardware included in our module, encased in aluminum housing, based on research and references from earlier in the semester.

1. Garmin GPS16: GPS receiver unit, used to determine the location of the robot.
2. PNI Vector 2X Digital Compass: Board-mounted digital compass, used to determine the robot's heading
3. Analog Devices Accelerometer: Mounted on a surplus board, used to detect the tilt of the robot and correct compass errors
4. National Semiconductor Universal Asynchronous Receiver Transmitter, used to add an additional serial port to the HC12 for GPS interfacing
5. RJ45 interface connections, used as the plug/socket connections between modules

Casing and Mounting

The original design called for a modular robot so that subsystems could be switched out between robots. Staying with this design stipulation, we originally decided to mount all of our components inside of a box of some type. This allowed us to easily disconnect our module from the robot and move it to another robot. Although this design stipulation was eliminated after the first few weeks of the semester, we decided that our robot would still maintain this modular design. Upon conferring with the chassis group, they decided to purchase small boxes for all of the other subsystems. They supplied us with a 3" x 5" x 7" aluminum box that would ultimately be mounted onto the top of the robot, as we needed a high vantage point for the GPS receiver.

It was soon discovered that the GPS receiver could not be mounted in contact with any metal surface, as this reduced the reception of the receiver. To remedy this problem a mast was manufactured out of 2 inch PVC pipe. The mast held the receiver about 18 inches above the rest of the robot which we found was tall enough to ensure proper reception of all satellites, including the WAAS satellites. This is shown in figure 3.

Figure 3: The Navigation Module

Besides the HC12, we also had to find room for the digital compass and the accelerometer inside of the box. Our original concern was that the compass would be affected adversely by the metal box. We discovered, however, that since aluminum is non-ferric, we were able to mount both of the other devices inside the box without disturbing their functionality. Details of this mounting can be found in the compass and accelerometer hardware sections.

Once a suitable system had been engineered for communications between the other subsystems, the proper connectors could be chosen. We opted to wait until the program had been written before we decided what type of connectors to use. The original program required

seven lines to transfer data, including the handshake lines. There are, however, no standard connectors that use seven pins. Because of this, we decided to use a standard 8-pin connector to go between the modules. Searching around the lab found several found several spare RJ-45 female connectors and cable from a recent remodeling. These were to be thrown away, so we decided to implement them. Cables were built to run between the modules, and the RJ-45 female connectors were mounted on the outside of the box.

All components inside of our box were mounted with standard non-conductive standoffs. The compass had to be mounted with the front of the compass facing (see compass manual for diagram of compass board) the front of the robot, so we determined which way our box would be facing before we mounted that component. Use of a square edge and a ruler ensured that the compass was parallel with the side of the box. The same was done for the accelerometer.

The physical connections that were made to the HC12 can be found in the appropriate hardware discussion (UART, Compass, etc.) or can be seen in the schematics and block diagrams included in Appendix A: Schematics.

Garmin GPS16

Most all GPS units we found transmitted data via RS-232 protocol. Most also used RS232 logic levels, although some offered TTL as well. To accommodate both of these, and to allow for normal use of the microcontroller serial programming port, we decided that an additional serial port would be necessary.

National Semiconductor UART

After deciding on the Garmin GPS16 receiver, we had to determine how we would communicate with the receiver. Since the receiver sent out its information serially, we only had

three options of how to pick up what it was sending. Option number one was to attach another HC12 to our board via SPI and use the other board's DB-9 serial port connector. This would allow us to see the GPS information on our master board through the DB-9 connector to the EVBU. This option required us to use two HC12's, which was determined unnecessary as it took up too much room in our box. Option number two was to program our board, and then use the DB-9 serial port on our board to communicate with the GPS. This did not allow us to troubleshoot easily, as we could not connect to the EVBU and the GPS receiver at the same time. The third option (which is the one we chose) was to add an expansion serial port to the HC12 using a Universal Asynchronous Receiver-Transmitter, or UART. This allowed us to communicate with the GPS receiver and the EVBU without using an additional HC12 board.

Online research found that National Semiconductors offered a dual-channel UART in a 44-pin PLCC package, the PC16552D. We chose the dual UART chip, shown in figure 4, because of preliminary claims from the communications group was that they might want a serial

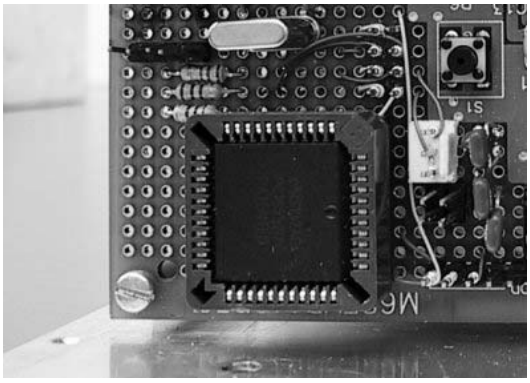


Figure 4: UART in Socket

port for communicating with us. This fell through, so we had an extra serial port channel in case something happened to the first one.

In order to talk to the serial port, we had to find an address range that would act as our serial port. The memory expansion attached to the HC12 used the address range from 0x1000 to

0x7fff. It also used addresses 0x400, 0x401 and 0x402 for the expansion I/O ports and data direction register. We chose the address of 0x410 and 0x420 to talk to the two serial ports. The address decoding was performed using the Altera expansion board. Since the chips used 8 bit communications and the memory expansion board uses 16 bit communications, we just only talked to the even address in the address range by ignoring the least significant bit. Altera Expansion Code for both memory expansion and UART can be found in Appendix C.

With the program written in Altera for address decoding, the last step was to wire wrap the UART to the HC12. Since the GPS receiver used RS232 voltage levels, and the HC12, memory expansion, and UART use TTL voltage levels, a MAX232 level shifter was used. A block diagram of the wiring can be seen in Appendix A. The connections to the pins of the HC12 were made using the wire-wrapping technique from EE308 Lab. The bottom of the HC12 after wire-wrapping is shown in figure 5, with connections as shown in Appendix A.

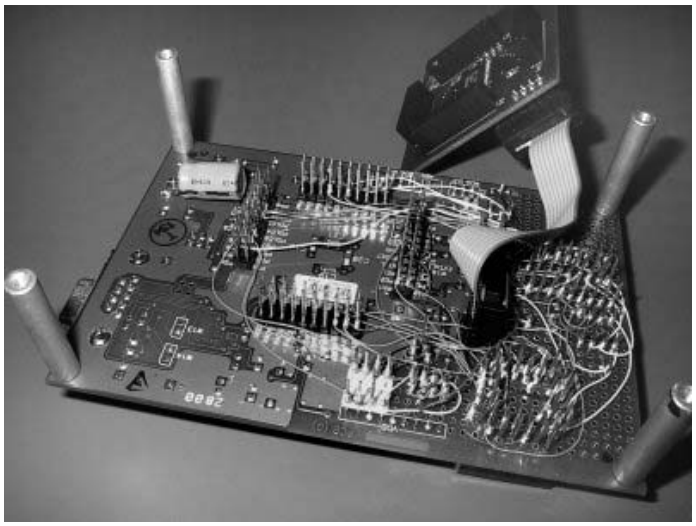


Figure 5: Bottom of the HC12

On the HC12 side, the GPS plug was a simple 4-pin header block. The GPS has a RJ45 connector with eight lines, however we need only four: Vcc, GND, Data In, and Data Out. We placed a RJ45 socket on the side of the module, and connected the GPS to the HC12 through this

connector, with only the four lines we need in the plug. This then plugged onto the header block, which was connected via wire-wrapping.

The UART required that we purchase a crystal to provide a clock. The data sheet listed many that would work, and we purchased a 1.8432 MHz ECS crystal from Digikey, and then divided down to get frequency of 19.2 kHz. This can be found in the UART Schematics, Appendix A.

Precision Navigation Vector 2X Digital Compass Module

From the module requirements, we knew that we would need some method of determining the robot's heading so that we could calculate turning angles. A GPS could be used to determine this (indeed the feature is built into most GPS modules), however the short distances and slow speeds (about 4mph, as specified by chassis module) that the robot travels at made this option undesirable at best, and highly inaccurate at worst. Therefore, we chose to implement a design using a digital compass.

The Precision Navigation, Inc. Vector 2X digital compass was chosen for the price and small accuracy errors when flat. It also has a published known error when tilted, so we knew that tilt-induced errors could be corrected for. Because of the contoured nature of the chipping green, we determined that tilt correction was vital for proper operation and accuracy; this will be discussed in the accelerometer hardware and software sections.

Ultimately, the deciding factor to use the PNI Vector 2X instead of a more expensive design such as a Honeywell HMR3000 is that the Vector interfaces with the microcontroller over the standard Motorola SPI interfacing protocols. This means that the physical connections were limited to three wires (Slave select, Data In, and Clock) for the compass communications, plus

lines for setting the various features of the compass, which only adds another six lines including power and ground, for a total of nine connections to the compass. These are shown in the following table.

Compass Line	HC12 Connection
Vcc	Vcc
Gnd	Gnd
!XFLIP, !M/S, !BCD/BIN, !RAW	Vcc
YFLIP, !RES	Gnd
!SS	PortS7
EOC	PortDLC1
!CAL	PortDLC2
!P/C	PortDLC3
SDO	PortS4
SCLK	PortS6
!RESET	PortDLC0

Table 1: Compass Connections

The compass sense the earth's magnetic field and computes the direction based on that. As such, there are exposed magnetometer coils on the compass board. The magnetometer coils on the compass are published to be sensitive to having particles such as dust and dirt in them. This meant that the compass could not be mounted outside the box, unless encased. Product literature provided with the compass states that the compass also does not have a "conformal coating" to protect it from moisture. This means that the compass should not be allowed to get wet. These facts meant that we wanted to mount the compass inside the case.

The aluminum casing provided by the chassis module was acceptable to meet both of the mounting requirements, so the compass was mounted inside the top of the case. Compass coils are known to be sensitive to ferrous metals such steel and iron; however, non-ferrous metals such as aluminum are ideal for mounting a compass on or in. Compass mounting to the case is shown in figure 6, below.

For power requirements, the compass runs on standard TTL level logic (5V +/- .25V and GND). The specifications on the compass allow that a voltage greater than 3.15V is logic "1"

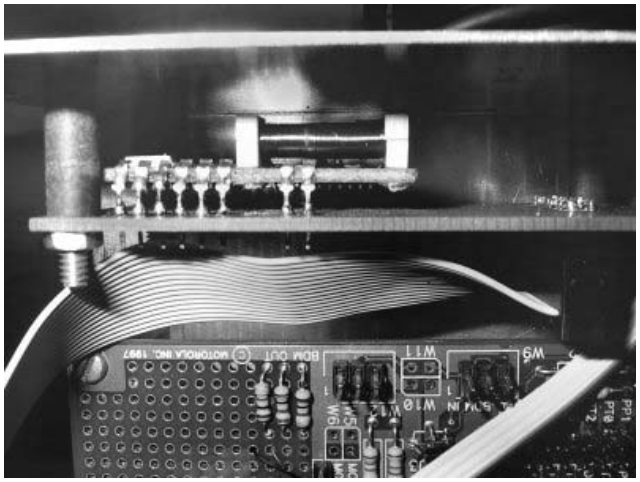


Figure 6: Mounted Compass

and less than 1.35V is logic "0." This is compatible with the HC12. Since we are running in slave mode at 5V levels, we only draw about 4mA during the poling cycles, and about 100µA during idle/sleep periods, which is low enough that the power can be drawn from the HC12 (which can supply up to ~20mA on a pin).

Because the compass came as a module design, shown in figure 7, there were exposed

Figure 7: PNI Vector 2X

leads on the bottom. A board was designed using ProTel, and the tech-room etching equipment was used to etch the board, shown below in figure 8, with the mounted compass. The compass was then soldered to the board and mounted in the casing. Three-quarter inch insulating spacers

were used to separate the compass from the casing, and eliminate possible contact of the coils with the casing.

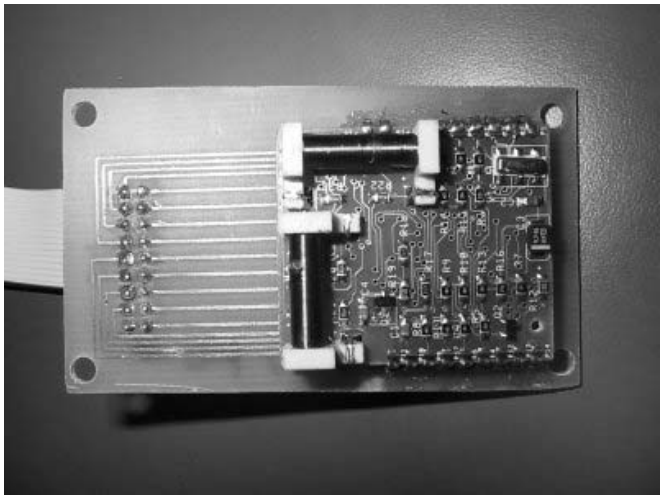


Figure 8: Compass on Etched Board

The compass has a total of 17 pins on the board (the photo shows 18, the lowest left side pin has no connections), and some of these can be tied either together or to power or ground, depending on the modes that the compass will be used in, and the SPI pin is not used in our design. We chose to design the board with no pins physically tied together on the compass module, or to tie pins on the etched board together, as this would not allow for possible future expansion, such as raw magnetometer output instead of heading output. The pins that could be physically (see Table 1) connected for our setup were tied together during the case construction and connector mounting via the wire-wrapped connections to the HC12. This method would be easy to undo if changes or expansions were later made. The wire wrapping was done on the bottom of the HC12 board (Figure 5), to the plug that was installed in the expansion area of the HC12. Please refer to Appendix of Schematics, for a diagram of the compass connections.

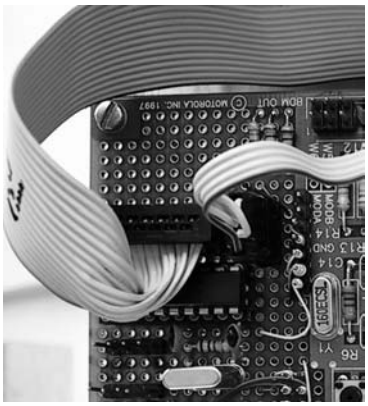


Figure 9: Compass Connector

For the connection to the HC12, we used a single 16-wire ribbon cable with IDE 16-pin connectors on either end. Both the compass board and the HC12 had blocks of 2x8 pin header pins mounted on, and the ribbon cables simply slid onto those. We especially liked this design since the design of the IDE plugs was such that they could not physically be connected to the side of where they should be, as the exposed pins would block the wide plug.

Because the chassis is using motors to drive the robot, we had some initial concerns about the effects that the magnetic fields would cause. The compass is not capable of correcting or calibrating for these types of field. We alleviated this concern by implementing a design that only took headings when the robot was stationary, when the fields were roughly constant. This eliminates most of the errors.

Analog Devices ADXL202JQC Accelerometer

The PNI Vector 2X digital compass utilized for heading determination was known to have errors for a given tilt, or angle of attack (AOA). The latitude of compass usage determines the error for a known AOA. In central New Mexico, where the robot has been developed, the digital compass information loses approximately three degrees of accuracy for every one degree

of tilt in both the x and y axes. To correct for this error, we have employed a two-axis accelerometer developed by Analog Devices.

The Analog Devices ADXL202JQC Accelerometer, shown in figure 10 mounted on the circuit board, was chosen for the task of tilt-correction primarily for ease of use and cost. The ADXL202, when powered by a required +5V DC signal, presents a pulse-width-modulated signal from both the x- and y-axis outputs. Optimally, at zero degrees of tilt, the PWM output signal has a 50% duty cycle. With our chosen calibration resistors and capacitors, this duty cycle

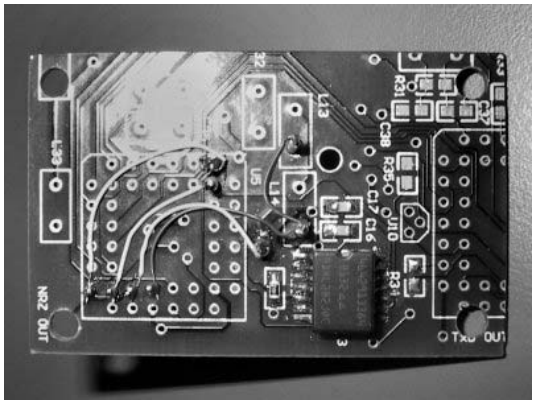


Figure 10: Accelerometer on Board

varies with tilt up to a maximum of approximately ± 12.5%. For ease of calculation, we initially made the assumption that the completed robot would never see a tilt of more than ±15 degrees in either axis. This decision was based upon the essentially linear operation of the accelerometer up to about 15 degrees. Beyond 15 degrees, the rate of change of the accelerometer changes enough to be problematic in calculation. The characteristic for the accelerometer change in output can be seen in Figure 11. The manner in which the program determined the acceleration proved, however, that we could in fact use the compass through any range of tilt we desired. A maximum tilt of ±30 degrees was chosen for convenience, however. The circuit we generated for the accelerometer can be seen in Appendix A. The component values shown in the circuit

were chosen to give a period of the accelerometer output of approximately 2 msec. In practice, however, neither the accelerometer, nor the components, was perfect and thus we managed a period of approximately 1.68 msec.

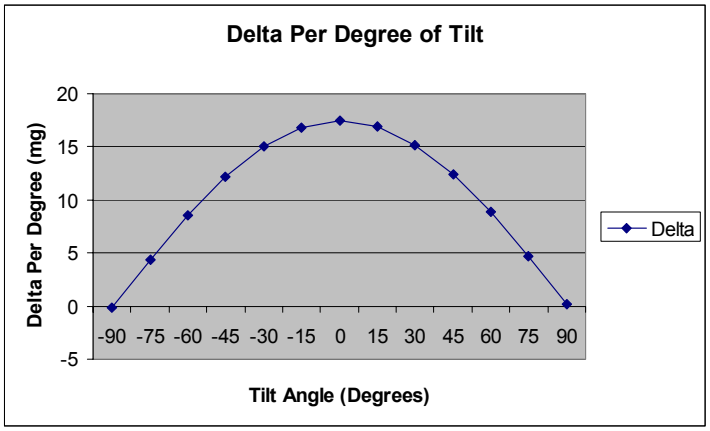


Figure 11: Accelerometer Tilt Curve

The accelerometer was mounted in the case in such a manner to facilitate ease of use and calculation in conjunction with the digital compass. The accelerometer was mounted on the high side of the module, in-line with the compass. This placed the accelerometer axes in-line with those of the compass. The mounting was accomplished by drilling holes through the top of the module's aluminum casing and placing screws with spacers on them through the board. The

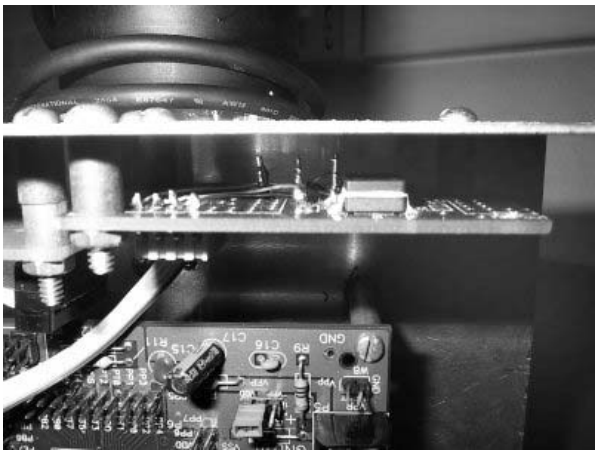


Figure 12: Accelerometer Mounted in Case

accelerometer was then attached to the HC12 itself using a small, four-pin ribbon cable and the requisite 4-pin header attached to the HC12. On the accelerometer board, we used a 4-pin Molex

connector attached to the underside of the board, with connections soldered to the wire-wrapped leads on the top of the accelerometer board (can be seen in figure 10, lower left). Connections to the HC12 can be seen in the included schematics in Appendix A: Schematics

Communications

Even though we chose to implement a design based on the HC12's simple SPI interfacing protocol, the actual physical interface was a bit complicated. Due to the availability of female RJ45 sockets locally, we chose to implement a design for interfaces based on the RJ45's eight-wire conductor Pattern B. The male plugs were provided by the chassis module. The eight lines in the RJ45 are Blue, Blue and White, Green, Green and White, Orange, Orange and White, Brown, and Brown and White. Physical line connections are listed below, as seen by the Navigation Module as the master device. Significance of the lines will be discussed in the Communications Software, Appendix B. For ease of interfacing this protocol onto our HC12, we built an expansion, shown in figure 7, board that would fit onto the top of the HC12. Header pin extensions were used plug the board onto the top of the HC12, as shown in figure X3. The connections from the RJ45 sockets on the side of the module were soldered to the corresponding pins on the HC12. Schematics can be found in Appendix A. To hide the solder joints and

Line Color	Function in Communications
Green	Input Capture, PortT2, 4, and 6
Green / White	Master In Slave Out,
Blue	Slave Select, PortP0, 2, and 4
Blue / White	Master Trigger, PortP1 and 4
Orange	Master Out Slave In, PortS5
Orange / White	SPI Clock, PortS6
Brown	Dedicated stop line
Brown / White	Not used, reserved for future expansion

Table 2: RJ45 Color Setup

prevent the pins and lines getting bent and touching together, a black box was installed over the "expansion board." This is shown in figure 13.

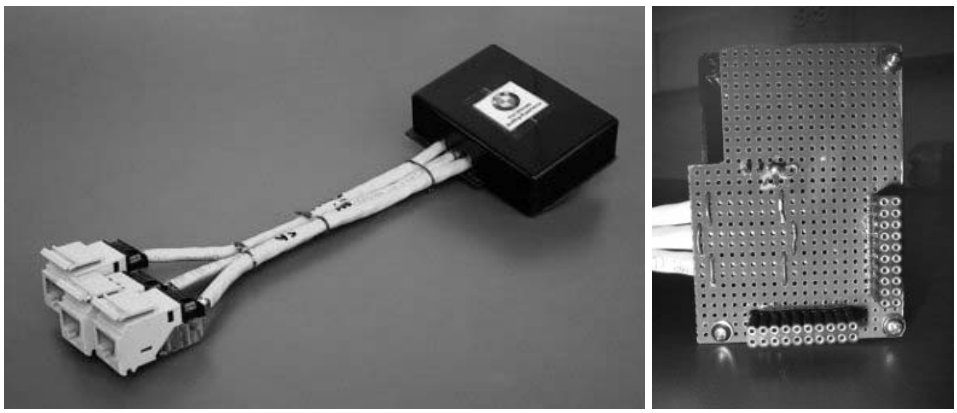


Figure 13: Communications Expansion Board

For ease in interfacing, we had a robot-wide meeting in which the details of the interfacing were discussed, including a description of the hardware setup required. A list of connections to the HC12's was distributed to each of the three other subsystems. Additionally, we tested and had meeting with each module separately to test the hardware interfacing.

For the patch cables we used straight lines of Ethernet cable (pattern B), set up as a straight through with no line crosses. We made three cables, all interchangeable and distributed to the groups. They were made using standard cable, connectors and crimpers.

Software Design

As in the hardware section the software discussion will be broken up by function: GPS, String Parsing, Math Functions, Compass Setup and Data, Accelerometer Setup and Data, and Communications.

GPS

The GPS emits several ASCII strings when power is applied, regardless of whether the positions are based on satellite data or not. This data comes at a configurable rate of once a

second. We configured our receiver to send only one string out of about 10 or so built in. This string gave:

- UTC (GMT)
- Degrees and minutes up to 1/10000 min in latitude and longitude (decmin)
- A number signifying whether the data being transmitted is accurate, as well as whether WAAS corrections are available
- The number of satellites being tracked
- Elevation in meters above sea level
- Horizontal dilution of position (HDOP): a measure of how probable a given location is to being correct
- Height of Geoid above WGS84 Ellipsoid
- A checksum, consisting of all characters up to the checksum field save the leading '$'

NMEA strings are comma delimited, so it is simple to find the data field needed and pluck the integers out. Simply count commas to find correct data field, so we then know where the head of the number we want is. Depending on how large the number is (all numbers have a consistent size in the string), we can subtract the value of ASCII 0, then multiply by the decimal power for which this character sits in the number, and add all of them together.

We decided to break the actual positioning data into integers: one each for the x and y degrees, the x and y whole number minutes, and the x and y fractional minutes. Since in the given specifications, the distances were relatively small, at least less than a whole minute, we concluded that it was feasible to only use the fractional minutes of position to calculate distance and heading information. Therefore we used units of decimal minutes (decmin). A simple check for minute rollover makes this a safe assumption.

Code for parsing the GPS string can be found in Appendix B: Code, the GPS section

Math Functions

Going to many Internet sites, you can find a Great Circle calculator that will tell you the distance and angle from north between two GPS coordinates. These calculations use a series of sine and cosine calculations and are extremely accurate over long distances by taking the curvature of the earth into consideration. We found out that all of our movements would be on a chipping green, with our greatest distance of travel not to exceed 150 feet. Across a distance this minute, the curvature of the earth is negligible. This means that our calculations can be considerably simplified. Instead of using the Great Circle calculations, we can assume that the chipping green is an X-Y coordinate plane. A difference in longitude shows up as a difference on the X-axis, while a difference in latitude shows up as a difference on the Y-axis. By using Pythagorean's theorem and some simple Trigonometry, we could easily find the distance to travel and the angle from north to traverse between one coordinate and the other.

The HC12 has some libraries and header files which can be included that would help in these calculations. There is, for instance, a square root function and a tangent function included that would be very useful in our calculations. Both of these functions use double floats to make their calculations. With these included functions, we could do all of our calculations with only a few lines of code. The HC12, however, doesn't include a print statement in DBug12 that will allow the user to view floats. The only way to view floats is to write your own printf function. Floating-point calculations also take up much more memory than integer calculations. For these two reasons, we opted to perform all of our calculations using integer math. Although these calculations were not quite as accurate as the floating-point arithmetic, the error was within

reason. We could still get as close to the target as the Ball-Hole locator group demanded. The code for the integer arithmetic was much longer, but it compiled smaller than the floating-point arithmetic would. It was also much easier to fix any problems that arose.

By using Pythagorean's theorem and trigonometry, we could easily find the distance and angle from the robot to the target. Pythagorean's theorem calls for a square root of the two sides of a triangle in order to determine the length of the third side. To find the angle, we had to use a tangent function. This means that we had to generate a square root function and a tangent function using only integers. For the square root, a simple FOR loop was written that would start with a guess of value zero, square it, compare it to the value to be rooted, and increment the guess value until it found the root. This always rounded down, but this problem was ignored. The tangent function used a look-up table by comparing the ratio of the two sides of the triangle to the values in the table. The X-Y coordinate plane was divided up into the four quadrants. A few IF statements determined which quadrant the coordinates were in. The look-up table only has 45 values in it. This is because the tangent values from 0 to 44 are inversely proportional to the tangent values from 45 to 90. By finding the values of tangent 0 through tangent 44, and applying a few other lines of code, we could find the actual angle from true north to travel to arrive at the target.

This process resulted in distance units of decmin (decimal minutes) and angular units of degrees. When distance and angle commands were passed the chassis module, they had these units. One decmin is approximately equal to .8 feet, or 9.6 inches.

The code for the math functions can be found in Appendix B: Code.

Compass Setup and Data

For interfacing with the HC12, the PNI Vector 2X uses the standard Motorola SPI interface, which was one of the primary reasons it was selected. There are two options in setting up the compass output: Binary Coded Decimal (BCD) or Binary. A discussion of differences can be found in Appendix D: References, compass handbook. We chose to implement the binary method since the math was simpler than the BCD, and the output would be equal to the heading, without masking and performing math on certain bits. Because the compass outputs a full 0° to 360° swing, and the char variable is limited to 255,we had to have two transfer cycles of the SPI, with only the last bit of the first transfer (bit 0, or the LSB) being important.

The SPI on the compass is built so that the compass transmits each bit of data on the falling edge of a clock that has a high polarity. These settings were made on the HC12 according to the register setup described in the HC12 manuals (the "Beige Book"). The compass is specified to be able to transfer data at up to 1 MHz, for testing, we ran the SPI at 500 kHz, set up in the SP0BR register. We had some issues during testing, so we ran the clock even slower. During interface testing, communications module that we run the clock as slow as possible so that they could meet their timing requirements and this meant that the SPI ended up being run at only 31.3 kHz. However, this did not affect the compass.

Since we have a nine-bit number, an unsigned integer-value variable was used in the program. The SPI transfers only eight bits at a time, so the following steps were used to retrieve the heading from the compass.

1. Transfer 1: the upper 8 bits of compass output (only bit 0 important)
2. Save to temporary variable, heading1
3. Transfer 2: lower 8-bits of heading (all important)

4. Save to temporary variable, heading2
5. Place heading2 into the integer heading variable, robot_heading
6. Check bit 0 of temp1. If value is "1", add 256 to robot_heading. If "0" do not modify robot heading
7. robot_heading now contains the compass heading and can be used in calculations,

An interesting bug developed while doing this: this method worked well as long as the dummy variable that the HC12 was sending to the compass (which was not actually connected to the compass) was 0x00. Any other values would cause the SPI to receive only the "dummy" values we placed in the SPI Data Register (SP0DR).

For setting up the compass, we wanted is to be in slave mode, binary output, not pole data continuously, not raw data output, low resolution (acceptable for our area of the country and recommended in the manual), do not flip the directions (compass is mounted right-side-up with no rotation), and send MSB of heading first. The following lines needed to be set:

Line on Compass	Value needed in settings
!P/C	1
!RAW	1
!M/S	1
!RES	0
!XFLIP	1
YFLIP	0

Table 3: Compass Line Settings

Since these lines would not change in our setup (unless the robot was upside down and still needed a heading, which would be a bit of a problem…) they were physically tied to Vcc or ground to eliminate the extra connections and lines of code needed to set them.

Because we need to calibrate the compass when we power on, a calibration routine was written, following the guidelines from the compass manual. This stipulated that the following procedure:

1. Power on the compass
2. Bring !CAL low for at least 10msec, then bring back high
3. Rotate 180°
4. Bring !CAL low for at least 10msec, then bring back high

This allowed us to correct for the static magnetic fields and hard-iron that were on the robot. For the rotate 180°, ideally we would send a command to chassis that would rotate and then inform us when finished. Since we were unable to test with the chassis module, for testing purposes we implemented a protocol that would inform the user to rotate the compass, wait three seconds, and then finish the calibration. This turned out to work very well.

There was a bug that appeared in the compass, and is in fact mentioned in the handbook: it is a requirement that certain lines (such as !P/C and !RES) must be set either high or low before the power is supplied to the compass. Since we were running off the HC12, the power would be supplied as soon as the HC12 turned on, thus the setup requirements would not be met because the power would be on before lines were set. However, there is a work around.

We were able to power the compass and setup the lines as needed initially, we then ran a reset on the compass, which overwrote the setup, and met the requirements. For some reason that is still unknown, we had to run a reset before each reading for headings. This is not an issue to the accuracy, since the calibration settings are not erased during this procedure, and was only a minor annoyance and short delay (16 msec) in the program.

Compass code can be found in Appendix B. The handbook from PNI is available for download. The link can be found in Appendix D: References.

Accelerometer Setup and Data

Using the Motorola 68HC12 microcontroller with which we were furnished previously, it was a simple task to interpret the output of the accelerometer. We set up two bits of the HC12's Port T, Pins PT1 and PT3, as input capture ports. The only other connections from the accelerometer to the HC12 were for power and ground.

In the initial accelerometer setup, both of these capture pins are made to capture rising edges. This allowed us to determine the time of the first PWM signal edge. Once the first edge is captured and the appropriate flags are cleared, the input capture pins are reconfigured to capture falling edges. This then captures the falling edge of the PWM signal. Simply subtracting the rising edge time from the falling edge time gives an accurate measure of how long the PWM output is high. The program then reconfigures the input pins to again capture rising edges, allowing the time of the third edge of the PWM signal to be captured. This provides for calculating the total period of the PWM. This is done by subtracting the first rising edge time from the second rising edge time. This data, when used with the logic high time, allows for the calculation of the PWM duty cycle through a division calculation (high time divided by total time to give percent duty cycle. Once the duty cycle is determined, a formula provided in the ADXL202's datasheet was used to calculate the acceleration in both axes. Once the acceleration had been determined, an if-else comparison is used to find the corresponding angle. This information can then be used to correct the heading information derived from the digital compass. The method of correction was determined from our knowledge of the compass' error. According to specifications, the compass would lose three degrees of heading accuracy for every single degree of tilt. In testing, this proved to be overkill, and we were able to use a correction of

two degrees for every single degree of tilt. The correction was simply added or subtracted, based on the direction of tilt, to the compass heading.

There were two major hurdles in the development of this program. The first was the initial use of interrupts to determine the timing. It seemed as though ghosts in the machine prevented the HC12 from working properly with the accelerometer while interrupts were enabled. The second hurdle was in getting an averaging routine to work with the accelerometer. Due to sensitivity of the accelerometer, it was necessary to implement an averaging routine such that the compass would hold data when it wasn't moving. It took several revisions of code to get averaging working correctly, but in the final program it worked correctly with several layers of averaging taking place before the final calculations. The completed program can be seen in Appendix B: Code.

Communications Protocols

For communications, we chose to use the simple Motorola SPI interface already on the HC12. It had the advantage of being able to share a clock and data lines between all the slave devices (compass and the three other modules on the robot) and use a separate slave select line for each of the other modules. We also chose a design that would use additional handshaking to help ensure that all modules were in the proper states to receive or send data. A waveform outlining the design is shown below. This is the design that was ultimately used by the other subsystems.

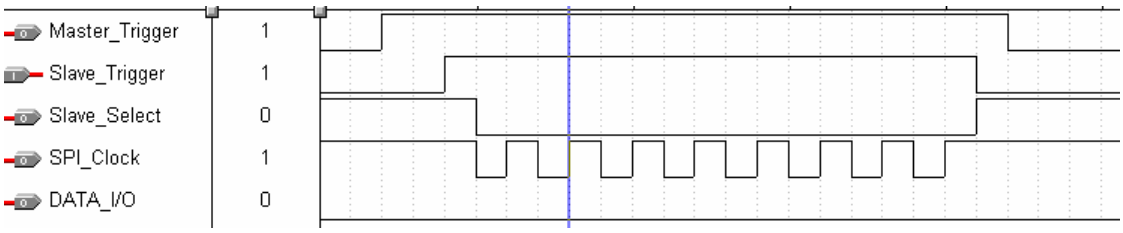


Figure 14: Waveforms of Communications Protocols

The protocol worked as follows:

1. Master would raise a trigger line telling slave that we wanted to transmit. Master would go into a "wait state" for a return signal
2. This would cause an interrupt on the slaves' input capture timer lines and set a flag, receiving = TRUE. The receive function would be called.
3. The receive function on the slave would send a trigger line high, causing an interrupt on the master HC12, and the transfer would begin.
4. Both master and slave would wait for the SP0SR register flag to be set, read the data out (to clear the flag). All flags (such as *receive* on the slave) would be reset to FALSE and the function would terminate. This completes the transfer.

Sending data from slave to master followed a similar cycle, except that the master does not start the first handshake. In this case, the only difference is that the master's trigger line is not used at all. The rising edge of the slave trigger would cause an input capture timer interrupt, and because the master knows that it is now sending data, a receive from slave function would be called.

The communications module informed us that they would not be able to run at a high SPI clock speed, so we ran at only 31.3 kHz (set up in the SP0BR register) and this made all the groups happy, since Ball/Hole and chassis did not have any preferences.

Since we needed to have three different slave select lines on the HC12, one for each module, we chose to use a generic I/O port that was not being used to be our trigger and slave select lines. Since our module does not use Pulse Width Modulation, we used PortP for the communications lines. PortP was setup as follows:

Port Number	Description
PortP0	Slave select for chassis module
PortP1	Master trigger for chassis module
PortP2	Slave select for ball/hole location module
PortP3	Master Trigger for ball/hole location module
PortP4	Slave select for communications module
PortP5	Master trigger for communications module

Table 4: PortP Communications Setup

For the trigger line flags to get set, we needed method to capture the change in level in the lines, so we chose to use the timer ports on the HC12 setup as input capture. The type of edge was not important, and the decision to use idle low and capture the rising edges was arbitrary at best. We had a few discussions among the modules and nobody had and issues with this setup.

On initial startup, we send a command to communications and the waited for a sequence of eight transfers in return. These consisted of the target coordinates of the ball and the hole. These were also used as the START command.

The communications setup is also where we chose to implement the STOP/PAUSE command from communications module. It works as follows:

1. Rising edge is received on TIC0 (Timer 0, input capture)
2. Stop and enter the interrupt
3. While PortT0 is still high, wait
    a. A stop command will result in a line that stays high indefinitely, so we never leave the function. STOP never leaves this place.
    b. A pause command will stay high for a while, then drop
4. Resume code

This was never actually tested, and we do not know if the communications module had implemented the function.

Main Program Flow

Initialization

The initial hardware setup starts with all the expansion serial port setup (setting flow control, modem settings, internal interrupt signaling, FIFO initialization, and data rate). Then the compass is setup by calling the compass setup functions. The SPI registers are then setup for communication with the compass as well as with the other subsystems. After communication is setup, the compass can then be calibrated to compensate for nearby magnetic fields.

Initial coordinates/Start

A sequence of eight bytes is then requested from the communications subsystem. The first two bytes become an integer representing the x coordinate of the ball, the next two the y coordinate. Similarly, the next four are the x and y coordinates of the hole. The completion of the eighth byte signals the start of the ball finding loop.

Finding Ball

After the current position and orientation of the robot is found. The distance calculation function is then called, which returns a flag as to whether the position is close enough for the ball/hole locator to find the ball. If the distance is under about 8 feet, then the loop is broken and the locator subsystem is signaled to find the ball. If the distance is greater, an angle to rotate to is calculated and one of two things may happen. If this distance is less than about 30 feet, then the chassis is signaled to rotate the calculated angle and move that distance. If the distance is greater, the robot rotates the prescribed angle, but only travels half the distance. This is due to

inaccuracies in the distance calculation function. Below this threshold, better measurements can be made. This process is repeated until the approximate eight foot threshold is met.

Ball capture

After the ball locator is signaled to start, it sends commands to move through the navigation subsystem to the chassis, which makes movement corrections. The navigation subsystem is not involved in this process except for the relaying to the chassis.

Hole location/Drop-off

The above location process for ball capture is then repeated for the hole location and ball drop-off.

Team Member Participation

For this section, each member wrote his own job descriptions, so the use of "I" and "my" is intentional and meaningful. This also accounts for the different writing styles that prevail in this section.

Jonathan

This project was divided up into several pertinent tasks at the beginning of the semester. My tasks are outlined below:

- Work with Dave in understanding the GPS receiver
- Write Altera code to allow for expansion serial ports using a UART
- Wire wrap UART chip and Max232 level-shifter

- Write custom integer math functions for determining direction and heading of robot travel
- Mounted all components into our module and made all internal connectors

The beginning of the semester saw our team dividing into the compass/accelerometer group and the GPS group. Dave and I decided to start working on the GPS receiver. We received some evaluation software from Garmin that allowed us to talk to the GPS receiver through a standard serial connector on a computer. We discovered that the GPS receiver used RS232 voltage levels, and the computer used TTL voltage levels, so a level-shifter was required.

Once we knew what was coming out of the GPS receiver, I started writing the Altera code to decode the address lines coming into the UART chip. Once this was completed, I began wiring up the UART chip and the level shifter. These were attached on the expansion section on the HC12 board. I eliminated expansion PORTA on the memory expansion module and used this to run the lines for selection of the UART chip. Testing of our board showed that we could communicate with another board using the serial port.

As soon as this was completed, Dave started writing the code to parse the string of information coming in from the GPS receiver. Feeling useless for having completed my pre-assigned tasks, I immediately began working on the math functions for determining the robot distance and heading to target given the GPS coordinates. These were done with custom integer math functions that were relatively simple in their design and implementation. The code complied small, but wasn't quite as accurate as the floating-point arithmetic would be. Testing of this with Dave's code showed proper operation.

When this was all completed, I began hardware mounting and internal connections. Our last step was final integration with everyone else and final code generation. Although the final

program flow was completed, final integration with all groups was not completed as of the time this report was due.

Matthew

My primary tasks in working toward the completion of our module included the design and etching of circuit boards and the implementation of our accelerometer for tilt correction of the onboard digital compass.

The circuit boards I constructed were made for interfacing the digital compass and the accelerometer to our HC12. Both boards were design using a version of ProTel that was included in the network software of the EE labs. The compass interface board included traces to connect all of the pins on the compass to a header that made it possible to interface with the different subsystems of the HC12 using a ribbon cable. The accelerometer board was much the same, but contained pads for a number of components to be added in addition to the surface mount accelerometer itself. In the end, due to numerous problems with both the board-etching process and the accelerometers we had originally chosen to use, the accelerometer board was discarded in favor of a board provided to us by Dr. Rison.

My other major area of expertise on our part of the robot was in writing the programs to interface the accelerometer with the HC12. This in itself was not complicated, but several problems crept up along the way. The first problem was the use of interrupts in the initial program to capture edge timings for the accelerometer. This problem had a number of possible causes, including running out of room in the stack on the HC12. As it could not be determined what was actually causing the problem with interrupts, I decided after talking with Dr. Rison to move away from interrupts. The next problem came with using averaging in the program. The

accelerometer reading tends to jitter about even when it is sitting still, and so I decided that averaging needed to be implemented. However, it took several versions of code to make this work and in the end averaging only worked in the final program and not in any subsequent version of the code. The final problem we had was with an incompatibility with the compass that caused the accelerometer program to miss capturing edges in its routines. This was solved by disabling interrupts in the accelerometer function.

My final contributions to this project came in the same manner as those of my teammates – namely, in troubleshooting, proofreading, and similar mundane tasks. I also did a great deal of the photography that was seen both in our presentation and this final paper.

David

My primary objective was to get information from the GPS into some useful form for heading and distance calculations. We had all decided early on that since all the GPS units we found spoke RS232, so the plans for extra serial ports were in the works from the early weeks. Jon and I worked on both the alterations to the Altera memory expansion to memory map the port, as well as the decrypting of the data sheet to figure out what connections we needed. I also figured out how to initialize and test the serial port once it was finally programmed and wire-wrapped.

I played with the GPS and determined the magic string that gave us all the information we needed. Then Jon and I started on getting the serial port to actually read characters in. It would write quite well, but the reading would all smear together. Later on, I discovered the data ready bit on the port, and read quality got much better. After that, reading the GPS went quite well. I also wrote all the final code to read, parse, and extract the needed data from the reading.

I also constructed all the cabling for communication between subsystems and between all the sensors and the microcontroller board, as well as final program flow and layout, and slapping all this engineer code into shape. And hundreds, nay, thousands of hours of testing.

Ryan

My duties this semester included:

- Initial research and proposing ideas about the type of GPS and compass we should use
- Ordering the components we needed
- Writing the compass code and testing the compass for correctness and accuracy
- Writing the communications protocols and code and testing the communications interfacing

Early on in the semester, I took it upon myself to begin conducting research into the various GPS's and compass that were available in the marketplace. This included many hours of on-line research and phone calls to various manufacturers and suppliers. I narrowed the initial choice down to three or four GPS's and compasses and then the group made a joint decision about what should be ordered. As I had been chosen as the purchasing representative during the first week of class, I ordered the equipment we decided to use, including the GPS and compass. Also order samples of UARTs, accelerometers and sockets as needed.

As soon as all the initial proposals and presentations were finished, I worked with Matt to etch the boards that we would need for the compass and accelerometer mounting in the case. Because I was the one that was most familiar with the layout, I spent a week working with Matt and tweaking the board designs. I also assisted in etching the.

Repeated equipment problems turned what should have been a one, perhaps two, day project into a three week project. Even after putting so much work into the boards, we ultimately abandoned the accelerometer board for a surplus board donated by Dr. Rison.

As the semester progressed towards (and beyond) midterm, I became primarily responsible for the compass, and the communications interfacing and software. For the compass, I wrote all the setup and function calls that are involved in using the Vector2X, and gave these to Dave to be integrated into the main Navigation Module code.

Additionally, I worked with Jon to develop a system of physically integrating the compass to the HC12. This included the primary decision of what type of wire (the 16-line ribbon cables) and connectors (16-pin IDE to a 16 pin header block) should be used.

At the end of the semester, my final contributions came in the form of endless hours in the Digital Lab testing and re-testing. This included many hours with Azmat from Ball/Hole location module testing my communications protocols. I also wrote most of the final presentation and worked on writing, proofing and correcting the final report.

Final Budget

Our initial budget for this project was $300.00 dollars. This is shown in Table 5.

Item Name	Price ($)
Garmin GPS16	122.00
PNI Vector2X	50.00
ADXL202AE Accelerometer	0.00
Module Casing	40.00
Miscellaneous IC's	20.00
Miscellaneous wiring, etc.	50.00
TOTAL	**~$300.00**

Table 5: Initial Budget

Our final budget is included below. It contains a list of all expenditures made through both Norton's shop and various online resellers. We are proud to note that our total expenditure of $242.71 is well under our projected budget of $300. Several items made this possible. First, we were able to purchase our GPS, with an academic discount, direct from Garmin. In addition to receiving our GPS at a discount, we received Windows-based software for the GPS for free.

Item	Quantity	Unit Cost	Total Cost
Garmin GPS16 GPS Receiver	1	$137.00	$137.00
PNI Vector2X Digital Compass	1	$52.68	$52.68
C-Thru Protractor	1	$2.13	$2.11
14 Pin Ribbon Cable Connector	1	$2.00	$2.00
14 Pin DIP Socket	2	$1.50	$3.00
Miscellaneous Pins	5	$0.60	$3.00
Defluxer, 12 fl. Oz.	1	$12.20	$12.20
6" x 6" PC Board	1	$5.00	$5.00
Masking Tape	1	$3.50	$3.50
100 Ohm Resistor	1	$0.10	$0.10
120 Kilo-Ohm Resistor	1	$0.10	$0.10
0.1 Micro-Farad Capacitor	3	$0.10	$0.30
DB9 Ribbon Connector, Male & Female	2	$2.00	$4.00
2 Pole Pushbutton Switch	1	$1.50	$1.50
16 Pin Ribbon Cable Connector	2	$2.00	$4.00
Small Connectors	3	$0.60	$1.80
PerfBoard	9 sq in	$0.50 / sq in	$4.50
6-32 x 3/4" Screws	4	$0.10	$0.40
6-32 x 3/8" Non-conductive Spacers	4	$0.10	$0.40
6-32 Nuts	4	$0.03	$0.12
4" x 4" x 1/8" Plexiglas	2	$1.00	$2.00
PVC Pipe	3 ft	$1.00 / ft	$3.00
Motorola 68HC12 Microcontroller	1	$0.00	$0.00
Analog Devices ADXL202 Accelerometer	1	$0.00	$0.00
National Semiconductor UART PC16552D	1	$0.00	$0.00
3" x 5" x 7" Aluminum Case	1	$0.00	$0.00
RJ45 Connectors	1	$0.00	$0.00
Power Cords	1	$0.00	$0.00
Epoxy	1	$0.00	$0.00
44-Pin PLCC Socket	1	$0.00	$0.00
		Total Expenditures	**$242.71**

Table 6: Final Budget

Also, we received our accelerometer free of charge as they have now been discontinued. Finally, we received a number of our parts as donations, not the least of which was the Motorola 68HC12 microcontroller that was used as the brains of our module. Further donations came from Dr. Bruder in the form of the 44-pin PLCC socket that was used to hold our UART, and the casing for our module that was donated by the Robot A Chassis Group.

Motorola 68HC12 Microcontroller	1	$80.00	$80.00
Memory Expansion for HC12	1	$70.00	$70.00
Analog Devices ADXL202 Accelerometer	1	$13.60	$13.60
National Semiconductor UART PC16552D	1	$5.00	$5.00
3" x 5" x 7" Aluminum Case	1	$12.00	$12.00
RJ45 Connectors	1	$5.00	$5.00
Power Cords	1	$7.00	$7.00
Epoxy	1	$3.99	$3.99
44-Pin PLCC Socket	1	$.89	$0.89
Total Additional Expenditures			**$197.48**

Table 7: Reproduction Budget

Power Budget

Since we were never able to run off the chassis power while running the programs and interfacing with other module, the below power budget is approximate. Measurements, while running tests, produced readings that were similar.

ADXL202JQC Accelerometer	2mA @ 5V
Garmin GPS16	80mA @ 5V
PNI Vector 2X	6mA @ 5V
68HC12 with expansion	150mA @ 5V
TOTAL	**238mA @ 5V**

Table 8: Power Budget

Conclusion

This project was a mixed bag of good and bad. While it was nice to have a different project that previous classes have had, ultimately this project, as presented, was a bit too long for a single semester. We feel that we were at about 95% done. The missing 5% is accounted for in the integration part of the entire robot. We were able to test integration with both Communications and Ball/Hole modules. However, we were never able to test with all three other modules, so that aspect of the design remains untested, hence the 5%. We did test with Chassis individually, so we believe that the integration could have been completed in a small amount of time once the Chassis module got to this stage and was ready to integrate.

To do this project over again, the biggest change we would make would be to spend a bit more of the total $2000 budget that each robot had. Specifically, since the Vector2X was not always reliable, we would perhaps like to have purchased something like the Honeywell HMR3000 series. However, that unit costs over $350, so perhaps a happy medium between the Vector and HMR could be found. The GPS was a good product, but perhaps we could have combined the compass and GPS budget together and purchased a GPS with built in compass. This may have ended up being a handheld, but they still have a serial interface.

The single biggest challenge in this project was integration. This is itself should not have been bad, but the Navigation Module is responsible for most of this, and we were not able to test with other modules until they reached completion. This did not happen until about the last week or two of the semester. We had hoped that by distributing code a pin-outs for the communications that this would help come integration time, but most of the problems seemed to be more hardware (i.e. the HC12's the other modules were using) rather than software based. We

believe that if all the modules would have gotten to the integration stage even a week earlier, the class would have had working robots.

Appendix A: Schematics

This appendix contains schematics, in block diagram form, of the physical connections that were made to the 68HC12 board from the added hardware, such as UARTs and Compasses. Passive elements, such as capacitors, have been added to the schematics where appropriate. Most of the physical connections were made with the wire-wrapping techniques, as explained in the Hardware Section, except for the Communications block, which was soldered to expansion header pins.

UART

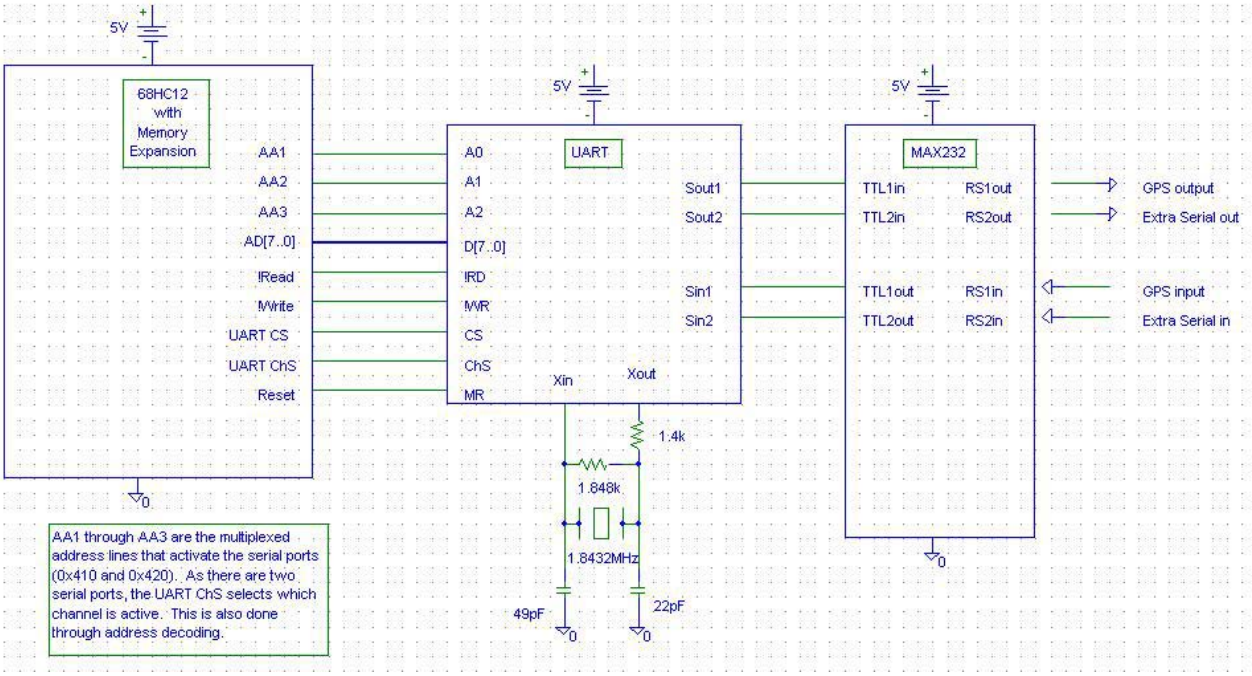


Figure 15: UART Schematics

Compass

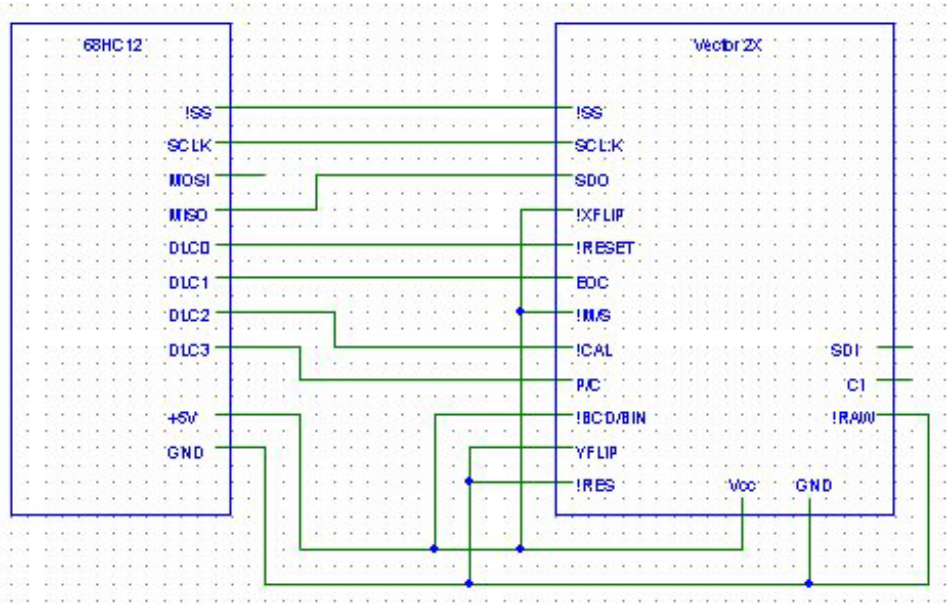


Figure 16: Compass Schematics

Accelerometer

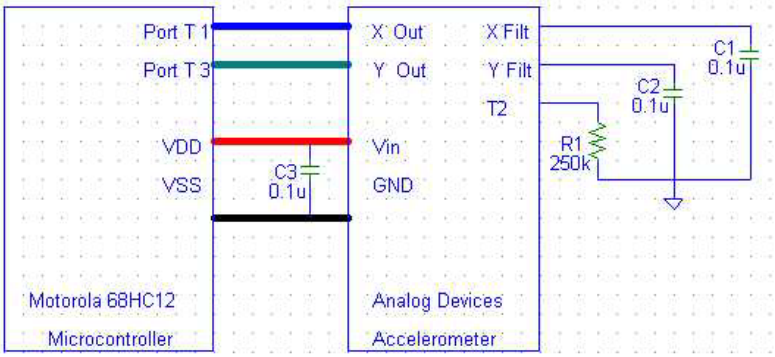


Figure 17: Accelerometer Schematics

Communications

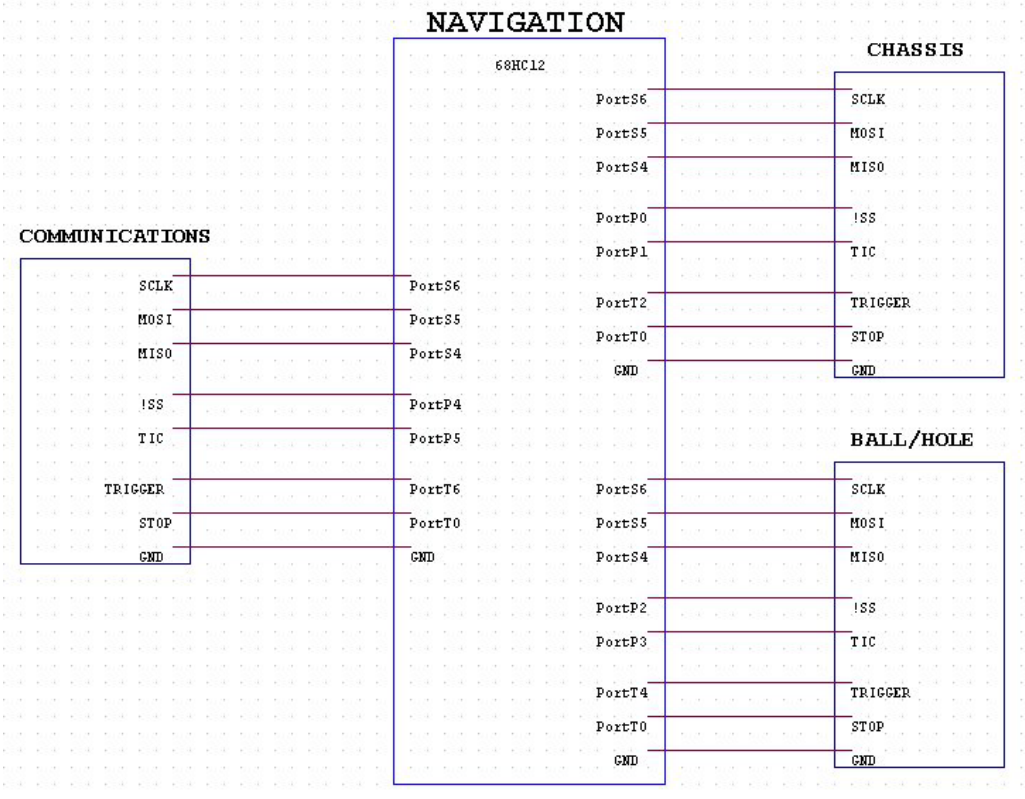


Figure 18: Communications Schematics

Appendix B: Code

This appendix is divided into six sections: GPS, Math Functions, Compass, Accelerometer, Communications, and Complete Program. Code for UART setup is included in the GPS section, and additional UART setup is shown in Appendix C: Altera Code. For ease of understanding some of the setup code, we recommend that you look at Appendix D: References for the manuals and data sheets that describe the settings. The complete code is 1270 lines long. In this Appendix, each part of the code (GPS, Math, Compass, Accelerometer, and Communications) has its own section and starts on a new page; this makes individual pieces of the code easy to locate by page number. The last section in this Appendix is the Complete Code, and is included last as a reference for how the completed code was structured overall.

Slave code, as distributed to the other modules, is included in the communications code section.

GPS

```
void setupuart1 ()                  // this function sets up the UART
{
   DBug12FNP->printf ("uart setup \r\n");
   S1SETUP1 = 0x00;                 // Figure out interrupts
   while ((S1SETUP2 & 0xc0) != 0x00)
      S1SETUP2 |= 0x01;             // turn on FIFOs
   //    S1SETUP2 &= ~0x01;         // turn off FIFOs
   S1SETUP4 = 0x00;                 // turn off some modem crap
   S1SETUP3 = 0x03;                 // set parity and stop settings
   S1SETUP3 |= 0x80;                // enter set data rate mode
   S1SETUP0 = 24;                   // set for 4800
   S1SETUP1 = 0x00;                 // I dunno why, just 'cause
   S1SETUP2 = 0x00;                 // turn off other features
   while ((S1SETUP3 & 0x80) == 0x80)
      S1SETUP3 &= ~0x80;            // set r/w mode

   D (DBug12FNP-> printf ("ch1: dlab %s\r\n",
                          ((S1SETUP3 & 0x80) == 0x80) ? "set" : "reset");)
   D (DBug12FNP-> printf ("ch1: fifos %s\r\n",
                          ((S1SETUP2 & 0xf0) == 0xc0) ? "on" : "off");)
   DBug12FNP->printf ("setup complete: entering read loop\r\n");
}

/*************** get one byte from the serial line *****************/

unsigned char get_gps_byte (short which)
{
   volatile register char status = 0;           // declare as volatile
   if (which == 1)
   {
      status = S1SETUP5;
      if (status & 0x08);           //printf("FE\r\n");
      while ((status & 0x01) != 1)              // check the status
      { status = S1SETUP5; if (status & 0x08); }
      return (S1DATA & ~0x80);
   }

/* deprecated */

   else if (which == 2)
   { while ((S2SETUP5 & 0x01) != 1); return (S2DATA & ~0x80); }
   else /* debugging */
   { _asm (" ldd #$0xdead"); _asm (" swi"); }
}

   /* get an entire string from the durn GPS thing        */
   /* this handles the specifics of beginning and end */
   /* now with checksums and a flavor the whole family can enjoy   */

   /*
      2 examples: this is the shortest a string should be
      $GPGGA,214722,3403.9876,N,10654.4448,W,0,00,,,M,,M,,*41
```

```
      $GPGGA,214724,3403.9876,N,10654.4448,W,0,00,,,M,,M,,*47
   */

short get_gps_cmd ()
{
   short i, j;
   char trash;

   LOOP:
   {
      char checksum = 0, incs = 0;
      for (i = 0; i < 100; i++)     /* clear out this string */
         gpsinstr[i] = '\0';
      gpsinstr[0] = '$';            /* insert 2 characters that for some */
      gpsinstr[1] = 'G';            /* reason don't get there themselves */
      trash = S1DATA;         /* clear the recd bit */

    /* wait for the head of a good string */

   while (get_gps_byte (1) == '$');

/* read bytes from GPS until we see a newline: then we're ready to
check it */

      for (i = 2; (gpsinstr[i] = get_gps_byte (1)) != '\n'; i++);
      gpsinstr[++i] = '\0';   /* just null it ! */

       /* it's long enough, and the checksum matches */
      if (i < 55) goto LOOP;
      for (j = 1; gpsinstr[j] != '*'; j++)
         checksum ^= gpsinstr[j];
      j++;
      incs = (gpsinstr[j] <= 57) ?
             (gpsinstr[j] - 48) * 16 :
             (gpsinstr[j] - 55) * 16;
      j++;
      incs += ((gpsinstr[j] >= 48) && (gpsinstr[j] <= 57)) ?
                gpsinstr[j] - 48 : gpsinstr[j] - 55;

      D(DBug12FNP->printf("\r\nCALC:%x READ:%x\r\n", checksum, incs);)
      if (checksum != incs) goto LOOP;

   }        // end LOOP code

     /* if you got here, it's good, means you got a string */

   return i;
}

   /*
    * at present, the only sentence we are using is GGA
    * $GPGGA,HHMMSS,####.####,(N,S),#####.####,(E,W),G,NT,
              HDOP,####.#,M,####.#,M,,*CS
    * OLD NOTE: .0001' = .58 feet at the equator
    * 4/16: .0001' = .6 AFAIK
    */
```

```
void get_position ()
{
   char *cur;
   short commacount = 0, num_track = 0, i;

/* check for validity first */
   for (*cur = '0'; *cur == '0';)
   {
      get_gps_cmd ();         /* gpsinstr is now valid */

   /* just checking, assume it's good for now */

      DBug12FNP->printf ("%s", gpsinstr);

      /*
      * first order of business is to see if this data is valid.
      * if not, I dunno yet. We need some sort of poll loop and
      * a timeout if that takes too long, I guess. Tell ball/hole
      * to go into the search routine if we can't figure it out.
      *
      * 4/16: AS IS, we just sit around waiting for valid data.
      * not, they're just gonna hafta wait
      */
      for (cur = &(gpsinstr[0]); commacount < 6; cur++)
      if (*cur == ',') commacount++;
   }

   switch (*cur)
   {
       // 0 can't happen now
       // case '0': DBug12FNP->printf("%s", "data no good\r\n"); break;

       case '1':        // is data is good
         DBug12FNP->printf ("%s", "data good\r\n");
         break;

       case '2':        // if data is good and has WAAS satelites
         DBug12FNP->printf ("%s", "data good with WAAS\r\n");
         break;
   }

   /*
    * Okay, the idea here is to count commas over to the data we want,
    * read those characters, subtract their ascii offset from 0, and
    * multiply by the appropriate power of ten. Now, was that so hard?
    */

   /* start this section *** dunt touch ******************************/
   /* get the number of satelites tracking, because I can */

   cur++;
   num_track = ((((*cur) - 0x30) * 10) + (*(cur + 1) - 0x30));
   D (DBug12FNP->printf ("Sats tracked: %d\r\n", num_track);)

   /* now that we know it's good, get the position */

   for (cur = &(gpsinstr[0]), commacount = 0; commacount < 2; cur++)
```

```
   if (*cur == ',') commacount++;
   bot.latdeg = ((*cur - 0x30) * 10) + (*(cur + 1) - 0x30);
   bot.latmin = ((*(cur + 2) - 0x30) * 10) + (*(cur + 3) - 0x30);
   bot.latdecmin = ((*(cur + 5) - 0x30) * 1000) +
                   ((*(cur + 6) - 0x30) * 100) +
                   ((*(cur + 7) - 0x30) * 10) +
                    (*(cur + 8) - 0x30);
   DBug12FNP->printf ("latdeg: %d latmin: %d latdecmin: %d\r\n",
                bot.latdeg, bot.latmin, bot.latdecmin);

   /* next 2.4 is latmin's. wonder how we're going to do this */

   for (; commacount < 4; cur++)
      if (*cur == ',') commacount++;
   bot.longdeg = ((*cur - 0x30) * 100) +
                 ((*(cur + 1) - 0x30) * 10) +
                 ((*(cur + 2) - 0x30));
   bot.longmin = ((*(cur + 3) - 0x30) * 10) + (*(cur + 4) - 0x30);
   bot.longdecmin = ((*(cur + 6) - 0x30) * 1000) +
                    ((*(cur + 7) - 0x30) * 100) +
                    ((*(cur + 8) - 0x30) * 10) +
                     (*(cur + 9) - 0x30);
   DBug12FNP->printf ("longdeg: %d longmin: %d longdecmin: %d\r\n",
      bot.longdeg, bot.longmin, bot.longdecmin);
}           // end the function, we now have coords.
```

Math Functions

```
bool gpsDist (void)
{
   int Dist;
   int flag = 0;
   DifX=0;
   DifY=0;

   DifX = GPSxBOT - GPSxTAR;
   DifY = GPSyBOT - GPSyTAR;

/*********************************************************************
MATH SECTION- Using Pythagorean theorem to find distance.
*********************************************************************/
   if ((abs (DifX) + abs (DifY)) >= 180)
   {
      Dist = (long int) ((((DifX + 5) / 10) * ((DifX + 5) / 10)) +
                  (((DifY + 5) / 10) * ((DifY + 5) / 10)));
      flag = 10;
   }
   else
   {
      Dist = ((DifX * DifX) + (DifY * DifY));
      flag = 1;
   }

/*********************************************************************
Square roots suck on HC12...here is a for loop to  find them.
*********************************************************************/
   for (guess = 0; Dist >= (guess * guess); guess++);
   guess--;
   guess = (guess * flag);
   D (DBug12FNP->printf ("%d difference in long\n\r", DifX);)
   D (DBug12FNP->printf ("%d difference in lat\n\r", DifY);)
   D (DBug12FNP->printf ("%d is distances sqaured and added\n\r", Dist);)
   DBug12FNP->printf ("%d is the Distance to travel\n\r", guess);
   return (guess < 20) ? TRUE : FALSE;
}

/*********************************************************************
Function to find direction given latitude and longitude coordinates
*********************************************************************/
int gpsDir (void)
{

/*********************************************************************
Now me need a table for computing the angle.  We don't have trig
functions, so we write a table with 90 values.  This is for the 90
degrees in each quadrant.  We will add code later to determine which
quadrant we are in.
*********************************************************************/

   int tanTable[44] = { 102, 105, 109, 113, 117, 121, 125, 130, 135,
         140, 151, 157, 163, 169, 176, 184, 192, 200, 209, 219, 230, 242,
         254, 268, 282, 299, 316, 338, 361, 387, 417, 451, 492, 542, 600,
         670, 760, 880, 1045, 1285, 1665, 2380, 4295 };
```

```
   int angle = 0, angleNew = 0;
   int YdivX=0;
   int tableSlct = 0;

/*********************************************************************
Okey dokey...now do the math to find what value we are in the table.
This is our angle
*********************************************************************/

   if (DifY == 0)
   { if (DifX > 0) angle = 90;
   else            angle = 270; }
   else if (DifX == 0)
     {if (DifY > 0) angle = 0;
   else          angle = 180; }
   else
   {
      if (abs (DifY) >= abs (DifX))
      {
            YdivX = (int) (abs (DifY * 100) / abs (DifX));
            D (DBug12FNP-> printf ("%d is Y divided by X, angle > 45\n\r",
            YdivX);)
            tableSlct = 1;
      }
      else
      {
            YdivX = (int) (abs (DifX * 100) / abs (DifY));
            D (DBug12FNP-> printf ("%d is X divided by Y, angle < 45\n\r",
                  YdivX);)
            tableSlct = 2;
      }
      for (angle = 0; (tanTable[angle] < YdivX) && (angle < 45); angle++);

      if (tableSlct == 1) angle += 45;
      else if (tableSlct == 2) angle = 45 - angle;
   }

   D (DBug12FNP->printf ("%d is angle\n\r", angle);)

/*********************************************************************
What quadrant are we in?? Well....
*********************************************************************/

   if ((DifX < 0) && (DifY > 0))    /* First quadrant */
   { angle = (270 - angle);
   D(DBug12FNP->printf ("Q1: rotate %d degrees\n\r", angle);) }

   if ((DifX < 0) && (DifY < 0))    /* Fourth quadrant */
   { angle = (270 + angle);
   D(DBug12FNP->printf ("Q4: rotate %d degrees\n\r", angle);) }

   if ((DifX > 0) && (DifY < 0))    /* Third quadrant */
   { angle = (angle);
   D (DBug12FNP->printf ("Q3: rotate %d degrees\n\r", angle);) }

   if ((DifX > 0) && (DifY > 0))    /* Second quadrant */
   { angle = (90 + angle);
```

```
   D (DBug12FNP->printf ("Q2: rotate %d degrees\n\r", angle);)}

   Bug12FNP->printf ("%d is angle from north to travel\n\r", angle);


   angleNew = (angle - robot_heading);
   if (angleNew < 0) angleNew = (angleNew + 360);
   DBug12FNP->printf ("%d is angle to rotate to!!\n\r", angleNew);
   return angleNew;
}
```

Compass

```
unsigned int robot_heading=0;             // this is a global setup

void setup_compass (void)
{
   DDRDLC = 0x0D;       /* set up input on portDLC as described below */
   PORTDLC = 0x0D;

/* ************************************************************************
Compass is set up with the following connections  (this is the bare minimum):

PORTDLC set up (lines to Vector2x):
PDLC0 = !RESET    ->    OUTPUT high during power up,  and normal operation
PDLC1 = EOC       ->    INPUT, goes low during calcs, high when calculations
                        have completed
PDLC2 = !CAL      ->    OUTPUT high during power up
PDLC3 = !P/C      ->    OUTPUT high during power up, calibration and reset.
                        low for polled data

Lines !XFLIP, !M/S, !RAW, !BCD/BIN are always high (+5V) and are tied
                  to Vdd
Lines YFLIP, !RES are always low (0V) and are tied to GND (Vss)

SPI port setting for the compass interfacing
PS4 = SDO   ->    serial data output, PS4 is the HC12 serial
                  data input (MISO)
PS6 = SCLK  ->    serial (SPI) clock
PS7 = !SS   ->    slave select, high during power up, low to  select
                  slave
************************************************************************** */

   DDRS    = 0xE0;                /* slave select, clock, MOSI outputs */
   PORTS   = PORTS | 0x80;        /* deselect compass, slave line = 1 */
   SP0CR1  = 0x5C;                /* enable SPI and set as master */
   SP0CR2  = 0x00;                /* normal mode, slave in */
   SP0BR   = 0x07;                /* set baud rate to 500kHz */

   TSCR    = 0x80;                 /* enable timer subsystem */
   TMSK2   = 0x01;                 /* set timer overflow rate @ 16ms */
   TFLG2   = 0x80;                 /* clear timer overflow flag */

   delay_16msec ();                /* need time to stabalize the lines */
   delay_16msec ();                /* so run a small delay */
   delay_16msec ();

   PORTDLC = PORTDLC & ~0x01; /* reset goes low for 16ms */

   delay_16msec ();           /* fix power problem from some lines
                                 being set before power on */
   PORTDLC = PORTDLC | 0x01;  /* see compass manual for details
                                 On the process */

DBug12FNP->printf ("Compass is set up\n\r");
}     // end the main function
```

```
/******************CALIBRATE COMPASS FNCN************************ */

void calibrate_compass ()                    /* calibrate the compass */
{
   char finished = 0x00;

   PORTDLC = PORTDLC & ~0x04;          /* set !CAL low */
   delay_16msec ();                    /* run a 16ms delay */
   PORTDLC = PORTDLC | 0x04;           /* set nCAL high */

/* we know need to send a command to chassis to rotate 180 degrees
   and zero distance.  Required by the compass calibration
   procedure */

   while(finished != x024)
   {
      send_data_chassis(0x24);       // tell chassis we are sending a move
                                     // command
      delay_16msec();
      send_data_chassis(0);               // upper bits of angle are 0x00
      delay_16msec();
      send_data_chassis(180);        // 0xb4 = 180 degrees
      delay_16msec();
      send_data_chassis(0x00);            // distance is 0x00

       while(!receiving_chassis);         // wait for finished
                                          // acknowledgement from chassis
      finished = recieve_data_chassis();
   }

   PORTDLC = PORTDLC & ~0x04; /* set nCAL low */
   delay_16msec ();           /* run a 16ms delay */
   PORTDLC = PORTDLC | 0x04;  /* set nCAL high */
   delay_16msec ();           /* run a 16ms delay */

/* inform user that compass is calibrated */

   DBug12FNP->printf ("Vector2X is now calibrated \n\r");

}   // end function

/********************** GET HEADING ****************************/

void getheading ()                   /* heading is 9 bits */
{
   char i = 0, j = 0;
   unsigned char heading1 = 0x00;    /* heading info */
   unsigned char heading2 = 0x00;

   PORTDLC = PORTDLC & ~0x01;        /* reset goes low for 16ms */
   delay_16msec ();
   PORTDLC = PORTDLC | 0x01;         /* reset goes back high */
   while (i < 40)                    /* delay at least 500msec */
   {
      delay_16msec ();
      i++;
```

```
   }

   robot_heading = 0x0000;          /* set heading to zero degrees */

   PORTDLC = PORTDLC & ~0x08; /* bring !P/C low bit 0 of PORTB */
   delay_16msec ();                 /* !P/C stays low >= 10msec */
   PORTDLC = PORTDLC | 0x08;  /* bring !P/C high after 16msec */

/* wait for EOC to go high, compass has then completed calcs. */

   while ((PORTDLC & 0x02) == 0x00);

/* delay at least 10msec before select compass slave line */

   delay_16msec ();
   PORTS = PORTS & ~0x80;             /* select slave (compass) */

   delay_16msec ();

/* get 1st 8 bits from compass, write garbage to SPO line */

   SP0DR = 0x00;
   while ((SP0SR & 0x80) == 0x00);        /* wait for transfer */

   heading1 = SP0DR;                      /* save in temp variable 1 */

   SP0DR = 0x00;                          /* get 2nd 8 bits from compass */
   while ((SP0SR & 0x80) == 0x00);        /* wait for transfer */
   heading2 = SP0DR;                      /* save in temp variable */

   PORTS = PORTS | 0x80;            /* deselect compass, slave line = 1 */

   if (heading1 & 0x01)             /* add 256 degrees if necessary */
   {
      robot_heading = heading2 + 256;
   }
   else                             /* last 8 bits of heading */
   {
      robot_heading = heading2;
   }

   DBug12FNP->printf ("%3u degrees\n\r", robot_heading);
   send_data_comm (0x10);
   for(j=0; j<10; j++)
   {
      delay_16msec();
   }
   send_data_comm ((char) (robot_heading >> 8));
   for(j=0; j<10; j++)
   {
      delay_16msec();
   }
   send_data_comm ((char) (robot_heading));

}  // end function
```

Accelerometer

```
void get_corr ()              /* start accelerometer function */
{
   int accel_cntr  =  0;      /* initialize counter variable    */
   int total       = 40;      /* take 40 readings for averaging */
   unsigned char j =  0;      /* initialize counter variable    */

   disable();         /* disable interrupts to avoid problems with compass */

   TSCR = 0x80;          /* turn on timer subsystem        */
   TIOS = TIOS & ~0x0A; /* TICs 1 and 3 are input capture */

   TFLG1 = 0x0A;        /* clear flags on TICs 1 and 3    */

   for (accel_cntr = 0; accel_cntr < total; accel_cntr++)
   {

   TCTL4 = (TCTL4 | 0x44) & ~0x88; /* 01000100 - TICs 1 and 3 rising edge*/
   while (!(TFLG1 & 0x02));         /* wait for channel 1 flag to be set*/
   TFLG1 = 0x02;              /* clear channel 1 flag  */
   while (!(TFLG1 & 0x0A));         /* wait for channel 3 flag to be set */
   TFLG1 = 0x0A;              /* clear channel 1 and 3 flags */

   Xrise1 = TC1;              /* read in channel 1 value   */
   Yrise1 = TC3;              /* read in channel 3 value   */

   TCTL4 = (TCTL4 | 0x88) & ~0x44;  /* 10001000 - TICs 1 and 3 falling edge*/
   while (!(TFLG1 & 0x02));         /* wait for channel 1 flag to be set */
   TFLG1 = 0x02;                    /* clear channel 1 flag */
   while (!(TFLG1 & 0x0A));         /* wait for channel 3 flag to be set */
   TFLG1 = 0x0A;                    /* clear channel 1 and 3 flags*/

   Xfall = TC1;                     /* read in channel 1 value */
   Yfall = TC3;                     /* read in channel 3 value */
   TCTL4 = (TCTL4 | 0x44) & ~0x88;   /* 01000100 - TICs 1 and 3 rising edge*/
   while (!(TFLG1 & 0x02));   /* wait for channel 1 flag to be set */
   TFLG1 = 0x02;              /* clear channel 1 flag */
   while (!(TFLG1 & 0x0A));   /* wait for channel 3 flag to be set*/
   TFLG1 = 0x0A;              /* clear channel 1 and 3 flags */

   Xrise2 = TC1;              /* read in channel 1 value */
   Yrise2 = TC3;              /* read in channel 3 value */

   TFLG1 = 0x0A;              /* clear channel 1 and 3 flags */

   Xhigh = Xfall - Xrise1;
   /* X high time = Falling Edge - 1st Rising Edge*/

   Yhigh = Yfall - Yrise1;
   /*Y high time = Falling Edge - 1st Rising Edge*/

   Xtotal = Xrise2 - Xrise1;
   /*X period = 2nd Rising Edge - 1st Rising Edge*/

   Ytotal = Yrise2 - Yrise1;
   /* Y period = 2nd Rising Edge - 1st Rising Edge*/
```

```
   /* X Duty Cycle * 1000 = X high / X period */
   Xduty = ((long) Xhigh) * 1000 / ((long) Xtotal);

   /* Y Duty Cycle * 1000 = Y high / Y period */
   Yduty = ((long) Yhigh) * 1000 / ((long) Ytotal);

   /* Acceleration formulas given in datasheet */
   Accel_X += ((((long) Xduty - 477) * 10000) / 1250);
   Accel_Y += ((((long) Yduty - 528) * 10000) / 1250);

   }        // end for loop

   Accel_X /= total;
   /* Divide Accel_X by number of readings to determine average */

   Accel_Y /= total;
   /* Divide Accel_Y by number of readings to determine average */

/*****************************************************************************/
/* Compare acceleration values read to known acceleration values and
determine*/
/* the tilt angle                                                            */
/*****************************************************************************/

   if (Accel_X >= -20 && Accel_X >= 0)
       { Corr_X = -2; }
   else if (Accel_X >= -45 && Accel_X < -20)
       { Corr_X = -4; }
   else if (Accel_X >= -75 && Accel_X < -45)
       { Corr_X = -6; }
   else if (Accel_X >= -120 && Accel_X < -75)
       { Corr_X = -8; }
   else if (Accel_X >= -150 && Accel_X < -120)
       { Corr_X = -10; }
   else if (Accel_X >= -175 && Accel_X < -150)
       { Corr_X = -12; }
   else if (Accel_X >= -225 && Accel_X < -175)
       { Corr_X = -14; }
   else if (Accel_X >= -255 && Accel_X < -225)
       { Corr_X = -16; }
   else if (Accel_X >= -290 && Accel_X < -255)
       { Corr_X = -18; }
   else if (Accel_X >= -315 && Accel_X < -290)
       { Corr_X = -20; }
   else if (Accel_X >= -345 && Accel_X < -315)
       { Corr_X = -22; }
   else if (Accel_X >= -370 && Accel_X < -345)
       { Corr_X = -24; }
   else if (Accel_X >= -400 && Accel_X < -370)
       { Corr_X = -26; }
   else if (Accel_X >= -430 && Accel_X < -400)
       { Corr_X = -28; }
   else if (Accel_X < -430)
       { Corr_X = -30; }

   if (Accel_X <= 30 && Accel_X >= 0)
```

```
    { Corr_X = 0; }
else if (Accel_X <= 50 && Accel_X > 30)
    { Corr_X = 2; }
else if (Accel_X <= 90 && Accel_X > 50)
    { Corr_X = 4; }
else if (Accel_X <= 130 && Accel_X > 90)
    { Corr_X = 6; }
else if (Accel_X <= 160 && Accel_X > 130)
    { Corr_X = 8; }
else if (Accel_X <= 200 && Accel_X > 160)
    { Corr_X = 10; }
else if (Accel_X <= 230 && Accel_X > 200)
    { Corr_X = 12; }
else if (Accel_X <= 265 && Accel_X > 230)
    { Corr_X = 14; }
else if (Accel_X <= 300 && Accel_X > 265)
    { Corr_X = 16; }
else if (Accel_X <= 330 && Accel_X > 300)
    { Corr_X = 18; }
else if (Accel_X <= 360 && Accel_X > 330)
    { Corr_X = 20; }
else if (Accel_X <= 400 && Accel_X > 360)
    { Corr_X = 22; }
else if (Accel_X <= 440 && Accel_X > 400)
    { Corr_X = 24; }
else if (Accel_X <= 470 && Accel_X > 440)
    { Corr_X = 26; }
else if (Accel_X <= 500 && Accel_X > 470)
    { Corr_X = 28; }
else if (Accel_X > 500)
    { Corr_X = 30; }


if (Accel_Y >= -20 && Accel_Y >= 0)
    { Corr_Y = -2; }
else if (Accel_Y >= -45 && Accel_Y < -20)
    { Corr_Y = -4; }
else if (Accel_Y >= -75 && Accel_Y < -45)
    { Corr_Y = -6; }
else if (Accel_Y >= -120 && Accel_Y < -75)
    { Corr_Y = -8; }
else if (Accel_Y >= -150 && Accel_Y < -120)
    { Corr_Y = -10; }
else if (Accel_Y >= -175 && Accel_Y < -150)
    { Corr_Y = -12; }
else if (Accel_Y >= -225 && Accel_Y < -175)
    { Corr_Y = -14; }
else if (Accel_Y >= -255 && Accel_Y < -225)
    { Corr_Y = -16; }
else if (Accel_Y >= -290 && Accel_Y < -255)
    { Corr_Y = -18; }
else if (Accel_Y >= -315 && Accel_Y < -290)
    { Corr_Y = -20; }
else if (Accel_Y >= -345 && Accel_Y < -315)
    { Corr_Y = -22; }
else if (Accel_Y >= -370 && Accel_Y < -345)
    { Corr_Y = -24; }
```

```
   else if (Accel_Y >= -400 && Accel_Y < -370)
       { Corr_Y = -26; }
   else if (Accel_Y >= -430 && Accel_Y < -400)
       { Corr_Y = -28; }
   else if (Accel_Y < -430)
       { Corr_Y = -30; }

   if (Accel_Y <= 30 && Accel_Y >= 0)
       { Corr_Y = 0; }
   else if (Accel_Y <= 50 && Accel_Y > 30)
       { Corr_Y = 2; }
   else if (Accel_Y <= 90 && Accel_Y > 50)
       { Corr_Y = 4; }
   else if (Accel_Y <= 130 && Accel_Y > 90)
       { Corr_Y = 6; }
   else if (Accel_Y <= 160 && Accel_Y > 130)
       { Corr_Y = 8; }
   else if (Accel_Y <= 200 && Accel_Y > 160)
       { Corr_Y = 10; }
   else if (Accel_Y <= 230 && Accel_Y > 200)
       { Corr_Y = 12; }
   else if (Accel_Y <= 265 && Accel_Y > 230)
       { Corr_Y = 14; }
   else if (Accel_Y <= 300 && Accel_Y > 265)
       { Corr_Y = 16; }
   else if (Accel_Y <= 330 && Accel_Y > 300)
       { Corr_Y = 18; }
   else if (Accel_Y <= 360 && Accel_Y > 330)
       { Corr_Y = 20; }
   else if (Accel_Y <= 400 && Accel_Y > 360)
       { Corr_Y = 22; }
   else if (Accel_Y <= 440 && Accel_Y > 400)
       { Corr_Y = 24; }
   else if (Accel_Y <= 470 && Accel_Y > 440)
       { Corr_Y = 26; }
   else if (Accel_Y <= 500 && Accel_Y > 470)
       { Corr_Y = 28; }
   else if (Accel_Y > 500)
       { Corr_Y = 30; }

/*************************************************************************/
/* Use Corr_X and Corr_Y to correct heading error based experimentation    */
/* Send corrected heading information to communications group              */
/*************************************************************************/

   robot_heading = robot_heading + 2*Corr_X;
   robot_heading = robot_heading + 2*Corr_Y;

/* send corrected heading to comm. Group */

   DBug12FNP->printf("Corrected Heading = %d \n\r", robot_heading);

   send_data_comm (0x10);           // ID byte to comm
   for(j=0; j<10; j++)
      {delay_16msec();}

   /*Send upper byte of heading*/
```

```
   send_data_comm ((char) (robot_heading >> 8)); for(j=0; j<10; j++)
      {delay_16msec(); }

   /* Send lower byte of heading */
   send_data_comm ((char) (robot_heading));

   enable();            /* re-enable HC12 interrupts */

}
```

Communications

```
// these are global setup variables

typedef enum { false, true } bool;

char RECEIVED = 0;
char DATA;

/* communication variables */
volatile bool sending_chassis = FALSE;
volatile bool sending_comm = FALSE;
volatile bool sending_ball_hole = FALSE;

volatile bool ready_chassis = FALSE;
volatile bool ready_comm = FALSE;
volatile bool ready_ball_hole = FALSE;

volatile bool receiving_chassis = FALSE;
volatile bool receiving_comm = FALSE;
volatile bool receiving_ball_hole = FALSE;


/*************** COMMUNICATIONS SETUP ******************************/
void comm_init ()
{
   DDRP = 0xff;                  /* all bits of port P are output */
   PORTP = 0x15;                 /* portP initially is 00010101 */

/* this deselects the slave and trigger (all handshake lines low */

   TSCR = 0x80;                  /* enable timer subsystem */
   TIOS = TIOS & ~0x02;          /* 0,2,3,4,5,6,7 to input capture */
   TCTL4 = (TCTL4 | 0x51) & 0x51;/* all capture the rising edge */
   TCTL3 = (TCTL3 | 0x55) & 0x55;
   TMSK1 = TMSK1 | ~0x02;        /* 0,2,3,4,5,6,7  interrupts */
   TMSK2 = 0x01;                 /* overflow rate at 16ms */

/* set up the SPI to communicate with the other subsystems */

   DDRS = DDRS | 0xE0;           /* ss, clk, MOSI outputs */
   PORTS = PORTS | 0x80;         /* deselct slave, for compass */

/* PORTP is !SS for other modules */
/* master device, MSB first, etc, idle high, valid on falling edge */

   SP0CR1 = SP0CR1 | 0x5c;

/* this is set up to match the compass needs */

   SP0CR2 = 0x00;

/* baud rate at 31.3kHz, also, this is
/* fastest the comm module can handle */

   SP0BR = 0x07;
```

```
   DBug12FNP->printf("Finished Communications setup \n\r");

}  // end function

   /*********************************************************************/
   /* Setup of the module communications are as follows: ****************/
   /*                                                                   */
   /* BALL/HOLE:  slave select: PORTP2                                  */
   /*             interrupt: TIMER4                                     */
   /*             handshake: PORTP3                                     */
   /*-------------------------------------------------------------------*/
   /* CHASSIS:    slave select: PORTP0                                  */
   /*             interrupt: TIMER4                                     */
   /*             handshake: PORTP1                                     */
   /*-------------------------------------------------------------------*/
   /* COMMUNICATIONS:     slave select: PORTP4                          */
   /*                     interrupt: TIMER6                             */
   /*                     handshake: not used for communications        */
   /*********************************************************************/

/*********************************************************************/
/* SEND TO ball/hole                                                 */
/*********************************************************************/

void send_data_ball_hole (DATA)
{
   char garbage_received = 0x00;
   sending_ball_hole = TRUE;

   delay_16msec ();        /* short delay to make sure evertone is ready */

   PORTP = PORTP & ~0x14;        /* slave select both ball/hole and comm */
   SP0DR = DATA;                 /* send 8-bit (char) average */
   while ((SP0SR & 0x80) == 0);  /* wait for transfer */
   garbage_received = SP0DR;
   PORTP = PORTP | 0x14;         /* this line slave deselects both the
                                    ball/hole and comm channels */

   DBug12FNP->printf("Sent %x \n\r", DATA);

   ready_ball_hole = FALSE;
   sending_ball_hole = FALSE;
}

/*********************************************************************/
/* RECEIVE FROM ball/hole                                            */
/*********************************************************************/

char receive_data_ball_hole ()      /* receive from ball/hole module */
{
   char RECEIVED = 0x00;            /* initialize variables */

   receiving_ball_hole = TRUE;
   PORTP = PORTP & ~0x04;     /* select slave line ports0 */
   SP0DR = GARBAGE;           /* send 8-bit (char) average */
   while ((SP0SR & 0x80) == 0x00);  /* wait for transfer */
```

```
   RECEIVED = SP0DR;          /* save the received data */
   PORTP = PORTP | 0x04;            /* deselect SLAVE */

   receiving_ball_hole = FALSE;

   D(DBug12FNP->printf("Receiving \n\r");)

   return RECEIVED;           /* return the received data */
}  // end function

/*****************************************************************/
/* SEND TO comm                                                  */
/*****************************************************************/

void send_data_comm (DATA)
{
   char garbage_received = 0x00;
   sending_comm = TRUE;

   PORTP = PORTP & ~0x10;           // slave select comm module
   SP0DR = DATA;              // load data into SPI
   while ((SP0SR & 0x80) == 0);           // wait for transfer to finish
   garbage_received = SP0DR;        // clear the SPI flag by reading garbage
   PORTP = PORTP | 0x10;            // deselect comm module

   ready_comm = FALSE;        // finished sending to comm, so clear the flags
to FALSE
   sending_comm = FALSE;
}

/*****************************************************************/
/* RECEIVE FROM comm                                             */
/*****************************************************************/

char receive_data_comm (void)
{
   char RECEIVED = 0;

   PORTP = PORTP & ~0x10;           /* select comm module */
   SP0DR = GARBAGE;                 /* send garbage to start Xfer */
   while ((SP0SR & 0x80) == 0x00);        /* wait for transfer to finish */
   RECEIVED = SP0DR;                /* store the received data */
   PORTP = PORTP | 0x10;            /* deselect comm module */

   receiving_comm = FALSE;          /* clear the flag */
   return RECEIVED;                 /* return received data */
}

/*****************************************************************/
/* SEND TO chassis                                               */
/*****************************************************************/

void send_data_chassis (DATA)
{
   char garbage_received = 0x00;
```

```
   DBug12FNP->printf("in send_data_chassis\r\n");
   sending_chassis = TRUE;

   PORTP = PORTP | 0x02;      /* set interrupt trigger line to chassis */
   while (!ready_chassis);    /* wait for the acknowledgement */
   PORTP = PORTP & ~0x02;     /* drop the handshake line */
   DBug12FNP->printf("out of handshake\r\n");

   delay_16msec ();

   PORTP = PORTP & ~0x11;      /* select both chassis and comm */

   SP0DR = DATA;                /* load data into SPI data register, and
                                   start the transfer */
   while ((SP0SR & 0x80) == 0);          // wait for transfer to finish
   garbage_received = SP0DR;  // read the garbage to clear the SPI flag
   PORTP = PORTP | 0x11;           // deselect chassis and comm

   ready_chassis = FALSE;        // clear the transmission flags
   sending_chassis = FALSE;
}

/*********************************************************************/
/* RECEIVE FROM chassis                                              */
/*********************************************************************/

char receive_data_chassis (void)
{
   char RECEIVED = 0;

   PORTP = PORTP & ~0x01;        // slave select the chassis module
   SP0DR = GARBAGE;              // send garbage to start Xfer
   while ((SP0SR & 0x80) == 0x00);       // wait for Xfer
   RECEIVED = SP0DR;             // store received data
   PORTP = PORTP | 0x01;         // deselect the slave device

   receiving_chassis = FALSE;    // clear the transmission flags
   return RECEIVED;
}

/*********************************************************************/
/* INTERRUPT SERVICE ROUTINES                                        */
/*********************************************************************/

@interrupt tic0_isr ()        /* dedicated STOP/PAUSE line from comm */
{
   while ((PORTT & 0x01) == 0x00);  // do nothing while line is high
   TFLG1 = 0x01;                    // clear the flag
}

/* CHANNEL 2 INTERRUPT */
@interrupt tic2_isr (void)
{
   if (sending_chassis) ready_chassis = TRUE;   // set ready if sending
   else receiving_chassis = TRUE;               // we are receiving
   TFLG1 = 0x04;                                // clear flag
}
```

```
/* CHANNEL 4 INTERRUPT */
@interrupt tic4_isr (void)
{
   if (sending_ball_hole) ready_ball_hole = TRUE; // set ready if sending
   else receiving_ball_hole = TRUE;               // we are receiving
   TFLG1 = 0x10;                                  // clear flag
}

/* CHANNEL 6 INTERRUPT */
@interrupt tic6_isr (void)
{
   if (sending_comm) ready_comm = TRUE;         // set ready if sending
   else receiving_comm = TRUE;                  // we are receiving
   TFLG1 = 0x40;                                // clear flag
}

/* nulls for safety... and sanity */
@interrupt tic7_isr () { TFLG1 = 0x80; }
@interrupt tic5_isr () { TFLG1 = 0x20; }
@interrupt tic3_isr () { TFLG1 = 0x08; }
```

Slave Code

```
/*
This is the code provided by the NAV A team for Robot A communications
between subsystems.  This is slightly different from the version sent out
yesterday, in that is has comments so the you can easily follow the code, and
a small error has been fixed.

The port and timer channel setup in this file is extremely flexible.  I set
it up this way on mine because I will likely be using these ports and
channels, but you can use anything you want, so long as the code is still
functional.

Thanks to Azmat from Ball/Hole for helping beat out some of the bugs.

Code is provided as is.  No warranty expressed or implied.  Void where
prohibited.  Contact your representative for more information.  Not valid in
Alaska, Hawaii, or Puerto Rico.

                                                          - Ryan
*/

#include "hc12.h"
#include "DBug12.h"

#define TRUE 1
#define FALSE 0
#define GARBAGE 0x00          // define garbage value to send for some xfers

typedef enum {false,true} bool;     //  This line defines boolean variables

char DATA = 0x00;             // initialize global data variable
char RECEIVED = 0x00;
```

```
volatile bool receive = FALSE;
volatile bool sending = FALSE;
volatile bool finished = FALSE;


main()
{
   DDRP = 0x40;         // code is setup to use PORTP (PWM port) if you are
   PORTP = 0x00;        // using this port, just change this to another, such
                        // as A, VB, ExpA., ExpB, A/D, whatever...

   TSCR = 0x80;            // enable timer system
   TIOS = 0x00;            // set up timer channel 2 and 3 to input capture
   TCTL4 = 0x10;           // channel 2 input capture rising
   TMSK1 = 0x04;           // enable channel 2 interrupts
   TMSK2 = 0x01;           // overflow rate at 16ms

// set up the SPI to communicate with the other subsystems

   DDRS = DDRS | 0x10;        // ss, clk, MOSI outputs, MISO is input
   SP0CR1 = 0x4c;             // slave, MSB first, etc.
   SP0CR2 = 0x00;
   // no baud rate register since master provides the clock

   DATA = 0xAA;                                  // initiaiize test data
   enable();
   DBug12FNP->printf("Waiting \n\r");

   receive = FALSE;
   while(TRUE)                // infinite test loop to receive data
   {
      receive = FALSE;
      while(!receive);
      receive_data();
      DBug12FNP->printf("received %u, %x \n\r", RECEIVED, RECEIVED);

   }

}

//*****************************RECEIVE DATA ********************************
void receive_data(void)
{
   SP0DR = 0xAA;                           // send back garbage
   DBug12FNP->printf("Receiving \n\r");

   PORTP = PORTP | 0x01;                   // raise the acknowledgement line

   while ((SP0SR & 0x80) == 0);            // wait for transfer to finish
   RECEIVED = SP0DR;                       // read out the received data

   PORTP = PORTP & ~0x01;                        // drop handshake

   receive = FALSE;                        // done receiving

}
```

```
//*******************************SEND DATA**********************************

void send_data(DATA)
{

   char crap_received;
   DBug12FNP->printf("Sending %u \n\r", DATA);

   SP0DR = DATA;

   PORTP = PORTP | 0x01;     // raise line to tell master we want to transfer

   while ((SP0SR & 0x80) == 0);            // wait for transfer
   crap_received = SP0DR;                  // clear spi flag

   PORTP = PORTP & ~0x01;                  // drop handshake line

}

//***************************************************************************
@interrupt tic2_isr(void)           // the timer channel 2, Input Capture
{

   DBug12FNP->printf("TIC2 ISR \n\r");

   receive = TRUE;

   TFLG1 = 0x04;                    // clear the channel 2 flag

}

//***************************************************************************
void delay_16msec(void)                    /* set up 16msec delay */
{
   TFLG2 = 0x80;                    /* clear timer interrupt flag */

   while(!(TFLG2 & 0x80));                /* wait for timer flag */
   TFLG2 = 0x80;                    /* clear timer interrupt flag */
}

//***************************************************************************
void delay_3sec(void)
{
   char i = 0;                      /* set counter to 0 */

   TFLG2 = 0x80;                    /* clear timer overflow flag */

   while(i < 188)                   /* wait for 118 16ms overflows */
   {
      while(!(TFLG2 & 0x80));       /* wait for timer flag */
      TFLG2 = 0x80;                 /* clear timer interrupt flag */
      i++;
   }
}
```

Complete Program

```
#include "hc12.h"
#include "DBug12.h"
#include "uart.h"

/* I've seen where this matters to define thing this way */
#define FALSE 0
#define TRUE !FALSE                 /* define TRUE with value 1 */
#define GARBAGE 0x00                /* SPI default send value   */

//#define ballhole                  /* ball hole stuff active */
//#define comm                      /* comm stuff active */
//#define chassis                   /* chassis stuff active */
//#define gpstest                   /* closed loop gps test */

/* to find: DC 4/4/02 */

//#define GPSxTAR 4572+40
//#define GPSyTAR 9084+640
//#define GPSxTAR 8135
//#define GPSyTAR 582
//#define GPSxTAR 4500
//#define GPSyTAR 9831

int GPSxTAR, GPSyTAR;
#define spin() move(0xff, 0)        // spin define after dropping the ball

/* XXX */
//#define D(x) x              // this defines some custom "comment" blocks
#define D(x)

unsigned int robot_heading=0;       // initialize robot_heading globally

int DifX = 0, DifY = 0;             // initialize difference globally
int guess = 0;

unsigned char gpsinstr[100];
struct position { short latdeg, latmin, latdecmin,  longdeg, longmin,
                  longdecmin; };
struct position bot, target;

#define GPSxBOT bot.longdecmin
#define GPSyBOT bot.latdecmin

typedef enum { false, true } bool;        // Boolean definitions
bool ball_close = FALSE;
bool hole_close = FALSE;

char RECEIVED = 0;
char DATA, DATA2, DATA3;

/* communication variables, described in comm. functions */
volatile bool sending_chassis = FALSE;
volatile bool sending_comm = FALSE;
volatile bool sending_ball_hole = FALSE;
```

```
volatile bool ready_chassis = FALSE;
volatile bool ready_comm = FALSE;
volatile bool ready_ball_hole = FALSE;

volatile bool receiving_chassis = FALSE;
volatile bool receiving_comm = FALSE;
volatile bool receiving_ball_hole = FALSE;

/* flag to mark arrivial */
bool not_there;

/* accelerometer variables */
int Xrise1, Xrise2, Xfall, Xhigh, Xtotal, Xduty, Accel_X,
    Corr_X = 0, Yrise1, Yrise2, Yfall, Yhigh, Ytotal, Yduty, Accel_Y,
    Corr_Y = 0;

/* can I inline this? */
int abs (int x) { return (x > 0 ? x : -x); }

// *************** end global setup *********************


/* GPS thingy ************************************************************/

void setupuart1 ()
{
   DBug12FNP->printf ("uart setup \r\n");
   S1SETUP1 = 0x00;           // Figure out interrupts
   while ((S1SETUP2 & 0xc0) != 0x00)
      S1SETUP2 |= 0x01;             // turn on FIFOs
   //    S1SETUP2 &= ~0x01;         // turn off FIFOs
   S1SETUP4 = 0x00;                 // turn off some modem crap
   S1SETUP3 = 0x03;                 // set parity and stop settings
   S1SETUP3 |= 0x80;                // enter set data rate mode
   S1SETUP0 = 24;                   // set for 4800
   S1SETUP1 = 0x00;                 // I dunno why, just 'cause
   S1SETUP2 = 0x00;                 // turn off other features
   while ((S1SETUP3 & 0x80) == 0x80)
      S1SETUP3 &= ~0x80;            // set r/w mode

   D (DBug12FNP-> printf ("ch1: dlab %s\r\n",
                          ((S1SETUP3 & 0x80) == 0x80) ? "set" : "reset");)
   D (DBug12FNP-> printf ("ch1: fifos %s\r\n",
                          ((S1SETUP2 & 0xf0) == 0xc0) ? "on" : "off");)
   DBug12FNP->printf ("setup complete: entering read loop\r\n");
}

/*************** get one byte from the serial line *****************/

unsigned char get_gps_byte (short which)
{
   volatile register char status = 0;           // declare as volatile
   if (which == 1)
   {
      status = S1SETUP5;
      if (status & 0x08);     //printf("FE\r\n");
      while ((status & 0x01) != 1)
```

```
      { status = S1SETUP5; if (status & 0x08); }
      return (S1DATA & ~0x80);
   }

/* deprecated */

   else if (which == 2)
   { while ((S2SETUP5 & 0x01) != 1); return (S2DATA & ~0x80); }
   else /* debugging */
   { _asm (" ldd #$0xdead"); _asm (" swi"); }
}

   /* get an entire string from the durn GPS thing        */
   /* this handles the specifics of beginning and end */
   /* now with checksums and a flavor the whole family can enjoy   */

   /*
      2 examples: this is the shortest a string should be
      $GPGGA,214722,3403.9876,N,10654.4448,W,0,00,,,M,,M,,*41
      $GPGGA,214724,3403.9876,N,10654.4448,W,0,00,,,M,,M,,*47
   */
short get_gps_cmd ()
{
   short i, j;
   char trash;

   LOOP:
   {
      char checksum = 0, incs = 0;
      for (i = 0; i < 100; i++)     /* clear out this string */
         gpsinstr[i] = '\0';
      gpsinstr[0] = '$';            /* insert 2 characters that for some */
      gpsinstr[1] = 'G';            /* reason don't get there themselves */
      trash = S1DATA;         /* clear the recd bit */

    /* wait for the head of a good string */

   while (get_gps_byte (1) == '$');

/* read bytes from GPS until we see a newline: then we're ready to
check it */

      for (i = 2; (gpsinstr[i] = get_gps_byte (1)) != '\n'; i++);
      gpsinstr[++i] = '\0';   /* just null it ! */

       /* it's long enough, and the checksum matches */
      if (i < 55) goto LOOP;
      for (j = 1; gpsinstr[j] != '*'; j++)
         checksum ^= gpsinstr[j];
      j++;
      incs = (gpsinstr[j] <= 57) ?
             (gpsinstr[j] - 48) * 16 :
             (gpsinstr[j] - 55) * 16;
      j++;
      incs += ((gpsinstr[j] >= 48) && (gpsinstr[j] <= 57)) ?
                gpsinstr[j] - 48 : gpsinstr[j] - 55;
```

```
      D(DBug12FNP->printf("\r\nCALC:%x READ:%x\r\n", checksum, incs);)
      if (checksum != incs) goto LOOP;

   }        // end LOOP code

     /* if you got here, it's good, means you got a string */

   return i;
}

   /*
    * at present, the only sentence we are using is GGA
    * $GPGGA,HHMMSS,####.####,(N,S),#####.####,(E,W),G,NT,
             HDOP,####.#,M,####.#,M,,*CS
    * OLD NOTE: .0001' = .58 feet at the equator
    * 4/16: .0001' = .6 AFAIK
    */

void get_position ()
{
   char *cur;
   short commacount = 0, num_track = 0, i;

/* check for validity first */
   for (*cur = '0'; *cur == '0';)
   {
      get_gps_cmd ();         /* gpsinstr is now valid */

   /* just checking, assume it's good for now */

      DBug12FNP->printf ("%s", gpsinstr);

      /*
      * first order of business is to see if this data is valid.
      * if not, I dunno yet. We need some sort of poll loop and
      * a timeout if that takes too long, I guess. Tell ball/hole
      * to go into the search routine if we can't figure it out.
      *
      * 4/16: AS IS, we just sit around waiting for valid data.
      * not, they're just gonna hafta wait
      */
      for (cur = &(gpsinstr[0]); commacount < 6; cur++)
      if (*cur == ',') commacount++;
   }

   switch (*cur)
   {
       // 0 can't happen now
       // case '0': DBug12FNP->printf("%s", "data no good\r\n"); break;

       case '1':        // is data is good
         DBug12FNP->printf ("%s", "data good\r\n");
         break;

       case '2':        // if data is good and has WAAS satelites
         DBug12FNP->printf ("%s", "data good with WAAS\r\n");
         break;
```

```
   }

   /*
    * Okay, the idea here is to count commas over to the data we want,
    * read those characters, subtract their ascii offset from 0, and
    * multiply by the appropriate power of ten. Now, was that so hard?
    */

   /* start this section *** dunt touch ****************************/
   /* get the number of satelites tracking, because I can */

   cur++;
   num_track = ((((*cur) - 0x30) * 10) + (*(cur + 1) - 0x30));
   D (DBug12FNP->printf ("Sats tracked: %d\r\n", num_track);)

   /* now that we know it's good, get the position */

   for (cur = &(gpsinstr[0]), commacount = 0; commacount < 2; cur++)
   if (*cur == ',') commacount++;
   bot.latdeg = ((*cur - 0x30) * 10) + (*(cur + 1) - 0x30);
   bot.latmin = ((*(cur + 2) - 0x30) * 10) + (*(cur + 3) - 0x30);
   bot.latdecmin = ((*(cur + 5) - 0x30) * 1000) +
                   ((*(cur + 6) - 0x30) * 100) +
                   ((*(cur + 7) - 0x30) * 10) +
                    (*(cur + 8) - 0x30);
   DBug12FNP->printf ("latdeg: %d latmin: %d latdecmin: %d\r\n",
                 bot.latdeg, bot.latmin, bot.latdecmin);

   /* next 2.4 is latmin's. wonder how we're going to do this */

   for (; commacount < 4; cur++)
      if (*cur == ',') commacount++;
   bot.longdeg = ((*cur - 0x30) * 100) +
                 ((*(cur + 1) - 0x30) * 10) +
                 ((*(cur + 2) - 0x30));
   bot.longmin = ((*(cur + 3) - 0x30) * 10) + (*(cur + 4) - 0x30);
   bot.longdecmin = ((*(cur + 6) - 0x30) * 1000) +
                    ((*(cur + 7) - 0x30) * 100) +
                    ((*(cur + 8) - 0x30) * 10) +
                     (*(cur + 9) - 0x30);
   DBug12FNP->printf ("longdeg: %d longmin: %d longdecmin: %d\r\n",
      bot.longdeg, bot.longmin, bot.longdecmin);
}           // end the function, we now have coords.
```

/********** MATH FUNCTIONS **********/

```
bool gpsDist (void)
{
   int Dist;
   int flag = 0;
   DifX=0;
   DifY=0;

   DifX = GPSxBOT - GPSxTAR;
   DifY = GPSyBOT - GPSyTAR;
```

```
/*********************************************************************
MATH SECTION- Using Pythagorean theorem to find distance.
*********************************************************************/
   if ((abs (DifX) + abs (DifY)) >= 180)
   {
      Dist = (long int) ((((DifX + 5) / 10) * ((DifX + 5) / 10)) +
                  (((DifY + 5) / 10) * ((DifY + 5) / 10)));
      flag = 10;
   }
   else
   {
      Dist = ((DifX * DifX) + (DifY * DifY));
      flag = 1;
   }

/*********************************************************************
Square roots suck on HC12...here is a for loop to  find them.
*********************************************************************/
   for (guess = 0; Dist >= (guess * guess); guess++);
   guess--;
   guess = (guess * flag);
   D (DBug12FNP->printf ("%d difference in long\n\r", DifX);)
   D (DBug12FNP->printf ("%d difference in lat\n\r", DifY);)
   D (DBug12FNP->printf ("%d is distances sqaured and added\n\r", Dist);)
   DBug12FNP->printf ("%d is the Distance to travel\n\r", guess);
   return (guess < 20) ? TRUE : FALSE;
}

/*********************************************************************
Function to find direction given latitude and longitude coordinates
*********************************************************************/
int gpsDir (void)
{

/*********************************************************************
Now me need a table for computing the angle.  We don't have trig
functions, so we write a table with 90 values.  This is for the 90
degrees in each quadrant.  We will add code later to determine which
quadrant we are in.
*********************************************************************/

   int tanTable[44] = { 102, 105, 109, 113, 117, 121, 125, 130, 135,
         140, 151, 157, 163, 169, 176, 184, 192, 200, 209, 219, 230, 242,
         254, 268, 282, 299, 316, 338, 361, 387, 417, 451, 492, 542, 600,
         670, 760, 880, 1045, 1285, 1665, 2380, 4295 };
   int angle = 0, angleNew = 0;
   int YdivX=0;
   int tableSlct = 0;

/*********************************************************************
Okey dokey...now do the math to find what value we are in the table.
This is our angle
*********************************************************************/

   if (DifY == 0)
   { if (DifX > 0) angle = 90;
   else            angle = 270; }
```

```
   else if (DifX == 0)
     {if (DifY > 0) angle = 0;
   else          angle = 180; }
   else
   {
      if (abs (DifY) >= abs (DifX))
      {
            YdivX = (int) (abs (DifY * 100) / abs (DifX));
            D (DBug12FNP-> printf ("%d is Y divided by X, angle > 45\n\r",
            YdivX);)
            tableSlct = 1;
      }
      else
      {
            YdivX = (int) (abs (DifX * 100) / abs (DifY));
            D (DBug12FNP-> printf ("%d is X divided by Y, angle < 45\n\r",
                  YdivX);)
            tableSlct = 2;
      }
      for (angle = 0; (tanTable[angle] < YdivX) && (angle < 45); angle++);

      if (tableSlct == 1) angle += 45;
      else if (tableSlct == 2) angle = 45 - angle;
   }

   D (DBug12FNP->printf ("%d is angle\n\r", angle);)

/*********************************************************************
What quadrant are we in?? Well....
*********************************************************************/

   if ((DifX < 0) && (DifY > 0))    /* First quadrant */
   { angle = (270 - angle);
   D(DBug12FNP->printf ("Q1: rotate %d degrees\n\r", angle);) }

   if ((DifX < 0) && (DifY < 0))    /* Fourth quadrant */
   { angle = (270 + angle);
   D(DBug12FNP->printf ("Q4: rotate %d degrees\n\r", angle);) }

   if ((DifX > 0) && (DifY < 0))    /* Third quadrant */
   { angle = (angle);
   D (DBug12FNP->printf ("Q3: rotate %d degrees\n\r", angle);) }

   if ((DifX > 0) && (DifY > 0))    /* Second quadrant */
   { angle = (90 + angle);
   D (DBug12FNP->printf ("Q2: rotate %d degrees\n\r", angle);)}

   Bug12FNP->printf ("%d is angle from north to travel\n\r", angle);


   angleNew = (angle - robot_heading);
   if (angleNew < 0) angleNew = (angleNew + 360);
   DBug12FNP->printf ("%d is angle to rotate to!!\n\r", angleNew);
   return angleNew;
}
/*********************************************************************/
/************************ COMPASS FUNCTIONS ***************************/
```

```
unsigned int robot_heading=0;             // this is a global setup

void setup_compass (void)
{
   DDRDLC = 0x0D;       /* set up input on portDLC as described below */
   PORTDLC = 0x0D;

/* ************************************************************************
Compass is set up with the following connections  (this is the bare minimum):

PORTDLC set up (lines to Vector2x):
PDLC0 = !RESET    ->    OUTPUT high during power up,  and normal operation
PDLC1 = EOC       ->    INPUT, goes low during calcs, high when calculations
                        have completed
PDLC2 = !CAL      ->    OUTPUT high during power up
PDLC3 = !P/C      ->    OUTPUT high during power up, calibration and reset.
                        low for polled data

Lines !XFLIP, !M/S, !RAW, !BCD/BIN are always high (+5V) and are tied
                  to Vdd
Lines YFLIP, !RES are always low (0V) and are tied to GND (Vss)

SPI port setting for the compass interfacing
PS4 = SDO   ->    serial data output, PS4 is the HC12 serial
                  data input (MISO)
PS6 = SCLK  ->    serial (SPI) clock
PS7 = !SS   ->    slave select, high during power up, low to  select
                  slave
************************************************************************** */

   DDRS    = 0xE0;                /* slave select, clock, MOSI outputs */
   PORTS   = PORTS | 0x80;        /* deselect compass, slave line = 1 */
   SP0CR1  = 0x5C;                /* enable SPI and set as master */
   SP0CR2  = 0x00;                /* normal mode, slave in */
   SP0BR   = 0x07;                /* set baud rate to 500kHz */

   TSCR    = 0x80;                 /* enable timer subsystem */
   TMSK2   = 0x01;                 /* set timer overflow rate @ 16ms */
   TFLG2   = 0x80;                 /* clear timer overflow flag */

   delay_16msec ();                /* need time to stabalize the lines */
   delay_16msec ();                /* so run a small delay */
   delay_16msec ();

   PORTDLC = PORTDLC & ~0x01; /* reset goes low for 16ms */

   delay_16msec ();           /* fix power problem from some lines
                                 being set before power on */
   PORTDLC = PORTDLC | 0x01;  /* see compass manual for details
                                 On the process */

DBug12FNP->printf ("Compass is set up\n\r");
}     // end the main function

/*****************CALIBRATE COMPASS FNCN************************ */
```

```
void calibrate_compass ()                    /* calibrate the compass */
{
   char finished = 0x00;

   PORTDLC = PORTDLC & ~0x04;          /* set !CAL low */
   delay_16msec ();                    /* run a 16ms delay */
   PORTDLC = PORTDLC | 0x04;           /* set nCAL high */

/* we know need to send a command to chassis to rotate 180 degrees
   and zero distance.  Required by the compass calibration
   procedure */

   while(finished != x024)
   {
      send_data_chassis(0x24);       // tell chassis we are sending a move
                                     // command
      delay_16msec();
      send_data_chassis(0);               // upper bits of angle are 0x00
      delay_16msec();
      send_data_chassis(180);        // 0xb4 = 180 degrees
      delay_16msec();
      send_data_chassis(0x00);            // distance is 0x00

       while(!receiving_chassis);         // wait for finished
                                          // acknowledgement from chassis
      finished = recieve_data_chassis();
   }

   PORTDLC = PORTDLC & ~0x04; /* set nCAL low */
   delay_16msec ();           /* run a 16ms delay */
   PORTDLC = PORTDLC | 0x04;  /* set nCAL high */
   delay_16msec ();           /* run a 16ms delay */

/* inform user that compass is calibrated */

   DBug12FNP->printf ("Vector2X is now calibrated \n\r");

}   // end function

/********************** GET HEADING ****************************/

void getheading ()                   /* heading is 9 bits */
{
   char i = 0, j = 0;
   unsigned char heading1 = 0x00;    /* heading info */
   unsigned char heading2 = 0x00;

   PORTDLC = PORTDLC & ~0x01;        /* reset goes low for 16ms */
   delay_16msec ();
   PORTDLC = PORTDLC | 0x01;         /* reset goes back high */
   while (i < 40)                    /* delay at least 500msec */
   {
      delay_16msec ();
      i++;
   }

   robot_heading = 0x0000;           /* set heading to zero degrees */
```

```
   PORTDLC = PORTDLC & ~0x08; /* bring !P/C low bit 0 of PORTB */
   delay_16msec ();                /* !P/C stays low >= 10msec */
   PORTDLC = PORTDLC | 0x08;  /* bring !P/C high after 16msec */

/* wait for EOC to go high, compass has then completed calcs. */

   while ((PORTDLC & 0x02) == 0x00);

/* delay at least 10msec before select compass slave line */

   delay_16msec ();
   PORTS = PORTS & ~0x80;             /* select slave (compass) */

   delay_16msec ();

/* get 1st 8 bits from compass, write garbage to SPO line */

   SP0DR = 0x00;
   while ((SP0SR & 0x80) == 0x00);        /* wait for transfer */

   heading1 = SP0DR;                      /* save in temp variable 1 */

   SP0DR = 0x00;                          /* get 2nd 8 bits from compass */
   while ((SP0SR & 0x80) == 0x00);        /* wait for transfer */
   heading2 = SP0DR;                      /* save in temp variable */

   PORTS = PORTS | 0x80;            /* deselect compass, slave line = 1 */

   if (heading1 & 0x01)             /* add 256 degrees if necessary */
   {
      robot_heading = heading2 + 256;
   }
   else                             /* last 8 bits of heading */
   {
      robot_heading = heading2;
   }

   DBug12FNP->printf ("%3u degrees\n\r", robot_heading);
   send_data_comm (0x10);
   for(j=0; j<10; j++)
   {
      delay_16msec();
   }
   send_data_comm ((char) (robot_heading >> 8));
   for(j=0; j<10; j++)
   {
      delay_16msec();
   }
   send_data_comm ((char) (robot_heading));

}  // end function


/*************************************************************************/
/************************* ACCELEROMETER FUNCTIONS ***********************/
```

```
void get_corr ()              /* start accelerometer function */
{
   int accel_cntr  =  0;      /* initialize counter variable    */
   int total       = 40;      /* take 40 readings for averaging */
   unsigned char j =  0;      /* initialize counter variable    */

   disable();         /* disable interrupts to avoid problems with compass */

   TSCR = 0x80;          /* turn on timer subsystem        */
   TIOS = TIOS & ~0x0A; /* TICs 1 and 3 are input capture */

   TFLG1 = 0x0A;        /* clear flags on TICs 1 and 3    */

   for (accel_cntr = 0; accel_cntr < total; accel_cntr++)
   {

   TCTL4 = (TCTL4 | 0x44) & ~0x88; /* 01000100 - TICs 1 and 3 rising edge*/
   while (!(TFLG1 & 0x02));         /* wait for channel 1 flag to be set*/
   TFLG1 = 0x02;              /* clear channel 1 flag  */
   while (!(TFLG1 & 0x0A));         /* wait for channel 3 flag to be set */
   TFLG1 = 0x0A;              /* clear channel 1 and 3 flags */

   Xrise1 = TC1;              /* read in channel 1 value   */
   Yrise1 = TC3;              /* read in channel 3 value   */

   TCTL4 = (TCTL4 | 0x88) & ~0x44;  /* 10001000 - TICs 1 and 3 falling edge*/
   while (!(TFLG1 & 0x02));         /* wait for channel 1 flag to be set */
   TFLG1 = 0x02;                    /* clear channel 1 flag */
   while (!(TFLG1 & 0x0A));         /* wait for channel 3 flag to be set */
   TFLG1 = 0x0A;                    /* clear channel 1 and 3 flags*/

   Xfall = TC1;                     /* read in channel 1 value */
   Yfall = TC3;                     /* read in channel 3 value */
   TCTL4 = (TCTL4 | 0x44) & ~0x88;   /* 01000100 - TICs 1 and 3 rising edge*/
   while (!(TFLG1 & 0x02));   /* wait for channel 1 flag to be set */
   TFLG1 = 0x02;              /* clear channel 1 flag */
   while (!(TFLG1 & 0x0A));   /* wait for channel 3 flag to be set*/
   TFLG1 = 0x0A;              /* clear channel 1 and 3 flags */

   Xrise2 = TC1;              /* read in channel 1 value */
   Yrise2 = TC3;              /* read in channel 3 value */

   TFLG1 = 0x0A;              /* clear channel 1 and 3 flags */

   Xhigh = Xfall - Xrise1;
   /* X high time = Falling Edge - 1st Rising Edge*/

   Yhigh = Yfall - Yrise1;
   /*Y high time = Falling Edge - 1st Rising Edge*/

   Xtotal = Xrise2 - Xrise1;
   /*X period = 2nd Rising Edge - 1st Rising Edge*/

   Ytotal = Yrise2 - Yrise1;
   /* Y period = 2nd Rising Edge - 1st Rising Edge*/
```

```
   /* X Duty Cycle * 1000 = X high / X period */
   Xduty = ((long) Xhigh) * 1000 / ((long) Xtotal);

   /* Y Duty Cycle * 1000 = Y high / Y period */
   Yduty = ((long) Yhigh) * 1000 / ((long) Ytotal);

   /* Acceleration formulas given in datasheet */
   Accel_X += ((((long) Xduty - 477) * 10000) / 1250);
   Accel_Y += ((((long) Yduty - 528) * 10000) / 1250);

   }        // end for loop

   Accel_X /= total;
   /* Divide Accel_X by number of readings to determine average */

   Accel_Y /= total;
   /* Divide Accel_Y by number of readings to determine average */

/*****************************************************************************/
/* Compare acceleration values read to known acceleration values and
determine*/
/* the tilt angle                                                            */
/*****************************************************************************/

   if (Accel_X >= -20 && Accel_X >= 0)
       { Corr_X = -2; }
   else if (Accel_X >= -45 && Accel_X < -20)
       { Corr_X = -4; }
   else if (Accel_X >= -75 && Accel_X < -45)
       { Corr_X = -6; }
   else if (Accel_X >= -120 && Accel_X < -75)
       { Corr_X = -8; }
   else if (Accel_X >= -150 && Accel_X < -120)
       { Corr_X = -10; }
   else if (Accel_X >= -175 && Accel_X < -150)
       { Corr_X = -12; }
   else if (Accel_X >= -225 && Accel_X < -175)
       { Corr_X = -14; }
   else if (Accel_X >= -255 && Accel_X < -225)
       { Corr_X = -16; }
   else if (Accel_X >= -290 && Accel_X < -255)
       { Corr_X = -18; }
   else if (Accel_X >= -315 && Accel_X < -290)
       { Corr_X = -20; }
   else if (Accel_X >= -345 && Accel_X < -315)
       { Corr_X = -22; }
   else if (Accel_X >= -370 && Accel_X < -345)
       { Corr_X = -24; }
   else if (Accel_X >= -400 && Accel_X < -370)
       { Corr_X = -26; }
   else if (Accel_X >= -430 && Accel_X < -400)
       { Corr_X = -28; }
   else if (Accel_X < -430)
       { Corr_X = -30; }

   if (Accel_X <= 30 && Accel_X >= 0)
       { Corr_X = 0; }
```

```
else if (Accel_X <= 50 && Accel_X > 30)
    { Corr_X = 2; }
else if (Accel_X <= 90 && Accel_X > 50)
    { Corr_X = 4; }
else if (Accel_X <= 130 && Accel_X > 90)
    { Corr_X = 6; }
else if (Accel_X <= 160 && Accel_X > 130)
    { Corr_X = 8; }
else if (Accel_X <= 200 && Accel_X > 160)
    { Corr_X = 10; }
else if (Accel_X <= 230 && Accel_X > 200)
    { Corr_X = 12; }
else if (Accel_X <= 265 && Accel_X > 230)
    { Corr_X = 14; }
else if (Accel_X <= 300 && Accel_X > 265)
    { Corr_X = 16; }
else if (Accel_X <= 330 && Accel_X > 300)
    { Corr_X = 18; }
else if (Accel_X <= 360 && Accel_X > 330)
    { Corr_X = 20; }
else if (Accel_X <= 400 && Accel_X > 360)
    { Corr_X = 22; }
else if (Accel_X <= 440 && Accel_X > 400)
    { Corr_X = 24; }
else if (Accel_X <= 470 && Accel_X > 440)
    { Corr_X = 26; }
else if (Accel_X <= 500 && Accel_X > 470)
    { Corr_X = 28; }
else if (Accel_X > 500)
    { Corr_X = 30; }


if (Accel_Y >= -20 && Accel_Y >= 0)
    { Corr_Y = -2; }
else if (Accel_Y >= -45 && Accel_Y < -20)
    { Corr_Y = -4; }
else if (Accel_Y >= -75 && Accel_Y < -45)
    { Corr_Y = -6; }
else if (Accel_Y >= -120 && Accel_Y < -75)
    { Corr_Y = -8; }
else if (Accel_Y >= -150 && Accel_Y < -120)
    { Corr_Y = -10; }
else if (Accel_Y >= -175 && Accel_Y < -150)
    { Corr_Y = -12; }
else if (Accel_Y >= -225 && Accel_Y < -175)
    { Corr_Y = -14; }
else if (Accel_Y >= -255 && Accel_Y < -225)
    { Corr_Y = -16; }
else if (Accel_Y >= -290 && Accel_Y < -255)
    { Corr_Y = -18; }
else if (Accel_Y >= -315 && Accel_Y < -290)
    { Corr_Y = -20; }
else if (Accel_Y >= -345 && Accel_Y < -315)
    { Corr_Y = -22; }
else if (Accel_Y >= -370 && Accel_Y < -345)
    { Corr_Y = -24; }
else if (Accel_Y >= -400 && Accel_Y < -370)
```

```
        { Corr_Y = -26; }
    else if (Accel_Y >= -430 && Accel_Y < -400)
        { Corr_Y = -28; }
    else if (Accel_Y < -430)
        { Corr_Y = -30; }

    if (Accel_Y <= 30 && Accel_Y >= 0)
        { Corr_Y = 0; }
    else if (Accel_Y <= 50 && Accel_Y > 30)
        { Corr_Y = 2; }
    else if (Accel_Y <= 90 && Accel_Y > 50)
        { Corr_Y = 4; }
    else if (Accel_Y <= 130 && Accel_Y > 90)
        { Corr_Y = 6; }
    else if (Accel_Y <= 160 && Accel_Y > 130)
        { Corr_Y = 8; }
    else if (Accel_Y <= 200 && Accel_Y > 160)
        { Corr_Y = 10; }
    else if (Accel_Y <= 230 && Accel_Y > 200)
        { Corr_Y = 12; }
    else if (Accel_Y <= 265 && Accel_Y > 230)
        { Corr_Y = 14; }
    else if (Accel_Y <= 300 && Accel_Y > 265)
        { Corr_Y = 16; }
    else if (Accel_Y <= 330 && Accel_Y > 300)
        { Corr_Y = 18; }
    else if (Accel_Y <= 360 && Accel_Y > 330)
        { Corr_Y = 20; }
    else if (Accel_Y <= 400 && Accel_Y > 360)
        { Corr_Y = 22; }
    else if (Accel_Y <= 440 && Accel_Y > 400)
        { Corr_Y = 24; }
    else if (Accel_Y <= 470 && Accel_Y > 440)
        { Corr_Y = 26; }
    else if (Accel_Y <= 500 && Accel_Y > 470)
        { Corr_Y = 28; }
    else if (Accel_Y > 500)
        { Corr_Y = 30; }

/*************************************************************************/
/* Use Corr_X and Corr_Y to correct heading error based experimentation    */
/* Send corrected heading information to communications group              */
/*************************************************************************/

    robot_heading = robot_heading + 2*Corr_X;
    robot_heading = robot_heading + 2*Corr_Y;

/* send corrected heading to comm. Group */

    DBug12FNP->printf("Corrected Heading = %d \n\r", robot_heading);

    send_data_comm (0x10);           // ID byte to comm
    for(j=0; j<10; j++)
       {delay_16msec();}

    /*Send upper byte of heading*/
    send_data_comm ((char) (robot_heading >> 8)); for(j=0; j<10; j++)
```

```
      {delay_16msec(); }

   /* Send lower byte of heading */
   send_data_comm ((char) (robot_heading));

   enable();            /* re-enable HC12 interrupts */

}


/*************************************************************************/
/************************* COMMUNICAITION FUNCTIONS ********************/


/*************** COMMUNICATIONS SETUP ******************************/
void comm_init ()
{
   DDRP = 0xff;                  /* all bits of port P are output */
   PORTP = 0x15;                 /* portP initially is 00010101 */

/* this deselects the slave and trigger (all handshake lines low */

   TSCR = 0x80;                  /* enable timer subsystem */
   TIOS = TIOS & ~0x02;          /* 0,2,3,4,5,6,7 to input capture */
   TCTL4 = (TCTL4 | 0x51) & 0x51;/* all capture the rising edge */
   TCTL3 = (TCTL3 | 0x55) & 0x55;
   TMSK1 = TMSK1 | ~0x02;        /* 0,2,3,4,5,6,7  interrupts */
   TMSK2 = 0x01;                 /* overflow rate at 16ms */

/* set up the SPI to communicate with the other subsystems */

   DDRS = DDRS | 0xE0;           /* ss, clk, MOSI outputs */
   PORTS = PORTS | 0x80;         /* deselct slave, for compass */

/* PORTP is !SS for other modules */
/* master device, MSB first, etc, idle high, valid on falling edge */

   SP0CR1 = SP0CR1 | 0x5c;

/* this is set up to match the compass needs */

   SP0CR2 = 0x00;

/* baud rate at 31.3kHz, also, this is
/* fastest the comm module can handle */

   SP0BR = 0x07;

   DBug12FNP->printf("Finished Communications setup \n\r");

}  // end function

   /*****************************************************************/
   /* Setup of the module communications are as follows: ***********/
   /*                                                               */
   /* BALL/HOLE:  slave select: PORTP2                              */
   /*              interrupt: TIMER4                                */
```

```
   /*              handshake: PORTP3                                       */
   /*--------------------------------------------------------------------*/
   /* CHASSIS:     slave select: PORTP0                                  */
   /*              interrupt: TIMER4                                     */
   /*              handshake: PORTP1                                     */
   /*--------------------------------------------------------------------*/
   /* COMMUNICATIONS:      slave select: PORTP4                          */
   /*                      interrupt: TIMER6                             */
   /*                      handshake: not used for communications        */
   /********************************************************************/

/*********************************************************************/
/* SEND TO ball/hole                                                 */
/*********************************************************************/

void send_data_ball_hole (DATA)
{
   char garbage_received = 0x00;
   sending_ball_hole = TRUE;

   delay_16msec ();        /* short delay to make sure evertone is ready */

   PORTP = PORTP & ~0x14;        /* slave select both ball/hole and comm */
   SP0DR = DATA;                 /* send 8-bit (char) average */
   while ((SP0SR & 0x80) == 0);  /* wait for transfer */
   garbage_received = SP0DR;
   PORTP = PORTP | 0x14;         /* this line slave deselects both the
                                    ball/hole and comm channels */

   DBug12FNP->printf("Sent %x \n\r", DATA);

   ready_ball_hole = FALSE;
   sending_ball_hole = FALSE;
}

/*********************************************************************/
/* RECEIVE FROM ball/hole                                            */
/*********************************************************************/

char receive_data_ball_hole ()      /* receive from ball/hole module */
{
   char RECEIVED = 0x00;            /* initialize variables */

   receiving_ball_hole = TRUE;
   PORTP = PORTP & ~0x04;     /* select slave line ports0 */
   SP0DR = GARBAGE;           /* send 8-bit (char) average */
   while ((SP0SR & 0x80) == 0x00);  /* wait for transfer */
   RECEIVED = SP0DR;          /* save the received data */
   PORTP = PORTP | 0x04;            /* deselect SLAVE */

   receiving_ball_hole = FALSE;

   D(DBug12FNP->printf("Receiving \n\r");)

   return RECEIVED;           /* return the received data */
}  // end function
```

```
/*********************************************************************/
/* SEND TO comm                                                      */
/*********************************************************************/

void send_data_comm (DATA)
{
   char garbage_received = 0x00;
   sending_comm = TRUE;

   PORTP = PORTP & ~0x10;           // slave select comm module
   SP0DR = DATA;              // load data into SPI
   while ((SP0SR & 0x80) == 0);          // wait for transfer to finish
   garbage_received = SP0DR;        // clear the SPI flag by reading garbage
   PORTP = PORTP | 0x10;            // deselect comm module

   ready_comm = FALSE;        // finished sending to comm, so clear the flags
to FALSE
   sending_comm = FALSE;
}

/*********************************************************************/
/* RECEIVE FROM comm                                                 */
/*********************************************************************/

char receive_data_comm (void)
{
   char RECEIVED = 0;

   PORTP = PORTP & ~0x10;           /* select comm module */
   SP0DR = GARBAGE;                 /* send garbage to start Xfer */
   while ((SP0SR & 0x80) == 0x00);        /* wait for transfer to finish */
   RECEIVED = SP0DR;                      /* store the received data */
   PORTP = PORTP | 0x10;                  /* deselect comm module */

   receiving_comm = FALSE;                /* clear the flag */
   return RECEIVED;                       /* return received data */
}

/*********************************************************************/
/* SEND TO chassis                                                   */
/*********************************************************************/

void send_data_chassis (DATA)
{
   char garbage_received = 0x00;


   DBug12FNP->printf("in send_data_chassis\r\n");
   sending_chassis = TRUE;

   PORTP = PORTP | 0x02;      /* set interrupt trigger line to chassis */
   while (!ready_chassis);    /* wait for the acknowledgement */
   PORTP = PORTP & ~0x02;     /* drop the handshake line */
   DBug12FNP->printf("out of handshake\r\n");

   delay_16msec ();
```

```
   PORTP = PORTP & ~0x11;      /* select both chassis and comm */

   SP0DR = DATA;                /* load data into SPI data register, and
                                   start the transfer */
   while ((SP0SR & 0x80) == 0);          // wait for transfer to finish
   garbage_received = SP0DR;  // read the garbage to clear the SPI flag
   PORTP = PORTP | 0x11;           // deselect chassis and comm

   ready_chassis = FALSE;        // clear the transmission flags
   sending_chassis = FALSE;
}

/*****************************************************************/
/* RECEIVE FROM chassis                                          */
/*****************************************************************/

char receive_data_chassis (void)
{
   char RECEIVED = 0;

   PORTP = PORTP & ~0x01;        // slave select the chassis module
   SP0DR = GARBAGE;              // send garbage to start Xfer
   while ((SP0SR & 0x80) == 0x00);       // wait for Xfer
   RECEIVED = SP0DR;             // store received data
   PORTP = PORTP | 0x01;         // deselect the slave device

   receiving_chassis = FALSE;    // clear the transmission flags
   return RECEIVED;
}

/*****************************************************************/
/* INTERRUPT SERVICE ROUTINES                                    */
/*****************************************************************/

@interrupt tic0_isr ()        /* dedicated STOP/PAUSE line from comm */
{
   while ((PORTT & 0x01) == 0x00);  // do nothing while line is high
   TFLG1 = 0x01;                    // clear the flag
}

/* CHANNEL 2 INTERRUPT */
@interrupt tic2_isr (void)
{
   if (sending_chassis) ready_chassis = TRUE;   // set ready if sending
   else receiving_chassis = TRUE;               // we are receiving
   TFLG1 = 0x04;                                // clear flag
}

/* CHANNEL 4 INTERRUPT */
@interrupt tic4_isr (void)
{
   if (sending_ball_hole) ready_ball_hole = TRUE; // set ready if sending
   else receiving_ball_hole = TRUE;               // we are receiving
   TFLG1 = 0x10;                                  // clear flag
}

/* CHANNEL 6 INTERRUPT */
```

```
@interrupt tic6_isr (void)
{
   if (sending_comm) ready_comm = TRUE;         // set ready if sending
   else receiving_comm = TRUE;                  // we are receiving
   TFLG1 = 0x40;                                // clear flag
}

/* nulls for safety... and sanity */
@interrupt tic7_isr () { TFLG1 = 0x80; }
@interrupt tic5_isr () { TFLG1 = 0x20; }
@interrupt tic3_isr () { TFLG1 = 0x08; }


/*********************************************************************/
/* FIND BALL FNCN                                                    */
/*************************************************FS******************/

void find_ball ()
{
   DBug12FNP->printf("in ball\r\n");
   enable ();
   send_data_ball_hole (0x21);
   RECEIVED = GARBAGE;
   while (RECEIVED != 0xff)
   {
      if (receiving_ball_hole)
      {
         RECEIVED = receive_data_ball_hole ();
         DBug12FNP->printf ("Received %x from B/H\n\r", RECEIVED);
         send_data_chassis (0x21);
         send_data_chassis (RECEIVED);
         DBug12FNP->printf ("Sent %x to chassis\n\r", RECEIVED);
      }
   }

   disable ();   // disable the HC12 interrupts

   DBug12FNP->printf("exit ball\r\n");

}                       // end find ball function

/*********************************************************************/
/* FIND HOLE FUNCTION                                                */
/*********************************************************************/

void find_hole ()
{
   DBug12FNP->printf("in find_hole\r\n");
   enable ();
   send_data_ball_hole (0x20);
   RECEIVED = GARBAGE;
   while (RECEIVED != 0xff)
   {
      if (receiving_ball_hole)
      {
          RECEIVED = receive_data_ball_hole ();
          DBug12FNP->printf ("Received %x from B/H \n\r", RECEIVED);
```

```
            send_data_chassis (0x20);
            send_data_chassis (RECEIVED);
            DBug12FNP->printf ("Sent %x to chassis\n\r", RECEIVED);
         }
      }

      disable ();

      DBug12FNP->printf("out of find_hole\r\n");

}           // end find hole function

/*********************************************************************/
/* MOVE FUNCTION                                                     */
/*********************************************************************/

/* chassis comm wrapper: ident:angle:angle:dist */

void move (int angle, int distance)
{
   enable();                          // enable interrupts
   not_there = TRUE;                  // set that we are not there
   send_data_chassis(0x24); delay_16msec();     // send ID byte
   send_data_chassis((char)(angle>>8)); delay_16msec(); // upper of angle
   send_data_chassis((char)(angle)); delay_16msec();  // lower of angle
   send_data_chassis((char)distance);           // send distance
   while(!receiving_chassis);              // wait for confirming transmission
   RECEIVED = receive_data_chassis();
   DBug12FNP->printf("got: 0x%x\r\n", RECEIVED);

   while(not_there)           // do this loop while not at ball/hole
   {
      get_position();               // GPS postion and compass heading
      send_data_chassis(/* stuff for movements (GPSDir, GPSDist) */);
      delay_3sec();                 // delay before repeating
   }
   not_there = FALSE;

}           // end move function


/********** GENERIC DELAY FOR 3 SEC AND 16MSEC **********************/

void delay_3sec ()
{
   char i = 0;                       /* set counter to 0 */
   TFLG2 = 0x80;                     /* clear timer overflow flag */
   while (i < 188)                   /* wait for 118 16ms overflows */
   {
       while (!(TFLG2 & 0x80));      /* wait for timer flag */
       TFLG2 = 0x80;                 /* clear timer interrupt flag */
       i++;
   }
}

void delay_16msec ()          /* set up delay for calibration */
{
```

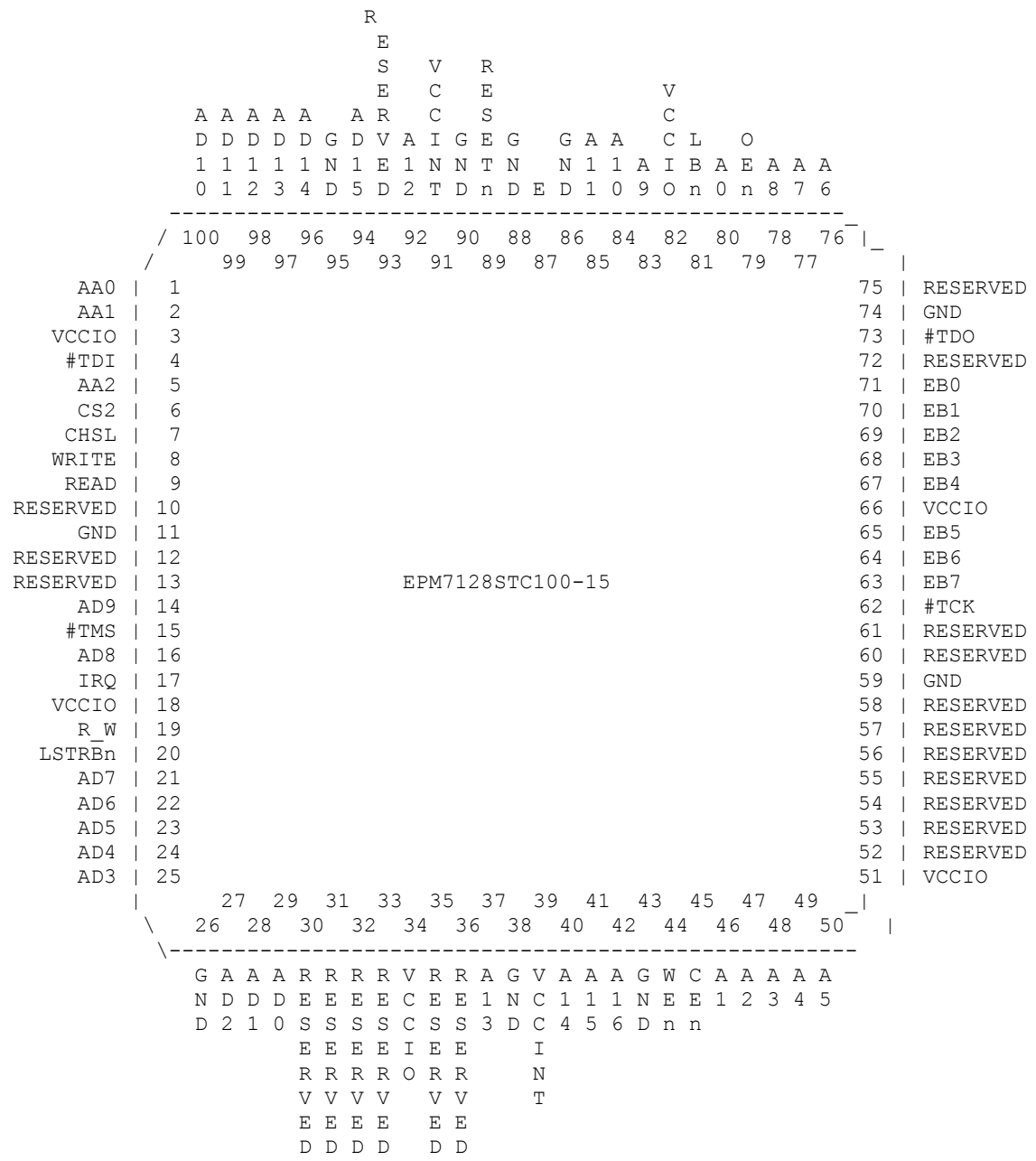
```
   TFLG2 = 0x80;              /* clear timer interrupt flag */
   TSCR = 0x80;
   while (!(TFLG2 & 0x80));   /* wait for timer flag */
   TFLG2 = 0x80;              /* clear timer interrupt flag */
}

/*****************************************************************/
/* MAIN                                                          */
/*****************************************************************/

void main (void)
{
   int holex, holey;          // target for the hole coords

/* setup all the components ***************************** */

   setupuart1 ();             // setup the UART
   setup_compass ();          // setup the compass
   comm_init ();              // setup the communications protocols
   calibrate_compass();       // run the calibrate compass function

   #ifdef gpstest             // run this if conducting a GPS test
      for(;;) {get_position(); getheading(); get_corr(); gpsDir(); gpsDist();
      delay_3sec();}
   #endif

/* end setup calls************************************** */

/* receive coordinates and start command ********************** */

   enable();

   send_data_comm(0x55);
   DBug12FNP->printf ("Sent a 0x55 to comm, waiting to receive \n\r");

   while(!receiving_comm);    // must wait for comm to send the flag high
                              // (set in TIC4 ISR)
   GPSxTAR = receive_data_comm() << 8;
   // receive and 8-bit transfer, upper bit set is shifted by 8 to the left

   while(!receiving_comm);
   // wait cycle for comm. flag must be between each transfer for timing

   GPSxTAR += receive_data_comm ();
   DBug12FNP->printf (" %x \n\r", GPSxTAR);

   while(!receiving_comm);
   GPSyTAR = receive_data_comm () << 8;

   while(!receiving_comm);
   GPSyTAR += receive_data_comm ();
   DBug12FNP->printf (" %x \n\r", GPSyTAR);

   while(!receiving_comm);
   holex = receive_data_comm () << 8;

   while(!receiving_comm);
```

```
   holex += receive_data_comm ();
   DBug12FNP->printf (" %x \n\r", holex);

   while(!receiving_comm);
   holey = receive_data_comm () << 8;

   while(!receiving_comm);
   holey += receive_data_comm ();
   DBug12FNP->printf (" %x \n\r", holey);

   DBug12FNP->printf ("Ball Tar-get: x:%d y:%d \r\nHole Tar-get: x:%d
      y:%d\r\n", GPSxTAR, GPSyTAR, holex, holey);
   DBug12FNP->printf ("Finished receiving the coords from comm \n\r");

/* this code setup uses the transmission of coords as a start command from
comm. */

   disable();           // disable HC12 interrupts

   delay_3sec();}             // run a delay before starting cycles

   ball_close = false;
   delay_3sec(); //delay_3sec();          // 6 sec delay in program

   while (!ball_close)
   {
      get_position ();               /* these 2 take a while */
      getheading ();
      ball_close = gpsDist ();
      delay_3sec();

      if (!ball_close)
      {
            int temp = gpsDir ();
            if (guess < 60)
                  move (temp, guess);   // walk this way!
            else
                  move (temp, guess / 2);     // walk this way!
      }

   }

   DBug12FNP->printf("we're close\r\n");

   find_ball ();                    // call this function to find the ball

   DBug12FNP->printf("out of find_ball: delay before starting !hole close
      routine... \n\r");

   delay_3sec();

   GPSxTAR = holex;                 // change target to find hole now
   GPSyTAR = holey;

   hole_close = false;
   while (!hole_close)
   {
```

```
      get_position ();
      getheading ();
      hole_close = gpsDist ();
      if (!hole_close)
      {
            int temp = gpsDir ();
            if (guess < 60)
                  move (temp, guess);           /* walk this way! */
            else
                  move (temp, guess / 2);       /* walk this way! */
      }
   }

   DBug12FNP->printf("we're close\r\n");

   find_hole ();    // call this function when we are finding the hol

   DBug12FNP->printf("We're done!!\r\n");
   _asm("swi");
}
```

Appendix C: Altera Code

This appendix contains the Altera code used for the memory expansion board and the added UART chip. Also included is the Altera pin-out form the report file



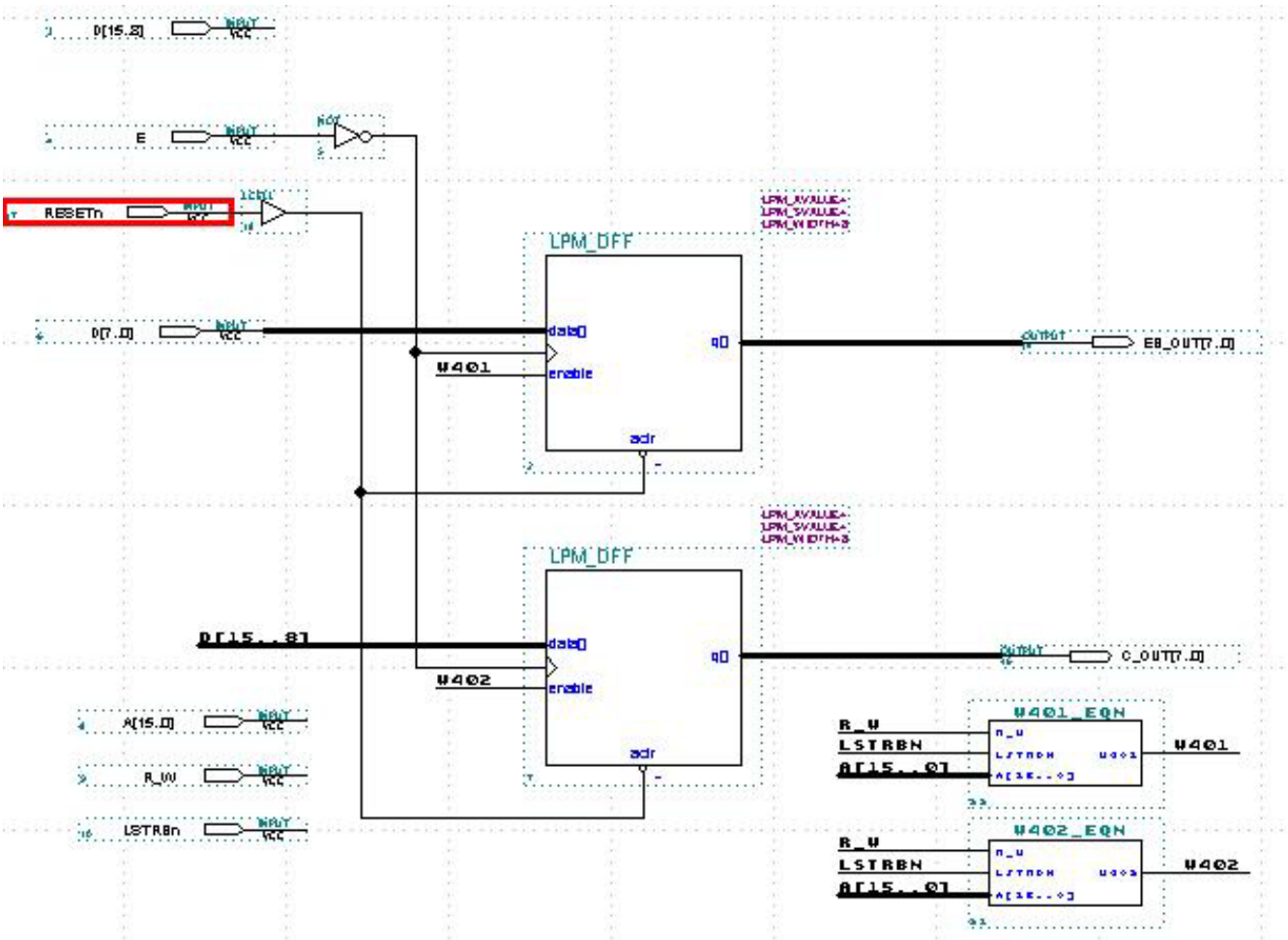
% ********** functions for read/write to addresses 401 and 402******** %

```
SUBDESIGN w402_eqn
(
   R_W               :  INPUT;   % R/W Line, LSTRBn %
   LSTRBn            :  INPUT;

   A[15..0]          :  INPUT;  % Demultiplexed address bits %

   w402              :  OUTPUT;
)
```

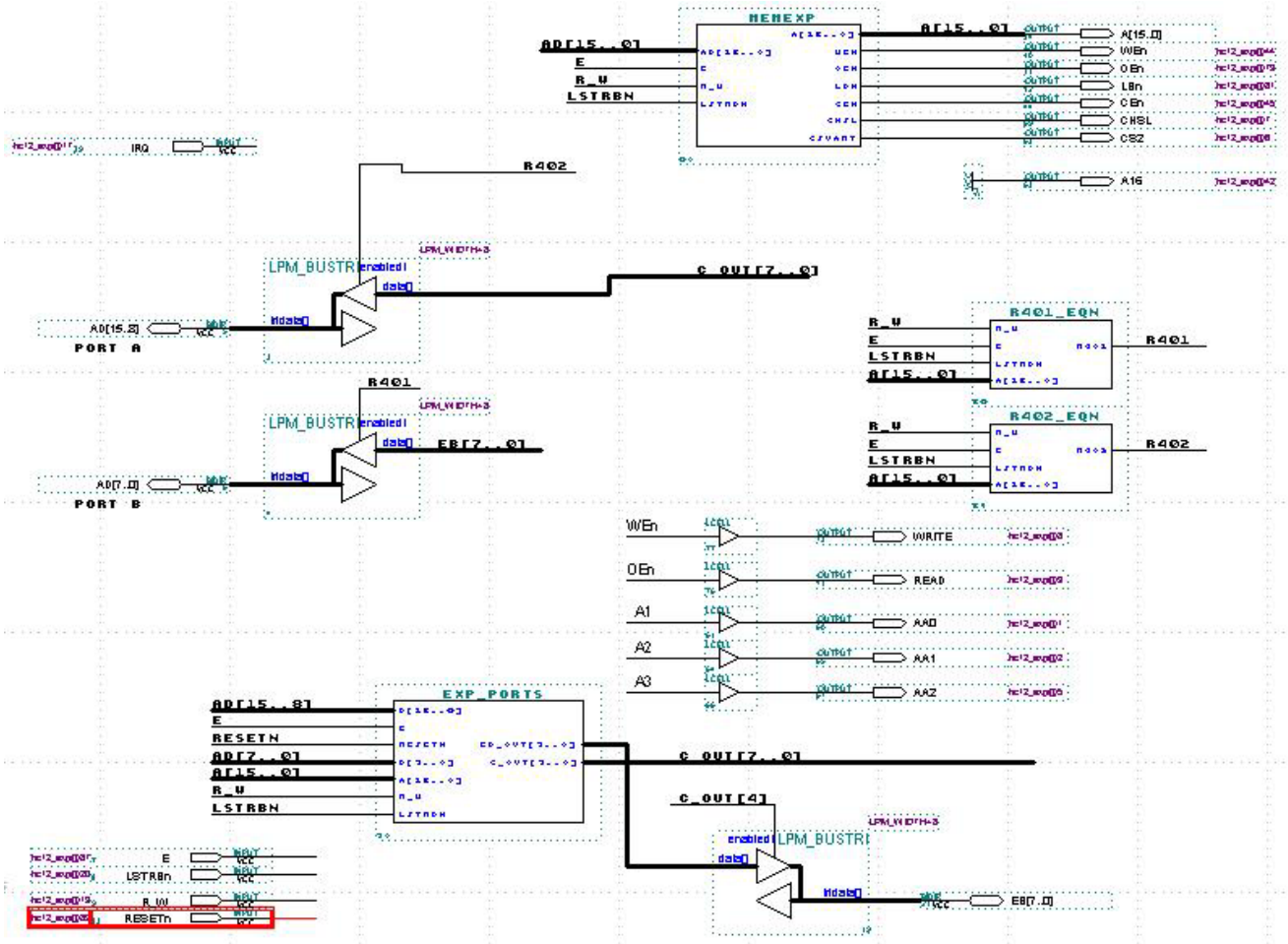
```
BEGIN
   % CHECK CONDITIONS %
   if ((A[15..0] == H"0402") & (R_W == GND)) THEN
         w402 = VCC;       % SET LINE HIGH %
   ELSE
         w402 = GND;       % SET LINE LOW %
   END IF;

END;


SUBDESIGN w401_eqn
(
   R_W               :  INPUT;   % R/W Line %
   LSTRBn            :  INPUT;

   A[15..0]          :  INPUT;  % Demultiplexed address bits %

   w401              :  OUTPUT;
)


BEGIN

   if (((A[15..0] == H"0401") OR (A[15..0] == H"0400")) & (LSTRBn == GND)
         & (R_W == GND)) THEN
         w401 = VCC;
   ELSE
         w401 = GND;
   END IF;

END;


SUBDESIGN r402_eqn
(
   R_W               :  INPUT;   % R/W Line %
   E                 :  INPUT;
   LSTRBn            :  INPUT;

   A[15..0]          :  INPUT;  % Demultiplexed address bits %

   r402              :  OUTPUT;
   )


BEGIN

   if ((A[15..0] == H"0402") & (R_W == VCC) & (E == VCC)) THEN
         r402 = VCC;
   ELSE
         r402 = GND;
   END IF;

END;
```

```
SUBDESIGN r401_eqn
(
   R_W               :  INPUT;   % R/W Line, E, LSTRBn %
   E                 :  INPUT;
   LSTRBn            :  INPUT;

   A[15..0]          :  INPUT;  % Demultiplexed address bits %

   r401              :  OUTPUT;
   )


BEGIN
         % CHECK CONDITIONS FOR ODD BYTE READ %
   if (((A[15..0] == H"0401") OR (A[15..0] == H"0400")) & (LSTRBn == GND)
         & (R_W == VCC) & (E == VCC)) THEN
         r401 = VCC;             % SET READ HIGH %

   ELSE
         r401 = GND;             % SET READ LOW %
   END IF;

END;
```

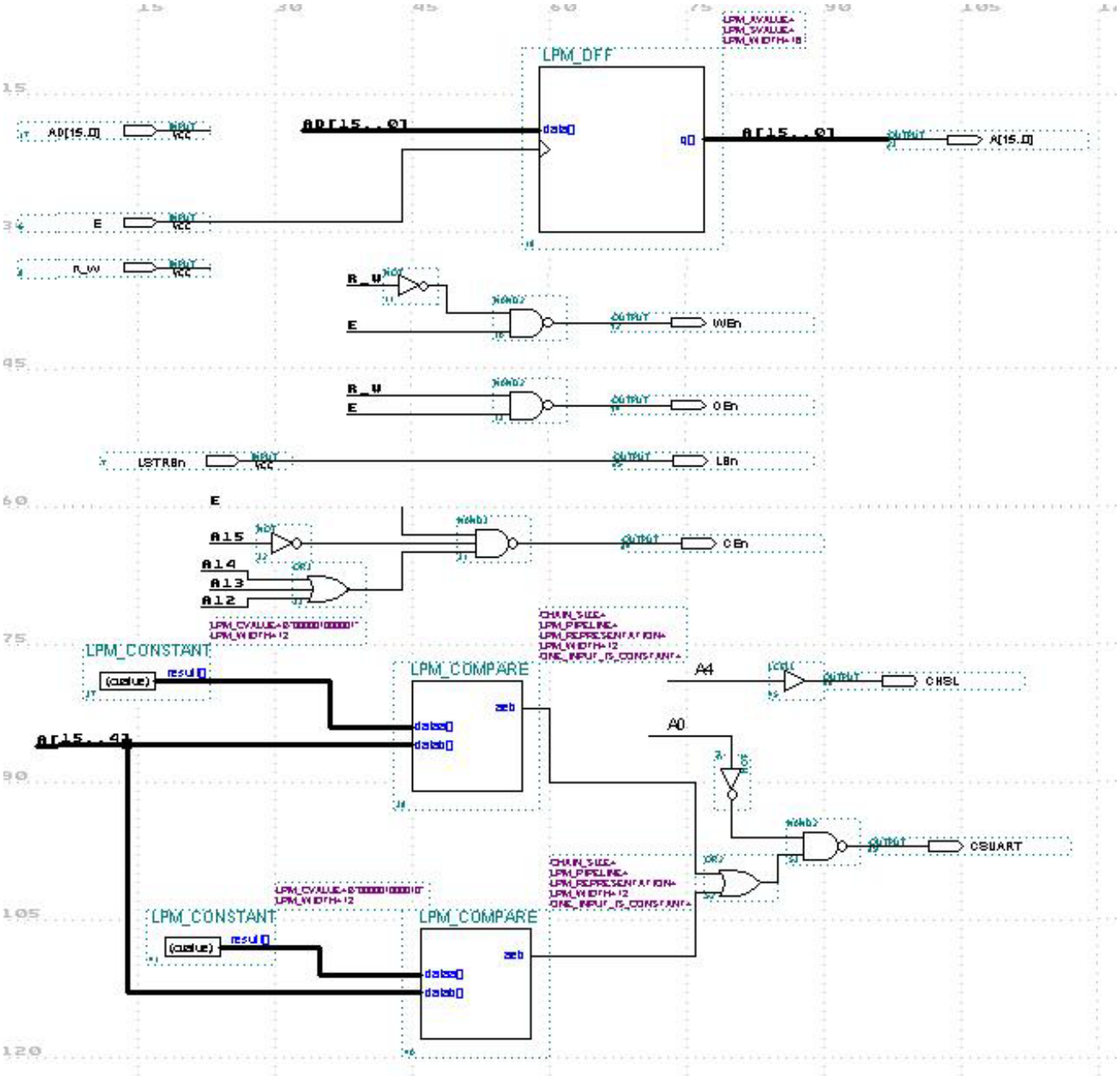
EXP_PORTS.GDF



HC12_EXP.GDF


MEMEXP
AD[15..0]
E
R_W
LSTRBN
A[15..0]
A[15..0]
WEn
OEn
LBn
CEn
CHSL
CS2
A16
IRQ
R402
LPM_BUSTR
enabled
data[]
Mdata[]
C_OUT[7..0]
AD[15..8]
PORT A
R401
EB[7..0]
AD[7..0]
PORT B
R401_EQN
R402_EQN
WEn
OEn
A1
A2
A3
WRITE
READ
AA0
AA1
AA2
EXP_PORTS
RESETn
C_OUT[4]
E
LSTRBn
R_W
RESETn
EB[7..0]

MEMEXP.GDF


LPM_DFF
AD[15..0]
A[15..0]
E
R_W
WEn
OEn
LSTRBn
LBn
A15
A14
A13
A12
CBn
LPM_CONSTANT
LPM_COMPARE
A4
CHSL
A0
CSUART

Appendix D: References

This section contains links to the various data sheets and manuals that we used for our hardware this semester. They are provided for a quick reference, and we do not make any claims to the material contained in them.

The data sheets and manuals themselves have been included on the CD accompanying this report.

Garmin GPS16

GPS16 Manual: http://www.garmin.com/manuals/gps16qsg.pdf

PNI Vector 2X

Vector 2x Manual: http://www.precisionnav.com/technical-information/pdf/vector-2x.pdf

Analog Devices ADXL202JQC

ADXL202JQC Data Sheet: http://www.analog.com/productSelection/pdf/ADXL202_10_b.pdf

National Semiconductor UART

PC16552D Data Sheet: http://www.national.com/ds/NS/NS16C552.pdf

ECS HP-49U Crystal

HP-49U Datasheet: http://www.ecsxtal.com/pdf2/HC49U.PDF